THE ALCHEMY OF RECONFIGURABILITY

DESIGN AND VERIFICATION OF A RECONFIGURABLE PROCESSOR ARCHITECTURE
THE ALCHEMY OF RECONFIGURABILITY

DESIGN AND VERIFICATION OF A RECONFIGURABLE PROCESSOR ARCHITECTURE

ADVISOR: DIÓGENES CECÍLIO DA SILVA JR.

Belo Horizonte
September 2015
alchemy
Alchemy and Transmute Toolchain can be downloaded at:
https://github.com/caioso/Alchemy
https://github.com/caioso/Transmute
Abstract

The quick advances in technology landscape and the fast increase of application complexity is making it mandatory for computer systems to provide some level of adaptability. Adaptability can be implemented in multiple abstractions levels such as circuit layout, micro-architecture, and runtime software. Digital circuits could take a special advantage of this adaptability with the advent of FPGA. They brought up the possibility of hard-wired circuits to assume different configurations according to the developer’s requirements, providing application-specific computations on a standard off-the-shelf device. In most cases, low-end computer systems cannot afford to integrate high-end ASIC circuits to implement their application-specific operations, due to prohibitive costs. They can recur to reconfigurable logic instead. This allows these systems to attain higher performance for a broad range of application domains. The embedded system becomes truly flexible in the sense that users can specify new hardware/software designs for each applications. However a scoop of challenges arise with this architectural approach - complex software/hardware workload partition, cross-domain implementation and costly debug. This document introduces Alchemy, a reconfigurable low-end embedded processor architectures, which integrates a CPU and an FPGA in the same die. The CPU is based on MSP430 ISA, including five new instructions. The FPGA is based on Xilinx XC2000 architecture and a new low-level cross-domain communication controller called BRIDGE was created. The memory architecture and other minor architecture elements are also presented. The architecture is detailed with its Compilation/Synthesis toolchain and Architecture tests.
ABSTRACT
Resumo

O rápido avanço no panorama tecnológico e o rápido crescimento na complexidade de aplicações tem tornado obrigatório que sistemas computacionais provenham algum nível de adaptabilidade. A adaptabilidade pode ser implementada em múltiplos níveis de abstração como no layout de circuitos, na micro-arquitetura, e em software. Circuitos digitais puderam tirar vantagem da adaptabilidade com o advento do FPGA. Esses dispositivos trouxeram a possibilidade de circuitos assumirem diferentes configurações de acordo com os requisitos do desenvolvedor, provendo computações específicas de aplicação em um dispositivo comum encontrado no mercado.

Na maioria dos casos, sistemas computacionais de baixo custo não podem se dar ao luxo de integrar caros aceleradores em hardware (ou ASICs) para implementar suas operações específicas de aplicação, devido a custos proibitivos de desenvolvimento. Pode-se recorrer a dispositivos reconfiguráveis como uma solução mais barata a ASICs. Esses dispositivos permitem que o sistema obtenha melhor desempenho em uma ampla gama de domínios de aplicação. Sistemas embutidos se tornam flexíveis no sentido de que usuários podem especificar novos designs de hardware/software para cada aplicação. Entretanto, uma grande quantidade de desafios surge com esta nova abordagem arquitetural como uma complexa partição da carga de trabalho entre software/hardware, implementação de aplicações em múltiplos domínios e depuração dispendiosa.

Este documento apresenta Alchemy, uma arquitetura de processador embutido reconfigurável de baixo custo, que integra uma CPU e um FPGA na mesma die. A CPU é baseada na ISA do MSP430, incluindo cinco novas instruções. O FPGA é baseado na arquitetura Xilinx XC2000 e criou-se um novo controlador de comunicação chamado BRIDGE. A arquitetura de memória e outros elementos também são apresentados. A arquitetura é detalhada com suas ferramentas de Compilação/Síntese, além de seus testes.
## Contents

Abstract

Resumo

1 Introduction 7
   1.1 Motivation ................................................. 7
   1.2 Objectives ............................................... 8
   1.3 Organization .............................................. 8

2 Literature Review 11
   2.1 Reconfigurable Processor Taxonomy ........................ 13
   2.2 Reconfigurable Architectures Structural Classification .... 14
   2.3 Reconfigurable Application Design Approaches ............... 16
   2.4 Featured Reconfigurable Processor Architectures .......... 16
      2.4.1 PRISM Architecture .................................... 17
      2.4.2 Nano Processor ......................................... 18
      2.4.3 Garp Architecture ..................................... 20
      2.4.4 MOLEN Processor ....................................... 21
      2.4.5 DySER Architecture .................................... 22

3 Alchemy Architecture 23
   3.1 Alchemy CPU Architecture .................................. 26
      3.1.1 CPU Modes ............................................. 26
      3.1.2 CPU Data Model ......................................... 27
      3.1.3 CPU Registers .......................................... 28
      3.1.4 Endianness .............................................. 30
      3.1.5 Addressing Modes ....................................... 31
      3.1.6 Alchemy Instruction Types ............................... 32
      3.1.7 Instruction Decoder .................................... 35
      3.1.8 Arithmetic and Logic Unit ............................... 36
      3.1.9 Control Unit ........................................... 37
   3.2 Alchemy FPGA Architecture .................................. 42
      3.2.1 FPGA Matrix Organization ............................... 43
      3.2.2 FPGA Modes ............................................. 44
      3.2.3 FPGA Signal Model ..................................... 45
      3.2.4 FPGA Coordinate Systems ............................... 46
      3.2.5 Configurable Logic Block ............................... 47
         I/O Interfaces ........................................... 48
CONTENTS

Look-Up Tables ........................................... 49
CLB Register .............................................. 51
3.2.6 FPGA Interconnects .................................. 51
3.2.7 Switch Matrix ....................................... 53
3.2.8 I/O Blocks ........................................... 55
3.2.9 FPGA Configuration Bitstream ......................... 55
3.3 BRIDGE .................................................. 59
3.3.1 BRIDGE Protocol ..................................... 60
3.3.2 BRIDGE Schedule Register ........................... 63
3.3.3 BRIDGE Control and Monitoring Unit .................. 63
3.3.4 BRIDGE Selection Plane .............................. 65
3.3.5 BRIDGE Register Bank ............................... 66
3.3.6 BRIDGE Memory Interface Unit ....................... 67
3.4 Memory Organization ................................... 68
3.5 Minor Architecture Elements ............................ 70
3.5.1 Interrupt Controller ................................. 70
3.5.2 Configuration Controller ............................. 71
3.5.3 System Busses ....................................... 72
Local Busses ............................................... 73
Global Bus ................................................. 73
4 Alchemy Applications ..................................... 75
4.1 Alchemy Application Organization ......................... 75
4.1.1 Alchemy Cross-Procedures ......................... 76
4.2 Alchemy Programming Model ............................. 77
4.2.1 Alchemy Flows ....................................... 77
4.2.2 Flows Life-cycle ..................................... 78
4.2.3 Shared-memory Flowed environment .................... 80
4.2.4 Synchronization with Flows .......................... 80
4.2.5 Considerations ....................................... 81
4.3 Alchemy High-Level Development Languages ............... 82
4.3.1 C Non-Standard Language Statements .................. 82
    Cross-Procedure Call Request Statements ................. 82
    Cross-Procedure Synchronization Barrier .................. 83
    Cross-Procedure Scheduling Statement .................... 84
    Cross-Procedure Abortion Statement ...................... 84
    #reach Acknowledgement Statement ........................ 85
4.3.2 Verilog Non-Standard Language Statements .......... 85
    Clock Signal ........................................... 85
    Return Signal ......................................... 86
    Memory Manipulation Conventions ......................... 86
    ‘reach Acknowledgement Statement ....................... 87
4.4 Alchemy Custom Assembly Structures ...................... 87
5 Transmute Toolchain ...................................... 91
5.1 Transmute Toolchain .................................... 91
5.2 Transmute Compilation and Synthesis Flow ................. 92
5.2.1 Source Code Preprocessing with Alkahest ............. 94
5.2.2 Verilog Module Synthesis, Place and Route with VTR .... 97
## CONTENTS

5.2.3 Device Packing with Azoth ............................................. 98
5.2.4 FPGA Cross-Procedure call request emission with Archeus ............. 100
5.2.5 C Programs Compilation, Assembly and Linking with GCC ............... 101
5.2.6 Memory Patching with Archeus ........................................ 102

6 Architecture Implementation and Validation .................................. 103
  6.1 System-C Transaction Level Model ....................................... 103
  6.1.1 Implementation ......................................................... 104
  6.2 Alchemy Register-Transfer Level Model .................................... 104
  6.2.1 Implementation ......................................................... 105
  6.3 Architecture Validation .................................................. 105
  6.4 Dynamic Property Checking .............................................. 106
  6.5 Exhaustive Verification .................................................. 109
    6.5.1 Test Vector Generation ............................................ 111
    6.5.2 Exhaustive Verification in CPU and FPGA ......................... 113
    6.5.3 Exhaustive Verification in BRIDGE ................................ 115
  6.6 Validation Considerations .............................................. 115

7 Architecture Evaluation ..................................................... 117
  7.1 Architecture Benchmarking Tests ........................................ 117
  7.2 Alchemy and Wireless Sensor Network .................................... 119

8 Conclusions ........................................................................ 123
  8.1 Discussion ................................................................. 123
  8.2 Future Works ............................................................. 125
    8.2.1 Improve FPGA Matrix Size ....................................... 125
    8.2.2 CPU Instruction Override ........................................ 125
    8.2.3 FPGA in Machine-Cycle .......................................... 126
    8.2.4 BRIDGE Watch Functionality .................................... 126

9 References ........................................................................ 127
## List of Figures

1. Binary number template. .................................................. 4

2.1 Reconfigurable processor design space evolution [Chattopadhyay, 2013]. ... 13
2.2 Reconfigurable Structural Classification [25] (modified). ....................... 15

3.1 Simplified Alchemy block diagram. The most relevant architecture components are depicted and labeled from 1 to 14. .......................................... 24
3.2 Alchemy CPU simplified Architecture. ........................................ 26
3.3 Alchemy Register File Structure. ........................................... 28
3.4 Alchemy Register File Registers. ............................................ 29
3.5 Little-endian memory organization model. ...................................... 31
3.6 Alchemy Instruction Fields. .................................................. 33
3.7 Alchemy Instruction Decoder. ............................................... 35
3.8 Alchemy ALU. ................................................................. 37
3.9 Alchemy Control Unit Finite State Machine. ................................... 38
3.10 Alchemy FPGA architecture. ................................................. 42
3.11 Alchemy Island-Style FPGA matrix. ......................................... 43
3.12 Alchemy FPGA Coordinate System in a sample 2-by-2 FPGA Matrix. .......... 46
3.13 Alchemy Configurable Logic Block architecture. ............................ 47
3.14 Alchemy Configurable Logic Block detailed architecture. .................... 48
3.15 Four-Input LUT setup. ...................................................... 49
3.16 Independent Three-Input LUT setup. ....................................... 50
3.17 Dynamically Selectable Three-Input LUT setup. .............................. 50
3.18 CLB Register. Input D is the synchronous data input, input S is the asynchronous set input, input R is the asynchronous clear input and Q is the register output. 51
3.19 General Interconnect design, regardless its domain. .......................... 52
3.20 Channel basic organization. ................................................ 53
3.21 Alchemy Subset Switch Matrix architecture. ................................... 53
3.22 Alchemy Switch Box architecture. .......................................... 54
3.23 I/O Block general structure. ............................................... 55
3.24 Configuration Bitstream logical organization. .................................. 56
3.25 Switch Matrices configuration Bitstream. ..................................... 57
3.26 Interconnect Configuration Bitstream. ....................................... 57
3.27 Configurable Logic Block Configuration Bitstream. .......................... 58
3.28 I/O Blocks Configuration Bitstream. ....................................... 58
3.29 BRIDGE internal structure. ................................................ 59
LIST OF FIGURES

3.30 BRIDGE Protocol timing diagram. 1: Full transaction with CPU as requester and FPGA as receiver. 2: Fill transaction with FPGA as requester and CPU as receiver. 3: Full memory read transaction from the FPGA. 4: Full memory write transaction performed by the FPGA. ........................................... 62
3.31 BRIDGE Control Unit state machine. .......................................................... 63
3.32 Monitoring Unit state machine. ..................................................................... 65
3.33 BRIDGE Bit internal organization. ................................................................. 66
3.34 Memory Block and its Bank architecture. ......................................................... 68
3.35 MMU and Memory Blocks organization. ........................................................ 70
3.36 Interrupt Controller internal structure. .......................................................... 71
3.37 Configuration Controller finite state machine. ................................................. 71

4.1 Alchemy Application Organization .................................................................. 76
4.2 Flow life-cycle diagram. .................................................................................. 79
4.3 Manual Synchronization method in Verilog Modules. ....................................... 81
4.4 Call Request sample usage. ............................................................................ 83
4.5 @wait statement usage. ................................................................................... 83
4.6 @stall statement usage. ................................................................................... 84
4.7 @destroy statement usage. ............................................................................. 84
4.8 #reach statement usage. ................................................................................ 85
4.9 clk signal declaration and usage on always block. ........................................... 85
4.10 return signal declaration and usage on always block. ..................................... 86
4.11 Memory busses declaration. .......................................................................... 86
4.12 Memory management signals declaration. ...................................................... 87
4.13 Memory handling state machines, written in Verilog. The addresses and the content to be written in memory are placeholders. ................................. 88
4.14 ‘reach keyword usage. .................................................................................. 88

5.2 Transmute command examples. ................................................................. 92
5.1 Transmute Toolchain Compilation/Synthesis Flow. ....................................... 93
5.3 Alkahest parameter-to-pin relationship. ........................................................ 95
5.4 GET structure and usage throughout an FPGA Cross-Procedure call cycle. ...... 96
5.5 VTR execution flow diagram. ...................................................................... 97
5.6 Archeus Cross-Procedure call request emission code. ................................. 100
5.7 MSP430-GCC compilation flow. ................................................................. 101

6.1 Alchemy Formal Verification Flow Diagram. ............................................ 107
6.2 Exhaustive Verification Test setup. .............................................................. 111
6.3 Aether user interface. .................................................................................... 113
6.4 Azure CPU tests user interface. ................................................................. 114
List of Tables

2.1 Reconfigurable Architectures classified according to The axis of Figure 2.1 . . . 17

3.1 CPU modes and corresponding State Controller encoding. . . . . . . . . . . . . 27
3.2 Control Unit Instruction Execution modes timing . . . . . . . . . . . . . . . . 41
3.3 CPU operations Timing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
3.4 FPGA modes and corresponding State Controller encoding. . . . . . . . . . . . 45

5.1 Parameter size related to pin size. . . . . . . . . . . . . . . . . . . . . . . . . 95

7.1 Test results and Speed-Up . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 118
Document Conventions

The following conventions are used throughout this document. The reader shall consider this a consultation section and refer back to it when further explanations are required.

Commonly Used Terms and Abbreviations

- **ADC**: stands for Analog-to-Digital Converter.
- **Application specific operations**: set of machine instructions designed for a specific computation domain. The operation circuitry has been carefully designed to take advantage of the hardware, improving certain computation performance that could be more expensive to be executed as a set of general purpose operations. These are generally implemented with custom hardware accelerators or embedded in FPGAs.
- **BRIDGE**: stands for Bilateral Register-based Interfacer for Data and Guidance Exchange.
- **Build**: stands for both synthesize and compile. This term is used whenever referring to the compilation/synthesis composed process.
- **CISC**: Complex Instruction Set Computer. A condensed and complex machine architecture in which complex instructions and very specific operations can be performed. CISC can save memory usage and reduce traffic by performing complex operations with a single instruction call, but their cores are somewhat slower than RISC due to its complexity and plenty of operation modes.
- **CPU**: stands for Central Processing Unit.
- **Cross-Procedure**: FPGA calls as seen from a program running in the CPU.
- **Coarse-grain Logic**: configurable logic device which provides complex functionalities, ranging from adders to full ALUs and CPUs.
- **Conditional flags**: a group of bits stored in register R2 which store the status of an arithmetic, logic or control instruction. Condition flags are always set after the execution of an instruction, and can be used by conditional branch instructions.
- **Configurable**: refers to a device in which the internal logic may be adapted according to the user’s requirements. This is the case of the FPGA in which internal memory cells are used to hold the circuit configuration. This document makes distinction between **configurable** and **programmable**.
- **Configuration**: a group of logically structured bits loaded in the FPGA and stored in Configuration-Memory. A configuration is generated from a synthesized Verilog code, which may contain one or several Modules. An **user configuration** is a configuration synthesized from an user Verilog code.

- **Destination operand**: operand that will receive the result of an instruction after it is executed. Destination operands may also be used in the instruction with the **source operand**.

- **Effective address**: refers to the actual address fetched by the CPU or BRIDGE in memory.

- **Execution**: in this project context, this term refers to a more general meaning rather than software execution. User programs are executed in the CPU, as a standard term for stored-program architecture such as Alchemy. In addition, despite not a coined term, this document states that FPGAs execute their configurations. The reader should notice that it does not mean that the FPGA executes a formal program like those in the CPU. The term only informs that the FPGA circuitry is free to change its inputs and outputs when 'executing'. In other words, the FPGA clock signal is set to change according to the processor main clock signal.

- **Execution call**: FPGA execution request emitted by the CPU which allows the device clock and inputs/outputs to change.

- **Fine-grain Logic**: configurable logic device which provides basic and unspecialized digital logic capabilities, generally as gate level functionalities.

- **Fixed-point**: binary integer representation.

- **Feature Fields**: sequence of at least one bit that holds meaning in an instruction word.

- **Flag**: a bit that carries any contextual meaning.

- **FPGA**: stands for Field Programmable Gate Array.

- **GCC**: GNU C Compiler.

- **General purpose operations**: set of machine instructions designed for a general computation domain. None of these instructions have been optimized to a specific application. They are focus on provide a generic framework for virtually allow the implementation of any operation. These are generally run in a CPU.

- **GET**: see **Global Exchange Table**.

- **Global-range Interconnects**: Wire groups used to broadcast signals to a selected group of FPGA elements.

- **Global Exchange Table**: Global scope memory resource used to temporarily store FPGA call parameters in a C program.

- **I/O Devices**: Input and Output Devices. Any device which is connected to the CPU or the FPGA is an I/O Device. They must be mandatorily mapped in memory and are used to provide an out-side world sight to the Alchemy device.

- **Interrupt Vector**: a set of memory locations, reserved as the 15 top most memory addresses, where the addresses of ISR are stored. The Interrupt Controller reads entries of the Interrupt Vector to guide the CPU to the right ISR according to the requested Interrupt.
LIST OF TABLES

■ **ISA**: stands for instruction set architecture.

■ **ISR**: Interrupt Service Routine. This is a small piece of program run by the CPU whenever an interrupt is issued. There are different routines for different interrupts and their address is stored in the Interrupt Vector.

■ **Logic Capacity**: stands for the amount of actual digital logic a specific configurable device can provide when configured by user logic.

■ **LUT**: stands for Look-Up Table.

■ **Memory hierarchy**: high-level memory architecture, establishing number of memory blocks and their relative function in the system.

■ **Memory organization**: memory structural description, regarding number of words, width and access time.

■ **Module**: high-level hierarchical abstraction for a digital circuit, Usually designed with Verilog or VHDL.

■ **Null**: generally used to represent zero values. The document uses null in several ways. The term can represent null value, null reference and null logic. Null values are standard zero values. As a null reference, registers and memory addresses are pointing to the first memory location. As a null logic, the term refers to FPGA resources that have not had its configuration bits set by the Configuration Controller.

■ **Opcode**: stands for Operation Code. It is a set of three or four bits that uniquely identify an instruction.

■ **PLA**: stands for Programmable Logic Array.

■ **Program**: a sequence of logically related machine instructions run in the CPU and stored in Program Memory. A program is composed by a compiled C code, which must contain a main function and several other secondary functions. An **user program** is a program compiled from an user C code.

■ **Programmable**: refers to a device that consistently executes a group of instructions over a data set, producing results. This term is usually referred to devices which can have their logic changed according to the user’s specifications, but in this document, these devices addressed as **configurable**.

■ **RAM memory**: Random Access Memory block which can be either read or written.

■ **RISC**: Reduced Instruction Set Computer. Denominates a CPU in which only simpler computational capabilities are available and a plenty of registers tries postpone memory access as much as possible. RISC processors tends to be more efficient than their CISC counterparts.

■ **ROM Memory**: Random Access Memory block which can be only read. The block is also randomly accessible.

■ **RTL**: Register Transfer Level.

■ **Run**: see **Execution**.

■ **Single Length Interconnect**: consists of an interconnect which length spans over a single Logic Block.
- **Single-Output-Format**: logic gate representation in which the logic function is described as an n-input, 1-output PLA, used in BLIF files.
- **Source operand**: operand that holds the base data to be used in an instruction. It is usually used to provide content that will be transformed by the instruction and then stored in the **destination operand**.
- **Suspension mode**: informs how the FPGA will finish its execution.
- **Target operation**: term used to refer to the instruction currently under execution in the CPU.
- **Target register**: refers to the register used in instructions. These registers can be Register File registers or BRIDGE Registers.
- **Triggering method**: the way the CPU requests the BRIDGE to start the FPGA. It can be either sequential or parallel and the suspension will always be emitted as an interrupt in the CPU.
- **VLIW**: stands for Very Long Instruction Word. This is a processor microarchitecture that takes advantage of instruction level parallelism as its very concept.
- **Virtual Instruction**: virtual instructions are instructions that implement special functionalities by only combining standard instructions with addressing modes or special registers.
- **VTR**: stands for Verilog-to-Routing.

### General Conventions

#### Binary Representation

This document refers to several binary numbers. They can be of different sizes and can potentially be addressed by individual bits or sets of bits. Most significant bits are always numbered with the highest bit number. Likewise, Least Significant Bits are always addressed by the lower numbers.

The document usually depicts binary numbers as strings of bits organized with their MSB as the left-most bit. Figure 1 shows a 16-bit binary number representation template.

![Binary number template](image)

Figure 1: Binary number template.

Bits are always numbered from zero to their width (minus one). Bits can be identified in an array notation like \texttt{Bin[4]} which refers to the fourth bit of the Bin. Bits can also be addressed by their position. In these cases only the ordinal number is used as a reference. Another notation is the simple bit number, being either written out or represented with the actual number.

When addressing one or more bits in the word, a sequential-array notation is used. If it is referring to the bits 5, 6 and 7, the notation \texttt{Bit[7...5]}. Another possible notation is the textual representation. In this case, the document refers to the bits as "bit 7 down to 5".
Hexadecimal Representation

Hexadecimal base is usually used for shortening large binary numbers. This document refers to binary numbers by simply attaching the 0x prefix to the number. Hexadecimal numbers uses 16 different symbols from 0 to 9 and from A to F. The letters can be written in both lower or upper case.

FPGA Classification

This document classifies FPGAs according to its routing-architecture. This means that more than the simpler routing behavior of the device, the routing architecture is used to differ between different devices characteristics and physical setups.
LIST OF TABLES
Chapter 1

Introduction

1.1 Motivation

Embedded systems are an important segment of today’s electronic industry. Embedded systems can be found in areas such as vehicle control, consumer electronics, communication systems, remote sensing, among other. All systems that consist of electronically programmable components designed to perform a specific function, though they are neither perceived nor used as computers, are generically known as embedded systems [41]. Studies show that above 90% of all processors produced are destined for them and they can be found in a diverse range of applications [51].

Modern embedded systems design must deal with several constraints regarding system complexity, manufacturing costs, product life cycle and performance. In most cases, embedded systems achieve better performance by entrusting application specific computation to dedicated hardwares devices [2] called Application Specific Integrated Circuits (ASICs). These devices are also known as hardware accelerators and are closely coupled to the CPU. This allows the CPU to be used in general control while application-specific computations are performed by the accelerators.

The main drawback of this kind of embedded system comes from the device itself: the systems become overly specified for a given domain, making reusability unattractive in practical standpoints. A device specified for a given application will most likely perform poorly in domains other than its own. A device designed for computer vision for example, have little to do with network protocols once the hardware was optimized to the former. Furthermore, ASICs usually introduce prohibitive costs for most developers and they are not flexible to changes after fabrication.

In early 1980, Xilinx introduced the Field Programmable Gate Arrays, or FPGAs. The main advantage of these devices is the ability to be reprogrammed to desired application or functionality requirements after manufacturing [16], being able to implement virtually any logic circuit. This feature distinguishes FPGAs from ASICs, which are designed and manufactured for specific design tasks [16].

FPGAs have several other advantages over ASICs. FPGAs usually provide faster time-to-market and no NRE (Non-Recurring Expenses), once there are not actual layout and masks manufacturing steps in a typical design flow. Furthermore, the design tools are cheaper than those used for ASICs and the device intrinsic flexibility makes changes and reusability more feasible [19]. However, FPGAs tends to be slower and consume more energy than ASICs, due to the generality of the device architecture.
CHAPTER 1. INTRODUCTION

While ASICs are still the primary source of application specific computations in an embedded system, FPGAs can be used to provide generality to the device. Tying an FPGA to a CPU makes it possible to detach application specific operations from the hardware and allows the configuration of new functionalities on demand. The FPGA can provide new hardware setups for each application, adding flexibility and a new layer of possibilities for embedded systems designers.

The FPGA can also render performance improvement when coupled to a CPU. Test results show that FPGA-accelerators can provide as much as 200-fold performance improvement for the same power or the same performance with a 90% power reduction [2], greatly reducing execution times and energy consumption.

Despite these advantages, FPGAs and CPUs are hardly found together in real world embedded systems. The integration of an FPGA to a CPU has been explored by [37], [47], [20], [45] and [11], and the way this integration is implemented is still an open topic. Considering how the CPU communicates with the FPGA or which is the most suitable memory model for the architecture poses as central questions in this subject.

1.2 Objectives

Given the stated scenario, it is possible to define the following objectives for this project:

1. Define a high-level architecture to implement an interacting CPU/FPGA pair. This high-level architecture must provide sufficient information regarding communication protocols and implementation guideline.

2. Implement the architecture defined in 1 with a transactional simulator language such as SystemC+TLM. This can provide a deeper understanding of the architecture behavior and a high-level system simulator, in a short development cycle. The main advantage of this approach is the ability of foreseeing the device prior to the hardware implementation.

3. Implement the surrounding architecture tools that interact with the system model. Two fundamental tools can be listed at this point: the system compiler and the system synthesizer. These tools must be developed targeting compatibility with the final hardware design and also with the system simulator.

4. Describe an RTL model of the architecture focusing on the actual device implementation. This is meant to be done using a high-level Hardware Description Language targeted for synthesis.

5. Verify and test the architecture models. This aims to test the architecture capabilities and expose its advantages and drawbacks. An implementation exercise must be performed to determine the effort required to design a fully working application in the platform.

1.3 Organization

Chapter 2 presents a Literature Review about Reconfigurable Architectures design in both software and hardware domains. A brief presentation of previous architectures is also described in this chapter. Chapter 3 introduces Alchemy Architecture and all relevant topics required to its understanding. The CPU architecture is first presented, followed by FPGA, the BRIDGE and other architectures elements. Chapter 4 presents a discussion about Alchemy Flows and Cross-Procedures, as well as the new elements in C and Verilog languages. A simplified Assembly guide for the CPU and the FPGA is also described.
Chapter 5 presents Transmute Toolchain, describing each compilation/synthesis step while Chapter 6 describes the architecture implementations, in both System-C (TLM) and VHDL (RTL) and the Validation procedures used to ensure that Alchemy models were correctly implemented. Chapter 7 presents a set of benchmarking tests performed in Alchemy and discusses the implementation of an Alchemy application. Chapter 8 presents conclusions of this project.

A lists of commonly used terms and conventions used through this document is presented before this introduction. Readers shall refer to this section whenever a term used in the text is not well understandable by its context, or to obtain further details about any term. References are presented in the last Chapter.
Chapter 2

Literature Review

Reconfigurable Architecture is a general term used to refer to heterogeneous computer devices that integrate a standard CPU with a reconfigurable hardware. In the last few years, the move of reconfigurable devices from prototyping to computing is more prominent due to several reasons. First, the increasing complexity of modern embedded devices without significant increase in the battery capacity makes it important to understand the interplay between performance and energy efficiency. It is not anymore sufficient to design a highly advanced processor without any concern for energy. This prompted designers to look for various architectural choices, which match the pattern of the application [25].

Second, increasing manufacturing and non-recurring engineering costs demands a design to be more flexible and tolerant to manufacturing and process variations via post fabrication design alterations. Reconfigurable devices provide a way to mitigate the effects of late design errors, process variations, and also ensure longer time-in-market by regular design updates. Finally, the emergence of intelligent devices in our everyday life makes embedded computing a major market compared to general-purpose computing. This made the prominent general-purpose computing vendors to look into the embedded architecture demands, resulting into general-purpose and domain-specific reconfigurable processors [25].

The field of reconfigurable computing is roughly composed of two main subfields: reconfigurable processor architectures and application design. The former is focused on the architecture hardware and how it is implemented on circuit level. The latter describes the challenge of develop, design, debug and deploy applications for this kind of hardware.

According to [52], "The term Run-time Reconfigurable Processor indicates a processor architecture that takes advantage of some form of run-time (dynamically) configurable hardware to provide adaptive instruction set modification, in order to meet application requirements". Similarly, according to [50], the main difference between classical processors and reconfigurable processors is that the latter tries to adjust its internal structure to the currently executed code. Classical processors have fixed, unchangeable structure while all reconfigurable processors have some changeable or adjustable components.

Researches in the last 20 years were engaged in providing different views of the reconfigurable architectures challenge, as presented by [25], and many open topics remain in reconfigurable architectures field. The first one is the device integration, which stands for how to, and in which level, should the reconfigurable device be tied to the CPU. The ideal dimensioning of the reconfigurable device segments, its granularity and the kind of the CPU instructions are also open topics. The programming model and the application design abstraction level are not formalized as well.
Architectures can differ significantly and it is not an easy task to compare them [50]. Due to all these differences, authors may use distinct terms and conventions which are often confusing for general readers. According to [25], this makes it hard to distinguish between a reconfigurable processor from a non-reconfigurable processor.

In [25] it is highlighted two important terms, widely used through this document. The first one is Programmability. [25] defines it as: "The property to be programmable is called programmability. When a device can be controlled by a programming language, then it is called programmable." This definition can be further extended to be more suitable for CPUs. This project defines that Programmability is the property of being programmable. Programmable refers to a device that can be controlled by a high-level language, that consists (in a lower level) of a coherent flow of instructions. Note that the term "coherent flow of instructions" makes its explicit the necessity of a semantically defined set of tasks to be executed by the device.

Configurability, in other hand, is the property of being configurable. A configurable device may have its internal logic adapted to the user requirements, by using a description language to do so. These high-level languages consists, in a lower level, of a bitstream, used to passively change the state of the device internal logic. Note that "passively" means that the device itself does not perform any operation on or with the bitstream. This definition is mostly suitable for FPGAs and PLAs. Being reconfigurable means that the device can be indefinitely configured.

[25] further expands reconfigurability concept by adding two extra terms. A device presents a partial reconfigurability when "it supports reconfiguration of part of it". A device is said to be dynamically reconfigurable when it has the "ability to alter its functionality by executing some task".

The heterogeneity of reconfigurable processors implies that different computation styles are integrated to form a single system. [49] defines two fundamental computation styles that can be used as a model to classify computer systems. The term spatial computers, is generally used to refer to custom hardware (ASICs, VLSI, etc.) devices that exploit parallelism as the base for computations. These devices are usually inflexible to changes and plot higher performance (when compared to other computer systems), once they implement solutions in which operators exist at different points in space and can perform computations simultaneously. Furthermore, due to its inflexibility, spatial computers are usually optimized to the applications they were designed for.

The term temporal computers is used to classify standard computer systems such as microcontrollers, embedded or general purpose processors. These devices perform computations in a sequential pace, taking as input a stream of instructions, executing them individually. These devices are highly flexible once tasks can be changed by changing the instruction stream. Due to its generality however, temporal computers tend to plot lower performance than spatial computers.

Reconfigurable hardware is usually implemented with FPGAs due to its flexibility in configuration and relatively high logic capacity. According [49], the key benefit of FPGAs is that they introduce a class of post-fabrication configurable devices which support spatial computations, providing new organizational points through the space. The CPUs in turn, provide a temporal computation core for the reconfigurable architecture, being useful in tasks that were not specialized by the FPGA. These devices pack both general-purpose temporal execution and application specific spatial parallelism which delivers inherent density and high performance when compared to other architectural approaches.
2.1. **RECONFIGURABLE PROCESSOR TAXONOMY**

[25] presents a complete taxonomy of reconfigurable processor architectures, that allows one to classify the earliest reconfigurable processor architecture development attempts, as well as the most recent and relevant products in the market. The diagram shown in Figure 2.1 presents several design dimensions including the user view, target application domain, and the micro-architectural choices [25].

Through the years, every design represents a consolidation and expansion of earlier research results rather than an unique, completely novel design approach [25]. In many architectures, the reconfigurability is achieved by a separate, closely coupled block, and the programmability is achieved by an existing mainstream processor. The former is referred to as the reconfigurable block following existing terminology and for latter, base/host processor is used. It is interesting to note that for earlier designs, where the reconfigurable block is coupled on a board with a separate processor, host processor is more commonly used to denote the base processor. In the recent designs, this is integrated on a single chip, and, the base processor is used more appropriately. On-board coupling is still a common integration method for emulation platforms [25].

The first criteria presented in the figure is the Coupling. Coupling stands for how do the

![Figure 2.1: Reconfigurable processor design space evolution [Chattopadhyay, 2013].](image-url)
configurable hardware ties to the general purpose processor. The design approaches span from board level, where the CPU and the reconfigurable resource are in different dies and usually are connected by a high speed bus, down to the micro-architecture level, where they are placed in the same die, closely sharing resources. Coupling can have important effects on performance and power consumption.

The Memory Access criteria deals with how does the processor and the reconfigurable resource deal with memory. This can directly affect the system performance and throughput. There are local models where each block has its own memory subsystem, isolated from one another, and global models, where each architecture component access to a common memory resource. Variations in between these two extremes are also possible.

The Target Domain falls in the application field for which the Reconfigurable device is designed for. There exist general purpose architectures, while other can be optimized for specific applications or class of applications. The Target Domain is usually closely related to the reconfigurable resource granularity which defines what kind of logic each component of the reconfigurable resource will perform.

The Base Processor Micro-architecture defines which is the CPU type. This criteria directly affects the system performance once the CPU efficiency is dependent on its internal architecture.

The Instruction Set Architecture (ISA) defines how many and which kind of instructions the CPU has implemented in its hardware. This criteria affects both the CPU performance and the reconfigurable resource architecture. If the CPU is bound to do too much specific operations, it allows no room for the reconfigurable resource to specialize itself. RISC, CISC, DSP or VLIW ISAs must be carefully designed to both improve the system performance and allow the reconfigurable resource to optimize the system for a given application.

The Configuration is the criteria that informs how the reconfigurable resource gets its functionalities configured, based on user requirements. It is directly related to the reconfigurable resource organization.

The Programming Environment criteria, deals with the development environment made available for users to design their applications. There exist several approaches, spamming from a low level assembly programming up to high-level kernel-based models. This criteria has nothing to do with the underlying architecture it is designed for. Most of the features provided to designers depend on the compiler/synthesizer, development libraries and design IDEs.

The Interconnect topology classifies the reconfigurable resource organization from its internal modules connection stand point. It intrinsically deals with the physical organization of the reconfigurable resource matrix and the efficiency of its routing architecture.

Lastly, Reconfigurable Logic Function classifies how do the functional blocks in the reconfigurable resource are organized. They can be designed to describe simple logic functions, with an LUT based block, or can be more specific with ALUs or DSP operations. This also affects the workload partition with the CPU. Complex reconfigurable logic functions may render the CPU underused in the system and over specify the system towards a given application.

2.2 Reconfigurable Architectures Structural Classification

Taking the taxonomy axis presented in Figure 2.1, reconfigurable architectures are usually classified by the coupling degree between the reconfigurable resource and the CPU. Some authors such as [9] can derive up to four different degrees of device coupling. [48] also presents an alternative classification, which defines up to five different degrees (or classes) of architectures. This project is attached to a simpler definition presented in [25] consisting of three distinct architecture groups.
The coupling also determines where and how many processing units and reconfigurable units will be in the system. Figure 2.2 presents a general three degree architecture classification. Note that the structural classification is not attached to the concept presented in Figure 2.1 once it specifies who the CPU is integrated with the reconfigurable device.

Degree I (Figure 2.2 (2)), entitled as *Add-On Reconfigurability* by [25], used the reconfigurable device to improve the overall system performance and provide post-fabrication flexibility. This is usually implemented with two separate blocks, generally coordinated by a bus protocol. This is the usual approach embraced by desktop/embedded processors vendors. The main goal of this approach is to maintain full software programability while keeping baseline software unchanged.

Architectures such as Intel Atom E6x5C implements this coupling degree by combining a standard processor with a commercial fine-grained FPGA via a high-speed PCI Express Bus [26]. The greatest advantage is that the processor can offload its computing task to the reconfigurable logic via the high-speed bus [25], once it does not provide any configurable resource by itself.

Degree I architectures can be implemented in two integration levels. At board level, CPUs and configurable resources in different dies or printed circuit boards, communicate by means of a bus interface. Projects such as LogiBone [28] is an example of board-level Degree I architectures. In the die level, the CPU and the reconfigurable device interact by internal wire set, generally behaving like a bus.

Degree II architectures (Figure 2.2 (1)), also called *Add-On Programmability* is an architectural trend started by FPGA and reconfigurable devices manufacturers like Xilinx and Altera. In this coupling style, there exist only a large reconfigurable resource, and part of it is allocated for the processor. It essentially configures the processor IP core into the reconfigurable resource and the remaining logic is available for users to implement their applications. This allows easy integration with user modules and other IPs, while maintaining programability through the processor core.

Examples of this coupling degree are Xilinx Zynq-7000 FPGA devices [57] which integrate an ARM Cortex-A9 IP core in the reconfigurable resource. Designers can extend the system capability by implementing custom functionalities with the FPGA resource surrounding the
CHAPTER 2. LITERATURE REVIEW

CPU. This coupling degree is directly implemented on die level and is not expansible like Degree I coupling.

The third coupling Degree (or Degree III) is called custom processing and reconfigurability. This approach defines a processor and a reconfigurable resource tuned to a specific application. Essentially, this coupling degree allows the adjustment of the system with functionalities tied to a reconfigurable logic. The CPU can have internal functional blocks changed during runtime by other predefined modules, based on the requirements of the user. This is the least common coupling style and can be found in ADRES architecture [14], for example. Figure 2.2 presents Degree III structure on diagram (3).

2.3 Reconfigurable Application Design Approaches

Good reconfigurable solutions are often quite different from the good sequential solutions familiar to most programmers. Building good reconfigurable designs requires an appreciation of the different costs and opportunities inherent to reconfigurable architectures [9]. Perhaps in an elementary level, reconfigurable application design can be optimized by compiler tools. These tools are in charge of converting high-level software specifications into applications that blend both hardware and software domains.

There exist two main approaches to design tools: the Annotation and Constraint-driven approach and the Source-directed Compilation approach [48]. The former attaches to high-level programming languages such as C or C++, syntactic annotations and semantic statements that hints the compiler when and which optimization should be performed. The user must explicitly write portions of code detailing their intentions to the compiler. This approach is usually available in kernel-based programming environments or in parallel or distributed systems.

For reconfigurable architectures domain, a number of compilers from C to hardware have been developed. These range from compilers which only target hardware to those which target complete hardware/software systems; some also partition into hardware and software components [48]. Their strength is that usually only minor changes are needed to produce a compilable program from a software description [48]. Five representative methods are SPC [31], Streams-C [55], Sea Cucumber [53], SPARK [54] and Catapult-C [7].

A different compiler design techniques is focused on the source-directed compilation. This adapts the source language to enable explicit description of parallelism, communication and other customizable hardware resources [48]. Examples of design methods following this approach include ASC [10], Handel-C [22], Haydn-C [40] and Bach-C [21].

Despite widely necessary, design tools per se are not enough to attain high quality solutions for reconfigurable architectures. It is valuable to identify and catalog design patterns for reconfigurable computing. These design patterns are canonical solutions to common and recurring design challenges which arise in reconfigurable systems and applications [9].

Simply put, a design pattern is a solution to a common problem. Their necessity come from the idea that writing in a code with a high-level language is not sufficient, by itself, to produce good reusable software. Design patterns are usually organized following a framework which includes Name, Intent, Motivation, Applicability, Participants, Collaborators, Consequences and Implementation.

2.4 Featured Reconfigurable Processor Architectures

This section presents some reconfigurable processor architectures, designed through the last 20 years. Basing on the taxonomy criteria introduced in Figure 2.2, the architecture presented
Table 2.1: Reconfigurable Architectures classified according to The axis of Figure 2.1

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Year</th>
<th>Programming</th>
<th>ISA</th>
<th>Coupling</th>
<th>Granularity</th>
<th>Reconf.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRISM</td>
<td>1992</td>
<td>HLL</td>
<td>M68010</td>
<td>Loose</td>
<td>Fine</td>
<td>Static</td>
</tr>
<tr>
<td>Nano</td>
<td>1993</td>
<td>ASM</td>
<td>RISC</td>
<td>Tight</td>
<td>Fine</td>
<td>Static</td>
</tr>
<tr>
<td>Garp</td>
<td>1997</td>
<td>HLL</td>
<td>MIPS</td>
<td>Coprocessor</td>
<td>Fine</td>
<td>Dynamic</td>
</tr>
<tr>
<td>MOLEN</td>
<td>2004</td>
<td>HLL</td>
<td>PowerPC</td>
<td>Tight</td>
<td>Fine</td>
<td>Dynamic</td>
</tr>
<tr>
<td>DySER</td>
<td>2012</td>
<td>HLL</td>
<td>RISC/CISC</td>
<td>Tight</td>
<td>Coarse</td>
<td>Dynamic</td>
</tr>
</tbody>
</table>

herein were selected based on their features and publication period. They are described in chronological order. A full list of architectures, from 1992 to 2012, can be found in [25].

Table XX presents a brief comparative between the presented architectures. The comparison features were based on the axis presented in Figure 2.1 and in the considerations presented in [25].

2.4.1 PRISM Architecture

PRISM stands for Processor Reconfiguration through Instruction-Set Metamorphosis. It was one of the earliest works in the area, in the early 1990s. The coprocessor was a loosely coupled device, with static configuration, rudimentarily implemented in a board level.

In the original paper [37], the author states that the applications typically spend most of their execution time in small portions of executable code [37], and that by adapting the processor to these frequently accessed portions of code can substantially improve performance. [37] enforces that these kind of architectures retain its general-purpose nature, while reaping the performance benefits of application-specific architectures.

[37] also claims that RAM-based FPGAs can be used to implement instruction set augmentation while keeping processing platforms general purpose. Furthermore, better performance improvements are most likely to be achievable when the configurable resource and the CPU are closely integrated.

PRISM approach is based on a three step convention protocol. This protocol establishes the way the CPU and the FPGA cooperate during runtime and how data is exchanged between them. First, data is passed from the CPU to the FPGA by a bus. Second, the reconfigurable device evaluates the inputs and produces outputs. Third the CPU acquires the FPGA outputs.

This is essentially the basic behavior of most of the reconfigurable architectures to come, and determines in a higher level, how do the architecture blocks are meant to interact. [37] also states that performance improvements can only be possible if these three steps take less than the average time to evaluate the same function in software.

PRISM hardware, codenamed PRISM-I was a prototype developed by the authors to demonstrate the architecture viability. They state that "the design was intentionally simple to demonstrate the utility of the architecture". The prototype was built with standard off-the-shelf components connected at board level. The CPU consisted of a 10-MHz Motorola 68010 processor and the reconfigurable resources consisted of four Xilinx 3090 FPGA, both tied by a 16-bit bus.

The authors of [37] were largely careful with the development environment and the consequences the reconfigurability would impose over programmers. To make the platform friendly for users a configuration compiler was developed for PRISM.

The authors reasoning for developing a compiler framework for the architecture falls into two arguments. First, software programmers usually do not have hardware development experience.
This could lead to suboptimal reconfigurable resource usage, potentially degrading the system performance. Second, the programmers should not be the ones to select the portions of code to be parallelized in the FPGA. These two tasks could be overwhelming for most programmers, and can potentially render the architecture nearly unusable.

The compiler is in charge of performing three basic operations. It must compile the user software into actual executable code, identify and extract candidate software parts that can be converted into actual hardware implementations in the FPGA, and finally it must synthesize the hardware.

The candidate code segments to be converted into hardware are selected by performing an estimation over the runtime behavior of the software. This way, the parts of the code that impose larger impact in the performance can be identified. The authors achieved that by using "hardware-architectures and instruction-set models, probability models for conditional execution and rule-based analysis" [37] without actually running the program.

Despite identifying the candidate functions, a minimal user-interaction is required. The compiler prompts the users to choose, between several candidates, which functions will be implemented in the hardware. The selected functions are then converted into hardware and an "access-definition file" determines how the software access the newly synthesized functions.

After synthesizing the hardware, a new program source code is generated. The compiler is in charge to generate code stubs that efficiently access new functions with the minimal amount of data movements. The compiler also loads library codes used to initialize the hardware configuration into the FPGA array. Later on, the compiler performs a series of standard software optimizations over the newly created software source code.

The architecture was tested with a set of programs specially developed to take advantage of its features. The speedups achieved fall in the order of few dozens of times, down to the double. Applications such as mathematical manipulation and binary transformation were those that best performed in PRISM.

The authors highlight that the outstanding performance improvements were constrained by a suboptimal and relatively slow bus interface. This interface overhead was considered a potential improvement on later architecture versions, being possibly reduced by a factor of 20.

2.4.2 Nano Processor

The Nano Processor is a so called customizable stored-program processor and was developed by Michael Withlings, Brad Hutchings and Kent Gilson in 1994. The main consideration taken by the authors to implement the architecture is that FPGAs can render good performance improvements for computer systems, while maintaining a good flexibility and lower development cost. They state that "reconfigurable logic systems can approach the performance of custom ASICs without the inflexibility of custom silicon" [47].

Nano Processor (nP) is developed strongly thinking of reusability and to provide higher flexibility and performance for application-specific operations running over a general-purpose platform. [47] also define a concept of Instruction Library, in which designers could share functionalities and easily make them available for different projects and domains, to easily address the hardware design, and widely encourage reusability.

Unlike PRISM, Nano Processor implements the general-purpose core in the reconfigurable resource. According to the authors, "Not only does this reduce the part count but it allows full control over processor operation". As with PRISM the Nano Processor offers available reconfigurable logic for implementing application-specific hardware to achieve application-specific performance [47].

The Nano Processor core was designed to require only a small portion of the reconfigurable
logic resource. Minimizing the control logic registers and busses frees the logic and routing resources necessary to implement application-specific hardware in a single FPGA [47]. The basic idea of Nano Processor is to build a minimalistic core surrounded by a custom instruction set shell which will allow software execution.

The nP core provides six general purpose essential and immutable instructions which can operate without any customization. It also provides a single register called Accumulator, along with an eight bit data path, opening way for 32 customized instructions. The core is intentionally simple to force users to specify their platforms towards a target application. [47] states that despite the simplicity of the core, "Several designs have been implemented (...) with little or no customization".

The custom instruction set sits upon the minimalistic nP core, being considered in a "higher level" architecture element. A custom instruction set is built by choosing instructions from an instruction library or designing new instructions [47]. New instructions are developed with standard schematic entry or high level synthesis tools. After a new custom instruction has been designed and verified it is placed in the instruction library of nP custom instructions [47]. This allows custom functions to be reused by users.

The architecture deals with custom instructions by encapsulating them into instruction entities that provide easy interfacing and control. The instruction can become an active member of the processor and operate in parallel with other events in the processor [47].

According to the authors, a special purpose data sorting processor could be built with high speed hardware sorting algorithms. Without any custom instructions the nP core could perform simple sorting algorithms. But, like most processors it must proceed byte by byte through the data structure and perform individual comparisons on the data set. A custom sort instruction could be developed that when given two address pointers, would read the values, compare, and swap if necessary. Much of the overhead in data calculation and instruction processing would be removed [47].

In a higher level, lies the Executable Software. Users program the Nano Processor with assembly language, using both the predefined instructions and the custom instructions. The assembly instructions allow minimal data movements like load/store from/to memory or load accumulator, add memory to accumulator, subtract memory to accumulator and jump, if no carry is identified.

For Nano Processor, the concept of custom instruction is way more complex than simply extending the arithmetic logic unit or patching the control unit with newer operations. Custom instructions must be integrated in the processor core architecture, by actively interacting with instruction register (they must decode their operands), and execute their functionalities, following the machine cycle pace.

A custom instruction is formally usable for user programs by using the Nano Assembler. This tool takes instruction definition file (.INS), which provides the name of the instructions, the opcode and the instruction length. After this definition, conventional assembly programs can make use of the custom instructions.

The Nano Processor was tested with a couple of applications that demanded the core to adapt to the intended task. As an audio interface controller, nP the configuration was designed to control a complex multimedia sound card. The second application is a set of software modules that run on the custom controller designed with nP. The authors focus was not performance but rather the flexibility the platform provides when integrated with other devices in a larger system, and running custom software.
2.4.3 Garp Architecture

Garp is a reconfigurable processor architecture based on MIPS developed by John Hauser and John Wawrzynek in 1997. It is a hybrid reconfigurable platform where the FPGA is a slave coprocessor attached to the CPU, in the same die. Users can choose to specify the platform for their applications or not use the FPGA at all. The authors enforce that the main thread of control is managed by the processor core. The reconfigurable resource is meant to be used in specific situations, like during a loop, to obtain speedups.

Garp makes external storage accessible to the reconfigurable array by giving the array access to the standard memory hierarchy of the main processor. This also provides immediate memory consistency between array and processor [20].

The CPU core executes a MIPS-II instruction set extended for Garp. Garp’s reconfigurable Array is composed of entities called block that roughly correspond to the CLBs of the Xilinx 4000 series [20]. The Garp Architecture fixes the number of columns of blocks at 24. The number of rows is implementation-specific, but can be expected to be at least 32. The basic quantum of data within the array is 2 bits. Logic blocks operate on values as 2-bit units, and all wires are arranged in pairs to transmit 2-bit quantities [20].

The loading and execution of configurations are under the control of the main processor. Several instructions have been added to the MIPS-II instruction set for this purpose, including ones that allow the processor to move data between the Array and the processor’s own registers. Four memory busses run vertically through the rows for moving information into and out of the Array. During Array execution, the memory busses are used for data transfers to and from memory and/or the main processor. Distributed within the Array is a cache of recently used configurations, so that programs can quickly switch between several configurations without the cost of reloading from memory each time [20].

The processor core has its instruction set modified to support the management of the reconfigurable resource. There exist 11 instructions that allow different management styles, depending on the needs of users. To avoid restricting the main processor implementation, the Garp Architecture does not specify how many main processor instructions might execute during each Array clock cycle. Instead, to keep processor and Array synchronized, many of the new processor instructions first wait for the Array clock counter to reach zero before performing their function [20].

Each block in the Array requires 64 bits to be configured. This results in a 6144 bytes for full Array configuration which takes about 50 \( \mu \) seconds to be completed. Since not all useful configurations will require the entire resources of the array, Garp allows partial array configurations [20].

Software tools have been created that make it possible to write programs for Garp and then simulate them with approximate clock-cycle accuracy. An array configuration is coded in a .ga file with a simple textual language. This source is fed through a program called the Configurator to generate a representation of the configuration as a collection of bits [20]. The only need for Assembly language programming is to invoke the Garp instructions that interface with the reconfigurable array. Since it is used the Gnu C Compiler (gcc), the same could be
accomplished with inline \texttt{asm} statements. This way, users can have high-level access to custom instructions. The major drawback of this approach is that the Array configurations are written in a low-level Assembly like language, requiring explicit description of the behavior of each block in the rows and columns of the Array.

The architecture was evaluated comparing programs implemented for Garp and a Sun UltraSparc processor. The authors developed a triple of simple applications, one for data encryption, a 640 x 480 image dither algorithm and the sorting of 1 million records. In the best scenario, the first application could perform 24 times faster than the UltraSparc processor. The other programs described a discrete improvement of about two times.

### 2.4.4 MOLEN Processor

MOLEN is a reconfigurable processor architecture concept designed by Georgi Kuzmanov, Georgi Gaydadjiev and Stamatis Vassiliadis in 2006. Like other reconfigurable processors presented in this Chapter, MOLEN is composed of two major architectural elements: the General-Purpose Processor (GPP) and the Reconfigurable Processor (RP).

An ARBITER is used to determine where an instruction should be issued during run-time - those instructions implemented in the fixed hardware are issued in the GPP, while custom instructions are redirected to the Reconfigurable Processor. Memory accesses are managed by a special controller with multiplexers and demultiplexers, which distribute data between each processing core.

According to the authors, differently from other architectures, MOLEN defines a reconfigurable microcode used to decide functionalities to the reconfigurable core. This microcode is also responsible to configure the functionalities into the Reconfigurable Processor. Data is exchanged between the GPP and RP through a set of registers known as "Exchange Registers" or XREG.

MOLEN’s reconfigurable resource can be set to run in seven different modes, depending on the instruction issued to configure it. It can perform a partial configuration, a full configuration, execute code (coming from the ARBITER), fetch configuration for bitstreams stored in the internal storage in the chip, synchronize parallel execution and move content from/to GPP registers to/from Exchange registers.

The General Purpose Processor was designed with a minimal ISA called \(\pi\)ISA. This Instruction Set Architecture provides four elementary instructions: \texttt{set}, \texttt{execute}, \texttt{mvtx} and \texttt{mvfx}. These instructions are always directed by the ARBITER to the GPP core. By running a \texttt{set} or \texttt{execute} the ARBITER puts the GPP into a wait state while RP is set to start running custom instructions. After the instruction is executed, the GPP regains control over the system, resuming in the instruction following the RP instruction. RP issues a sequence of instructions to give the control back to GPP.

Custom instructions are coded following the standard of unused opcodes from IBM’s PowerPC architecture. The PowerPC ISA was chosen once the prototype of the system was designed based on a Xilinx XC2VP20-5 device, which is shipped with two PowerPC cores. According to [45], MOLEN disposes two memory resources of 64 KB each to store, separately data and instructions. Larger amounts of memory can be made available as external modules. The system also defines three different clock domains, one for GPP, other from RP and another for the Memory.

The architecture was evaluated by running a set of tests only using the available PowerPC cores in the prototype and another using MOLEN’s reconfigurable capabilities. There was implemented an MPEG2 decoding algorithm, which could achieve speedups of 8 to 300 [45], depending on the algorithm parameters.
2.4.5 DySER Architecture

DySER stands for Dynamic Specializing Execution Resources. It is a reconfigurable processor architecture designed by Govindaraju et al (2012), supports both customizing its internal architecture to provide new functionalities as well as the way these functionalities are performed (in parallel) over data. By dynamically specializing frequently executing regions and applying parallelism mechanisms, DySER can provide efficient functionality and parallelism specialization [11]. According to the authors, it outperforms an out-of-order CPU, streaming SIMD Extensions (SSE) acceleration and GPU acceleration, while consuming less energy [11].

The authors define two terms for functionality. Parallelism specialization is used for architectures that employ some sort of data-level parallelism such as GPUs or SSE. Functionality specialization refers to architectures that provide custom hardware targeted at application functionalities. These are FPGAs and reconfigurable architectures.

DySER is conceived considering that previous reconfigurable architecture have targeted only parallelism or functionality specialization, but not both. DySER implements both parallelism and functionality specialization by providing a configurable lightweight switching network that connects a set of heterogeneous functional units and allows customization. DySER exploits parallelism by creating logical lanes of independent computation in this substrate and exploits functionality by creating specific data paths for each particular computation [11].

In practical terms, DySER is integrated into a general-purpose processor’s execution stage, which acts as a load/store engine to feed the DySER computation substrate. To achieve functionality specialization, a compiler synthesizes data paths among functional units. To achieve parallelism specialization, it is used a judicious mix of vectorization techniques and novel hardware mechanisms [11].

High performance is enabled by providing a dense computational fabric with low-latency integration to a processor, and energy efficiency is attained through the elimination of per-instruction overheads by converting code regions into dynamically formed compound functional units. These gains are possible without significant disruption to either the general-purpose processor architecture into which DySER is integrated or the software development environment [11].

Employing functionality specialization, DySER outperforms a dual-issue out-of-order (OOO) processor by 1.1 times to 4.1 times, while simultaneously reducing energy consumption by 9 percent. Employing parallelism specialization, DySER outperforms single instruction, multiple data (SIMD): it is 1.3 times to 4.7 times faster than SSE, consuming 86 percent less energy. It also outperforms GPUs, with an average speedup of 1.4 times, while consuming 8 percent less energy consumption [11].
Chapter 3
Alchemy Architecture

Alchemy is an embedded reduced instruction set computer integrated with a field programmable gate array. The architecture implements a polymorphous ISA based on Texas Instruments MSP430 ISA that can be expanded by user specification with the FPGA. It can be seamlessly integrated in ordinary processing environment, being attachable to libraries and existing softwares and toolchains. Its reconfigurable resource allows the implementation of specific operations that improve the overall system performance.

The architecture was designed to fill the gap on adaptability in embedded systems. Most of embedded systems consist of static pieces of processing hardware, tightly attached to its domain, and poorly performing out of it. Alchemy’s basic concept is to create something that any embedded system designer could use in their projects, without drastically increasing development cost and time. Alchemy tries to simplify the architecture usage, management and development by using several different techniques.

Alchemy was designed to serve as a low-end embedded system architecture. It provides general purpose computations and custom application-specific instructions by combining user-defined hardware accelerators with customized control software routines. This allows the implementation of various Flynn’s SISD, SIMD or MIMD multiprogramming contexts [39] in the same device.

The CPU and the FPGA interact by means of a general communication protocol. It defines a data exchange model, five CPU instructions and a set of FPGA signals for a multilateral and concise interaction framework. This protocol is implemented and handled by a special block called BRIDGE Controller. It allows data to be sent and received between the CPU and the FPGA, enables the emission of coercive FPGA Cross-Procedure call requests\(^1\), allows the reception of FPGA return interrupts\(^2\) and provides direct memory access to the FPGA.

Alchemy memory organization is defined with three 16-bit wide 64-KB blocks. Two static memory resources are the Program Memory and Configuration Memory which are only allowed to be read during run-time. A Shared Data Memory is a dynamic memory resource that is shared among the CPU and the FPGA. It can be either read or written during execution. In Alchemy memory hierarchy, each memory block is little-endian and was designed after Harvard organization.

\(^1\)FPGA Cross-Procedure call requests are unidirectional execution requests emitted by the CPU, asking for the system to allow the FPGA to receive clock signal and run its configuration.

\(^2\)Whenever an FPGA suspends its execution a return signal must be sent from the FPGA to the BRIDGE. This signal is then redirected to the CPU’s interrupt controller which issues an interruption to the CPU core.
It is also defined a set of interrupts and busses which compose the architecture and allows outside-world visibility. A heterogeneous architecture diagram can be seen in Figure 3.1.

Figure 3.1: Simplified Alchemy block diagram. The most relevant architecture components are depicted and labeled from 1 to 14.

In the context of Figure 2.1, in the Configuration dimension of the chart, Alchemy fits in Configuration Memory level. Its Instruction set is a RISC and its Base processor microarchitecture is pipelined. The Architecture’s Target domain is general, Its primary Memory access is in both shared and shared and its Coupling is On-chip tight. Alchemy’s Reconfigurable logic function is a LUT and its Interconnect topology is a Mesh. Finally, Alchemy’s Programming environment is both Assembly, HLL.

The following sections will detail the architecture structure in an implementation-independent manner. Since this project presents two different versions of the same architecture (one as a System-C TLM model and other as VHDL RTL model), no implementation details will be given at this point. A high-level detailed description is given instead.

Alchemy CPU (1) is a multi-cycle processing core which implements various instructions, a wide set of registers and multiple control blocks. Its micro-architecture was designed to provide high instruction throughput with short machine cycles and wide memory resource availability. Details regarding the CPU architecture will be presented in section 3.1.

Alchemy FPGA (2) was designed to provide plenty of logic resources along with feasible routing capabilities. Henceforth its architecture consists of a fine-grained, island-style mesh device, with 64 I/O Ports and 256 general purpose configurable logic blocks. Alchemy FPGA micro-architecture is presented in section 3.2.
The BRIDGE (3) is a bus arbiter, a protocol simplifier, and a device manager for both the CPU and the FPGA. When acting as a bus arbiter, the BRIDGE deals with both CB (4) and FB (5) busses forwarding requests from both sides. When acting as a protocol simplifier, the BRIDGE provides direct memory access to the FPGA simplifying the actual memory protocol. As a device manager, the BRIDGE is used to block the FPGA clock, outputs and inputs to change. More details regarding the BRIDGE is presented in section 3.3.

The architecture defines two high-speed local busses called CB (4) and FB (5) busses. CB bus connects the CPU to the BRIDGE and has a hierarchical organization with data, address and control lines. FB bus connects the FPGA to the BRIDGE and categorizes its lines as either control or I/O. Three major global busses provides data (8), address (7) and control (6) signals exchange between the several architecture elements. These busses are wider and busier than the local CB and FB busses. The FPGA also has a set of internal busses called "interconnects". Alchemy busses architecture will be described in section 3.5.3.

The figure also describes three memory elements. These are the Program Memory (9), Configuration Memory (11) and RAM Memory (10) (Shared Data Memory). The first two blocks store static system software and hardware binary data that can only be read in runtime. The Shared Data Memory is a dynamic memory resource that is shared among the CPU and the FPGA. Further memory organization details and Memory management unit (12) will be presented in section 3.4. Alchemy Interrupt Controller (13) is presented in Section 3.5.1, Alchemy Configuration Controller (14) is presented in Section 3.5.2.
3.1 Alchemy CPU Architecture

The CPU is the programmable element in Alchemy architecture. The CPU is mainly focused on general purpose computations and overall control procedures. Basic logic and arithmetic operations are implemented in the CPU. More complex and application-specific operations are left to be implemented in the FPGA.

Alchemy CPU is a coordinator and an important computation element in the architecture. Its primary function is to execute user programs that are stored in Program Memory. The resulting dynamic data is stored in the Shared Memory resource. The CPU can only execute one instruction at a time and these instructions can take different times to be executed.

The CPU was designed to be an efficient RISC implementation of MSP430 ISA. Alchemy retains the architecture instructions, but its internal logic is potentially completely different from the one in MSP430 devices. Furthermore, Alchemy includes new instructions to the set. It was also designed to be an active entity in the processor ecosystem. The CPU can directly deal with the BRIDGE and the FPGA by exchange data and control signals using CB Bus. The CPU includes 27 instructions, 5 BRIDGE control instructions and 22 virtual instructions. A wide Register File allows convenient local operand storage while memory can be accessed with a simple and low-overhead memory load/store instruction. It was also designed to perform simpler fixed-point arithmetic rather than more complex floating-point operations, to keep the hardware complexity low. The CPU counts with 13 customizable interrupt and high-speed local busses. Figure 3.2 depicts the essential CPU elements. Note that the Memory Controller is not part of the CPU.

![Figure 3.2: Alchemy CPU simplified Architecture.](image)

3.1.1 CPU Modes

At any given time, the CPU is in one out of four possible modes: Master, Partner, Independent and Off. These modes represent the interaction-model between the CPU and the FPGA. When
running in Master mode, the FPGA is not active and the CPU runs independently.

When running in Partner mode, the CPU uses the FPGA as its co-processor to execute any application-specific operation configured on it. In this mode, the CPU must request the FPGA as needed, and memory access can be performed concurrently. A request queue guarantees memory access coherence during execution.

Independent mode also provides concurrent execution when the CPU and the FPGA run independently from one another. In this mode the FPGA does not require CPU requests to start running. The FPGA is triggered from the system startup and will continue so until the system is shut down.

The Off mode disables CPU during runtime. This means that the CPU will be turned off and only the FPGA will be active. This mode can be useful for power-saving, when only simple operations are needed.

It is not defined a hierarchy between the CPU and the FPGA. Priority over memory accesses is decided based on the principle of "first come first served" and no privileged CPU instructions are defined, in order to keep the architecture simple enough for most users. It is transferred to them the responsibility of implementing features like security levels and privileged modes.

The CPU includes a small block called State Controller which is in charge of enabling or disabling the control unit depending on the desired operation mode. Two bits are used to encode the CPU state, as shown in Table 3.1.

Table 3.1: CPU modes and corresponding State Controller encoding.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Binary Code</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master Mode</td>
<td>00</td>
<td>CPU only</td>
</tr>
<tr>
<td>Partner Mode</td>
<td>01</td>
<td>CPU + co-processor</td>
</tr>
<tr>
<td>Independent</td>
<td>10</td>
<td>CPU + FPGA</td>
</tr>
<tr>
<td>Off</td>
<td>11</td>
<td>FPGA only</td>
</tr>
</tbody>
</table>

### 3.1.2 CPU Data Model

Alchemy CPU is defined with three basic data types: bits, bytes and words. A bit is a single binary digit and is considered the elementary data type. A byte is a group of eight bits. Each memory block is composed of several rows of bytes. Words are compositions of two bytes (16 bits) and are used to represent integers, addresses and instructions.

Complex data types such as strings, structs and enums are software abstractions for those basic hardware data types. Basic types such as integer are directly represented with 16 or 8 bit variables which are in turn words and bytes, respectively.

Integers (aka int) can hold values ranging from 0 to $2^{16} - 1$ when unsigned and represented with a word. When signed, word-size integers range from $-2^{15}$ to $2^{15} - 1$. Negative values are encoded with two’s complement representation. Likewise, short integers (represented as a byte) range from 0 to $2^8 - 1$ when unsigned and from $-2^7$ to $2^7 - 1$ when signed.

The compiler can encode and treat integers larger than words by handling upper and lower parts separately. These are long integers that are 32-bit wide words and double-words integers (long long integers) that are 64-bit wide. Similarly, strings and arrays are collections of bytes held sequentially in memory. For strings, bytes are labeled as char and each one can represent up to 255 different characters, depending on the encoding format.

Structs and other complex data types are clusters of memory entries of variable size. They can be individually accessed following an address frame which defines a base address and a set
of offsets that match each basic data type composing it. These data types can spread through the memory, largely exceeding basic data types length.

The processor can run in either word or byte mode. When running in word mode, arithmetic and logic operations are performed over a word-size value. When running in byte mode, operations are performed over the least-significant byte of a word and the rest of the word is ignored by the CPU. This can potentially change the behavior of operation results and conditional flags. Each instruction has a special field (1-bit wide) to inform the execution mode. Alchemy retains these two operation modes to provide full compatibility with MSP430 ISA.

3.1.3 CPU Registers

Registers are essential memory resources. They provide local and high-speed data storage for computations. Registers are the core storage element in RISC processors. Alchemy CPU provides a single register-group called Register File. The register file basic structure can be seen in Figure 3.3.

![Alchemy Register File Structure](image)

The Register File takes as input a set of three register addresses. These addresses are four-bit words each referring to a specific register\(^3\). Two addresses are used to read the content of registers, while the third one is used to store the data in a target register.

The Register File is a set of sixteen registers. This number of registers tries to balance the amount of bits required to address each one of them, and the availability of memory resources in the CPU. A register is labeled as General Purpose Registers (GPR) and referred as \(R_x\) where \(x\) is a number from 0 to 15. The high amount of registers provided by the CPU matches a trend of postponing memory access in RISC CPUs. The compiler is designed so that local data is primarily stored in registers and are gradually stored in memory as needed. Memory accesses are much more expensive than Register Files accesses and so that they must be avoided.

A register in the Register File has the length of a word (two bytes). It has twice the size of a memory location\(^4\). and can store 16-bit binary data or a full memory address. Instructions can make direct reference to any register by means of a special four-bit wide field. Instructions can use up to two registers, one for source operands and another for destination operands.

\(^3\)in order to encode 16 possible values, four bits are necessary.

\(^4\)MSP430 defines that memory locations are 8-bit each. Accesses are performed in pairs so that 16-bit are read or written in memory
3.1. **ALCHEMY CPU ARCHITECTURE**

GPRs can be freely read or written during run-time. They can be classified as Common or Special. Figure 3.4 shows the basic organization of the Register File.

![Alchemy Register File Registers](image-url)

The registers numbered from 0 to 3 are the Special Registers. They are reserved in the sense that they perform special functions in the CPU, but can be directly accessed by the user code. Register R4 to R15 are context-free registers and their primary function is to store program data. Further functionalities may be assigned to any one of these registers, depending on the user program.

The first special register is the Program Counter (R0 or PC). Its primary function is to store the memory address of the current instruction. The PC is used as a reference to the program memory and can have its content changed in three situations. Once at each machine-cycle, the PC is automatically incremented by the CPU and FPGA, so that the next instruction in memory can be executed. Instructions are organized sequentially in the Program Memory and the increment is performed taking into account the address alignment (two units for each instruction).

The PC can also have its content changed by a branch instruction. Branches can sum signed 10-bit integer values to the PC so that the execution can be brought to desired points. Subroutine call and Return from interrupt instructions can also change the PC address in the same way.

User code can also change PC content by writing or performing arithmetic or logic operations on its content. This is an unusual situation, despite being valid. This procedure can be observed in the system prologue, when the PC is directly written with the immediate address of the **main** function of the user program.

The Program Counter is the only register that cannot have its content changed by the CPU mode. That means that whatever is the operation mode set by the instruction (8-bit or 16-bit) the program counter will still remain 16-bit wide.

The Stack Pointer (SP or R1) is another special register. Its function is to store current system stack head address. Several instructions make reference to its content.
Despite being a special register, SP can be optionally used by the user code as long as the stack is not used. For each push performed to the stack, the Stack Pointer content is decremented by two units. Likewise, a pop increments SP by two units. The Stack pointer is generally used as an index for memory accesses (with indirect addressing mode).

Status Register (R2 or SR) is the third special register. It is used to store bits that represents the status of an early performed operations. The status register also carries bits for controlling maskable interrupts and to shut the CPU down. Not all instructions change the content of SR.

R2 has four status flags defined on its bits. C represents the carry flag and is always set when an arithmetic or logic operation generates a carry. Z flag informs if the target operation resulted in null value. This can have either arithmetic or logical meanings and are treated accordingly by the target instruction.

N informs if the most significant bit resulting from a target instruction is set. This bit is checked once two's complement negative numbers have always 1 as their MSB. This is used by either arithmetic or logic instructions. V is the overflow flag and informs if the result of a target operation will not fit in an ordinary register. This means that the result will exceed 16 bits and the flag is used to sign this situation.

Any status flag can be freely changed by the user code. Reads and writes can be performed over the Status register but its content is most likely to be changed by the next instruction result. Conditional Branch instructions use SR flags to define whether a branch is taken or not. They can only perform reads over its content.

The status register also stores system-flags. The Control Unit is directly tied to bits 3 and 4 and use their content to perform special operations in the system. The first system-flag is the GIE flag.

GIE stands for General Interrupt Enable and is the bit that controls interrupt masking. If GIE set to 1, maskable interrupts are allowed to preempt the CPU and request ISR executions. If it is set to 0, the CPU ignores maskable interrupt requests. GIE is usually set during the system prologue and can be unset at any time by the user code.

The OFF flag triggers the system shut-down routine. Whenever set to 1, the CPU finishes its execution and bring the system to shut-down state. Despite its function in the system, OFF can be set at any time by the user code. The compiler automatically sets OFF at the end of a program, when the outermost return statement is reached by the CPU.

The last special register is the Constant Generator. Labeled as CG or R3, it can generate up to four different hard-wired values to be used as source operands. The Constant Generation is performed by combining the address of R3 with addressing modes. In Fact, R2 can also be used as so, and a detailed description will be presented in Section 3.1.7.

The Register File architecture denies attempts to write in CG. Reads will always output zero and any usage as source operand will output the constant value referred to the addressing mode used in the instruction. Using CG as a destination operand will result in content loss.

The remaining twelve registers can be freely used by user code. The compiler will automatically allocate variables and store content in the registers R4 to R15 in order to avoid directly handling special register. The Common Registers can also be used to store function parameters and to provide other minimal functionalities that will be detailed further in this document.

3.1.4 Endianness

Endianness is the term used to refer to the order in which the bytes of a value larger than one byte are stored in memory [42]. It can have critical effects over integers and strings once the byte ordering can drastically change binary number value. Arrays and structs, however are not affected by the endianness.
3.1. ALCHEMY CPU ARCHITECTURE

The architecture memory systems are little-endian, as defined by MSP430 ISA. This means that the Register File and the Memory Controller will always store the least significant byte in a word (or double-word) in the lower-memory address. Figure 3.5 shows a word organized in memory.

\[
\begin{array}{c|c}
0xFE & \text{address: A + 1} \\
0x01 & \text{address: A} \\
\end{array}
\]

Figure 3.5: Little-endian memory organization model.

In the figure, the value 0xFE01 is stored in memory by dividing it into two 8-bit parts. Once 0x01 is the least-significant byte, it is stored in address A. Likewise, 0xFE is the most significant byte of the number and will be stored in a larger address (A + 1).

The CPU was designed to maintain this access consistency. This means that all reads and writes implements this convention and that the user must take it into account in order to design correct programs. Note that this concern will only fall upon programs directly written in assembly, once the compiler abstract this convention when compiling higher level C code.

3.1.5 Addressing Modes

An addressing mode refers to how the CPU accesses its instructions operands. It also describes the way operands are fetched and changed after the execution of an instruction. Source operands (src) can be used in one out of four possible ways. Destination operands (dst) can only be referenced in two possible ways. Note that source and destination may not be included in all instructions.

There are seven addressing modes and a single-pseudo addressing mode. It is not defined any new addressing mode, when compared with MSP430.

- **Register-direct Addressing Mode**: Register-direct is Alchemy’s simplest addressing mode. In it, the content of a requested register Rx is used as the operand. The content refers to the 16-bit binary value stored in the Register File, in the address of Rx. This addressing mode is available for both source and destination operands.

- **Indexed Addressing Mode**: In Indexed mode, the operand address in memory is computed based on a given target register content and a 16-bit offset. The target register address is coded into the instruction word while the offset is searched by the CPU in the next valid address in memory. The effective address is then computed by adding the target register content to the fetched offset. This addressing mode is available for both source and destination operands.

- **Symbolic Mode**: In Symbolic addressing mode, the base register is steadily selected as the PC. This means that the offset must be selected taking into account the address of the current instruction in the memory, which is a static value, instead of volatile register contents. This provides much freedom to the constants and global variable allocation process, that can be moved without needing to drastically change the code. This addressing mode is available for both source and destination operands.
■ **SP-based Addressing Mode**: In SP-based Addressing Mode, the stack pointer (SP) is used as the base register when fetching operands. It turns out that this addressing mode provides a way to directly access the stack entries, without having to pop content out of it. This addressing mode is available for both source and destination operands.

■ **Register-indirect Addressing Mode**: Register-indirect mode allows registers to hold pointer to data stored in the Shared Memory. When making references to memory positions through Register-indirect mode, the register content is directly forwarded to the MMU which in turn fetches the operand in memory. The address held by the register is still referred to as effective address. This addressing mode is available for only source operands.

■ **Register-indirect Auto-increment Addressing Mode**: Register-Indirect Auto-increment mode is a more general case of Register-indirect, in which the register content is used as a reference to memory and incremented afterwards. It can be said that Register-indirect mode has an increment of zero after each reference fetch, while Auto-increment mode provides non-null increments. This addressing mode is available for only source operands.

■ **Immediate Addressing Mode**: In immediate mode, the register is stored in the next word following the instruction’s. The operand is used as read from memory and no further changes on its value are required. Once the program counter is incremented by two right after fetching the current instruction, it always points to the next word. If the source operand is meant to be immediate, the fetch processes consists of loading the next word and incrementing the PC by two, afterwards. This brings the PC to the right position in memory and makes the fetch of the next operand or next instruction concise. This addressing mode is available for only source operands.

### 3.1.6 Alchemy Instruction Types

The instruction set architecture is the part of a computer system that the programmer or the compiler must understand to write a correct program. Alchemy Instruction sets provide consistent machine instructions that combine registers, addressing modes and memory to implement the CPU functionalities.

Alchemy Instruction Set is divided into three classes. **Native Instructions** are a set of 27 instructions that are directly implemented in the CPU hardware. There are defined unique opcodes and instruction fields for these instructions. An extra set of 24 **Virtual Instructions** combines addressing modes and Native Instructions to implement new instructions. These instructions do not have any actual logic reserved for their implementations. The last instruction class is composed by the **BRIDGE Instructions**. These are five instructions used to control the BRIDGE in order to interact with the FPGA. BRIDGE instructions have specific opcodes and require special logic in the CPU.

The bits in the instruction words are classified according to a set of special fields as presented in Figure 3.6. These fields must have correct bit values embedded into them in order to provide a valid instruction. Wrong value in fields may generate unexpected behavior or Invalid Instruction Exceptions during runtime.

The figure shows the fields encoded in the instruction words. The fields were labeled according to their actual function in the instruction. To attain simplicity, there are defined only 8 fields which will be presented next.
3.1. **ALCHEMY CPU ARCHITECTURE**

Native Instructions

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Positions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Instruction</td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>15</td>
</tr>
<tr>
<td>OP</td>
<td>0</td>
</tr>
<tr>
<td>10-bit Immediate</td>
<td></td>
</tr>
</tbody>
</table>

Single Operand Instruction

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Positions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>15</td>
</tr>
<tr>
<td>OP</td>
<td>0</td>
</tr>
<tr>
<td>MD</td>
<td>0</td>
</tr>
<tr>
<td>AS</td>
<td></td>
</tr>
<tr>
<td>Source</td>
<td></td>
</tr>
</tbody>
</table>

Two Operands Instruction

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Positions</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>15</td>
</tr>
<tr>
<td>Source</td>
<td>0</td>
</tr>
<tr>
<td>AD</td>
<td></td>
</tr>
<tr>
<td>MD</td>
<td></td>
</tr>
<tr>
<td>AS</td>
<td></td>
</tr>
<tr>
<td>Destination</td>
<td></td>
</tr>
</tbody>
</table>

BRIDGE Instructions

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Positions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write BRIDGE Register Instruction</td>
<td></td>
</tr>
<tr>
<td>OP</td>
<td>15</td>
</tr>
<tr>
<td>Source</td>
<td>0</td>
</tr>
<tr>
<td>RSV</td>
<td></td>
</tr>
<tr>
<td>OP</td>
<td></td>
</tr>
<tr>
<td>AS</td>
<td></td>
</tr>
<tr>
<td>RSV</td>
<td></td>
</tr>
<tr>
<td>BDST</td>
<td></td>
</tr>
</tbody>
</table>

Read BRIDGE Register Instruction

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Positions</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>15</td>
</tr>
<tr>
<td>RSV</td>
<td>0</td>
</tr>
<tr>
<td>BSRC</td>
<td></td>
</tr>
<tr>
<td>AD</td>
<td></td>
</tr>
<tr>
<td>OP</td>
<td></td>
</tr>
<tr>
<td>RSV</td>
<td></td>
</tr>
<tr>
<td>Destination</td>
<td></td>
</tr>
</tbody>
</table>

BRIDGE Scheduling Instructions

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Positions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>15</td>
</tr>
<tr>
<td>OP</td>
<td>0</td>
</tr>
<tr>
<td>RSV</td>
<td></td>
</tr>
<tr>
<td>OP</td>
<td></td>
</tr>
<tr>
<td>TINPUT</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.6: Alchemy Instruction Fields.
Branch instructions have the fewest number of fields. There are two fields that encode the type of the branch (OP) and the branch signed offset (10-BIT IMMEDIATE). OP is a three bit field and stands for opcode while the other field is self-explanatory.

Single Operand Instructions encode their functionalities by using four fields. OP encodes the type of single operand instruction. MD or Mode informs whether the CPU will run in byte or word mode. Mode holds a single bit in which 0 encodes word mode and 1 encodes byte mode.

AS (Addressing-Source) is a special field for source addressing mode encoding. It uses two bits to inform how the source operand should be fetched. When set to 00 the source is fetched in Register-Direct mode. When set to 01, the source operand is fetched with Indexed-Mode, being possibly Symbolic, if the base register is the PC or Absolute, if the base register is SP. When set to 10 the source operand is fetched in Register-indirect mode. If AS is equal to 11 the source operand is fetched with Register-Indirect Auto-increment mode or Immediate (when the base register is the PC). The last field, called SOURCE, encodes Register File register with four bit words.

Two Operand Instructions require fields to encode both source, destination and their respective addressing modes. The OP field for this kind of instruction requires four bits. Likewise, SOURCE and DESTINATION uses four bits each to encode Register File registers for the source and destination operands, respectively.

The operand addressing modes are encoded with AD and AS. AD (Addressing-Destination) refers to the Destination Addressing mode and only uses one bit of the instruction word. AS does the same for source operand. Note that due to the lower number of bits used in AD, destination operands have fewer addressing modes. When AD is equal to 0 the destination operand is fetched with Register-Direct mode. If it is equal to 1 the operand is fetched with Indexed or Symbolic mode, depending on the base register.

AD can also be used to encode constants with the Constant Generator Feature. By combining its value with the source base register, the Register file can output various constant values directly to the ALU. When AS is equal to 01 and the base register is SR the source operand is loaded as 0x4. When AS is equal to 10 and the base register is SR the source operand is loaded as 0x8. When AS is equal to 00 and the base register is CG the source operand is loaded as 0x0. When AS is equal to 01 and the base register is CG the source operand is loaded as 0x2. When AS is equal to 11 and the base register is CG the source operand is loaded as -0x1.

MD encodes the instruction mode (byte or word mode), like presented for Single Operand Instructions.

Write BRIDGE Register Instructions are encoded with four fields and a few reserved bits that should not be changed. This instruction has an OP field that requires five bits and which is split into two parts: four bits are placed in the most-significant bits of the word while a single bit is placed in the MSB of the lower instruction byte.

Since this instruction is intended to write content in the BRIDGE register, a Register File register and a source addressing mode must be provided. This register is encoded with the four-bit SOURCE field and the addressing mode is encoded with the AS field. The source is fetched as any standard operand, taking into account the addressing mode provided in the instruction. A BDST or BRIDGE Destination field uses two bits to refer to a specific BRIDGE register.

Read BRIDGE Register instruction follows the same basic structure presented in Write BRIDGE Register instruction. The only difference are the position of the BRIDGE Register, the Register File registers and the addressing mode bit. In this case, the SOURCE field is replaced by a reserved two-bit field and a two-bit BSRC (BRIDGE Source) field which encodes the BRIDGE register to be read. The BDST field (and its preceding two reserved bits) are replaced by a DESTINATION which encodes a Register File register. AD field encodes the one-bit
3.1. ALCHEMY CPU ARCHITECTURE

BRIDGE Scheduling Instructions is used to emit an FPGA Cross-Procedure call. The instruction requires a five-bit OP field split into one part of three bits and another of two bits. TINPUT field encodes the operation type by using four bits.

BRIDGE Destroy Instruction is a framework to abort an FPGA Cross-Procedure call. The only difference from Scheduling Instructions is the two-bit reserved field and that no operands are needed in TINPUT and OP which remain as 0000 and 111, respectively.

The figure also shows a few RESERVED or RSV fields. These are static fields that cannot be changed. They can either be null or preset values that define the instruction structure and may generate an illegal instruction exceptions if not kept consistent.

Based on these six basic instruction formats, all 27 Native Instructions, all 24 Virtual instructions and all 5 BRIDGE instructions are implemented. This section provides a basic overview of the Instruction Set.

Instructions have a distinct two step path in the CPU. A special block called Instruction Decoder is responsible to break a given instruction word into its fields and inform the Control Unit about the actions to be performed in order to correctly execute the instruction.

Once the CPU identifies, decodes and fetches the instruction operands, the ALU (Arithmetic-Logic Unit) is summoned to perform the actual computation. The ALU has a direct path to the Instruction Decoder in the Local Control Bus which allows it to identify which instruction operation is meant to be performed. Once the computation is completed, the ALU outputs its results to the Local Data Bus, allowing the target destination to be written afterwards.

### 3.1.7 Instruction Decoder

The Instruction Decoder is the CPU block responsible to decode instructions and inform other blocks about the procedures required to execute that instruction. It mostly breaks a given instruction word into the fields presented in Section 3.1.7 and converts them into a flow of control signals forwarded to the Control Unit, register File and ALU. Figure 3.7 shows the Instruction Decoder basic structure.

The instruction words are forwarded directly from the memory to the Instruction Register. The word is firstly broken into three major sections which are used to identify the instruction

---

Figure 3.7: Achemy Instruction Decoder.
type. The first section holds the six most significant bits of the word. These bits are used to determine if the instruction is a Branch instruction, as presented in Section 3.1.6.

Two other sections are extracted simultaneously from the word. The nine MSB of the word are taken to determine if the instruction is a Single Operand Instruction. The four MSB of the word are also analyzed to check if the instruction is a Two Operand Instruction. BRIDGE Instructions may fall in one of these three basic groups so no special analysis is performed for them.

If the instruction word corresponds to a valid instruction, exactly one of the extracted field will acknowledge a match. If the fields could not be identified by the Decoder, the machine cycle will be lost and the CPU will stall for four clock-cycles.

After the identification of the instruction, the actual feature fields are extracted and analyzed. Once the first decoding stage also extracts the instruction opcode, no further identifications are needed. The Instruction Decoder breaks the word in the fields presented in Section 3.1.6, according to the instruction type.

These fields are then forwarded to other CPU blocks that may use this information to execute the instruction, when they become active within the machine-cycle. The Instruction Decoder sends the feature fields by either using the Local Control Bus, The Local Data Bus or the CB Bus, which are over its control during the Instruction Decoding Stage.

The Instruction Decoder should be called exactly once per machine-cycle. It always works over full 16-bit words, regardless the CPU mode. Unaligned memory access is potentially hazardous for the instruction decoder once any misplaced field or invalid values may generate and Illegal Instruction.

3.1.8 Arithmetic and Logic Unit

The Arithmetic and Logic Unit is the block in which most of the instructions are actually executed. It implements most of Alchemy’s Instruction Set and provides a simple control interface to the Control Unit. Once the ALU implementation may widely vary depending on the abstraction level, this section will present an overview of its functionalities.

The ALU has a pair of input operands and an output register to store the instruction results. Figure 3.8 shows the simplified ALU structure. It can be divided into three sections. The first one is the input set, the second is the logic and the third is the output register. The logic can be divided into two minor sections which are the type selector and the instructions logic.

The Input Set is composed of two 16-bits registers which hold the source and destination operands. These registers are meant to be used at the moment when the instruction is to be executed, but it also can hold intermediate values from operand fetches.

The ALU inputs can be set to come from various sources. If the register content should not be used, the operand input is configured in Bypass mode, so values coming from the input multiplexers are directly forwarded to the ALU and the register content is ignored. This is used when intermediate summations are performed during indexed operand fetches (for example).

Prior to the register is a four input multiplexer which provides different operand values from different sources. These multiplexers are controlled by the control unit and their select signal can point to either the Local Data Bus, the Register File Outputs, the Program Counter output, the ALU output, a positive 2 constant value and the Instruction Decoder 10 bit branch Offset.

The ALU executes instructions in its logic section. This is a wide set of logic elements which uses the input operands and produces a 16-bit result plus a set of four status flags. This block should be designed so that each operation fits the machine-cycle duration. Alchemy may
3.1. ALCHEMY CPU ARCHITECTURE

reserve any amount of clock-cycles for their execution. The cycle duration may depend on the instruction itself.

The operands are directed to one out of three locations. A set is selected based on a signal provided by the Instruction Decoder which informs the ALU if it is a Brand Instruction, a Single Operand Instruction, a Two Operands Instruction or a Bridge Instruction. Once the operand is forwarded to the correct location, the ALU defines which logic should be applied to it. A special signal from the Instruction Decoder is used to identify the logic to be used and the instruction is executed afterwards.

Once the instruction is executed, a final section is used to extract the status flags. As presented in Section 3.1.3, the status flags are stored in register $SR$ and the ALU reserves a direct path to the Register File in order to update them. The flags can be generated during the instruction execution (like Carry and Overflow) or can be extracted from the instruction result (like Negative and Zero).

After the instruction execution, the ALU stores the result in an output register which is connected to the Register File, the MMU and the BRIDGE. The ALU also writes the status flags back in the status register and informs the Control Unit about the execution completion.

3.1.9 Control Unit

The Control Unit is the management structure in Alchemy CPU. Like a maestro who conducts an orchestra, the Control Unit mostly coordinates the execution of other CPU elements. The Control Unit is organized in a FSM model which performs transitions at each machine cycle.

The Control Unit handles the Local Control Bus and most of the signals that are in it. These signals are used by the CPU modules to determine which action are meant to be performed and when they should take place. Alchemy has a wide set of control signals that come either from or to the Control Unit. The Local Control Bus is used by the CPU blocks to report their status back to the Control Unit, so that it can plan the next execution stage.

In an architectural view, the Control Unit consists of a finite state machine that is in one out of six states at each clock-cycle. The control unit executes four basic operations: fetch, decode, execute and write. These operations are organized into a sequence called "machine-cycle". Each machine-cycle must be completed in order to execute an instruction. The architecture defines that a machine cycles requires at least four clock-cycles.
Differently from pipelined machines, in which instructions are completed in a steady pace of generally one per clock-cycle, Alchemy Control Unit executes instructions in uneven intervals, implementing a multi-cycle CPU. This design decision can directly affect the CPU performance. Depending on the addressing mode or type, instructions can take up to 12 clock-cycles to be executed. The Control Unit also deals with System Interrupts. The Interrupt Controller uses special Control Unit register to inform the occurrence of interrupts.

It is important to realize that the Control Unit has only control over the user program. The FPGA configuration control and its execution are out of its scope. There are defined extra blocks for this special case. The Control Unit is restricted to the Program Memory domain and to the dynamic data generated by the instruction execution (stored in either the Register File, the Shared Memory and the BRIDGE registers). Figure 3.9 shows Alchemy Control Unit Finite State Machine.

In the figure, it can be seen that the Control Unit always starts at **Contention** state. The CPU will stay in this state as long as the FPGA is still being configured, and no actual work is performed. This busy wait aims to ensure runtime correctness and cannot be avoided by user code. **Contention** keeps the CPU steady until a FPGA_CONFIG_COMPLETE signal is emitted by the Configuration Controller.

The **Contention** state can only be immediately left if the system initialization is not bound.
by the FPGA configuration. In that case, it takes one clock-cycle to the Configuration Controller to report the Control Unit that the FPGA configuration is meant to be skipped. From that point the CPU is brought to the **Reset** state and an actual machine-cycle is started.

**Reset** is a typical initialization state. It is used to minimally place the Control Unit in a *ready* state so further instructions can be executed. **Reset** usually clears the CPU interrupt Buffer, requests the Register File, and clears several Control Unit control registers. Power-up (when the FPGA is not meant to be programmed) and Reset events must bring the CPU to **Reset** state.

When the CPU is ready to run the user code, the Control Unit is prompted to **Fetch** state. **Fetch** may incur in a group of search operations for either instructions and operands. Once every machine-cycle begins with **Fetch** state, a few other operations are performed when in this state.

The CPU verifies the occurrence of interrupts during its **Fetch** period. Furthermore, memory and register file signals are emitted during **Fetch**. There are eight fetch types that are requested by the previous states and that should be correctly identified and executed when the Control Unit is at **Fetch**.

If the CPU is meant to fetch an instruction, the **Fetch** state is sent to **STANDARD_FETCH**. In this case, the CPU is starting a new machine-cycle, so the Interrupt Buffers must be checked in order to determine whether an ISR should be executed. The CPU also requests a memory read and goes to the **Decode** state.

When fetching immediate source operands (Section 3.1.5), formally an **IMMEDIATE_FETCH**, the CPU must read the next memory word and send its content to the ALU. The Control Unit does so by informing the ALU that the source operand to be used is read from memory and requesting the Memory Management Unit to perform a memory read. The next machine state is set to **Execute and Analyze**, when the instruction is actually executed.

Indexed addressing uses three different fetch types to be fully implemented by the CPU. When fetching indexed offsets for either source or destination, the fetch type is called **INDEXED_FETCH** and a memory read is requested like in immediate operand fetches. The base register content is requested to the Register File by the Control Unit. The CPU is prompted to an intermediate state in which the effective address is computed and placed at the Address Bus. This state is called **INTERMEDIATE_SUM**.

When in **INTERMEDIATE_SUM**, the effective address is computed. Internal registers inform the Control Unit if the base register is provided from the source or destination operand. The Control Unit requests the ALU to perform the sum and places the result in the Address Bus. The Control Unit is then brought back to **Fetch** and depending on the targeted operand, the fetch type is set to **INDEXED_SOURCE_OFFSET_FETCH** or **INDEXED_DESTINATION_OFFSET_FETCH**.

Back in **Fetch**, the Control Unit requests the memory to read the operand in the effective address. This operand is set to be either the source or destination operand in the ALU. If the instruction requires a second indexed operand, the same fetch procedure is performed, this time aiming the destination operand. If there are no further indexed fetches, the CPU is sent to **Execute and Analyze** state.

Another type of fetch is the **INDIRECT_FETCH**. This addressing mode requires the Control Unit to request the Memory Management Unit to read a memory position in the address held by any of the Register File Registers. The memory content is then directly forwarded to the ALU source. The State Machine is then forwarded to **Execute and Analyze** state.

A special case of **INDIRECT_FETCH** is the **INDIRECT_AUTOINCREMENT_FETCH**. In this fetch type, there are performed the same procedures required for an **INDIRECT_FETCH** but a special

---

5When there are two indexed operands, the source is always fetched before the destination.
TARGET_INCREMENT signal is sent to the Register File. Right after the register content is sent to the MMU, its content is incremented by one or two, depending on the CPU mode. No further operations are performed during this fetch type.

When the Interrupt Controller informs that an interrupt have happened, the Control Unit executes an ISR_FETCH. In this case, the ISR address provided by the Interrupt Controller should be written in the PC. Hence, the Control Unit must request the Register File a write operation and send the ISR address to the PC, so that it can be executed in the next machine-cycle. The Control Unit is sent to Write Back state in order to perform the write.

During the Fetch state, the CPU also checks if there are Shut down or Idle requests. If an Idle state is requested, the CPU stays in this state, checking for the Reset interrupt flag on the Interrupt Buffer. If the CPU is requested to shut down (bit 5 of RS), the Control Unit stays in Idle for a single clock-cycle and then it is prompted to OFF.

Apart from the Fetch stage, the CPU must decode each instruction it reads from the memory. The Decode stage Interacts with the Instruction Decoder in order to identify and extract the instruction fields. These fields are then sent to their respective targets with the CPU.

The only work performed when in Decode state is the decoding of the word read from memory in Fetch state. The instruction decoding procedure should only be executed once per machine-cycle and it is always followed by Execute and Analyze state.

Execute and Analyze state is a more complex state in the FSM structure. It performs three classes of operations. It determines if the instruction operand should be fetched with special addressing modes, checks in the BRIDGE if FPGA Cross-Procedure call are requested and asks the ALU to execute the actual instruction.

By using special signals from the Local Control Bus, the Control Unit determines if extra fetches should be executed. In this case, it is verified if only one or both operands are meant to have its content fetched and a special control register called FETCH_STATE informs the fetch procedure status. Execute and Analyze checks and controls this register in order to coordinate the fetching process.

It is during this state that the Fetch types are decided and the Control Unit is sent back to Fetch in order to load the operands. The FETCH_STATE register consists of four bits which inform if the source and destination should be fetched and which is the operation status. If all operands are fetched, the status bits in the register are set to one, and the CPU can actually execute the instruction. The Control Unit is sent to Fetch state afterwards.

When allowed to execute the instruction, Execute and Analyze first checks if the instructions is meant to decrement the Stack Pointer. In this case a special signal is sent to the Register File in order to decrement SP. The Control Unit also sends the instruction type and opcode to the ALU, enables the operands to be used by the ALU and sets the next state to Write Back.

If the Instruction Decoder identifies an FPGA Cross-Procedure call request, the Control Unit must watch the BRIDGE execution and request it be read or written, depending on the intended operation. This is performed during Execute and Analyze when BRIDGE signals are asserted and read. During this procedure, a special BRIDGE_STATUS register is set to inform which is the operation undergoing in the BRIDGE. This register is used to decide which will be the Control Unit state in the next machine-cycle. Like in standard executions, the Control Unit is also brought to Write Back afterwards.

The last step in any machine-cycle is the Write Back state. It is used to allow instruction results to be stored in either the memory or Register File, to update control registers, like SP and SR and to send the CPU to a wait state, when FPGA sequential Cross-Procedure calls are performed.
3.1. ALCHEMY CPU ARCHITECTURE

Write operations are divided into two groups. When the operand is meant to be written in memory, the Control Unit requests the MMU to write the ALU output in a cached address which will be either the one used to fetch the source (if the executed instruction was a Single Operand Instruction) or the destination (for Two Operand Instructions). When the operands to be written resides in the Register file, the Control Unit asks the Register File to write the ALU output in the write address.

If BRIDGE_STATUS flags sequential FPGA execution request, no writes are performed and the next state is set to WAIT state. In WAIT state, the CPU is put in a hold and only Interrupt Buffers are checked. This state is only executed if the FPGA executes in sequential Mode. When the FPGA is running parallel to the CPU, machine-cycles are integrally executed. If an FPGA_RETURN interrupt is flagged by the Interrupt Controller, the CPU sets the FETCH_TYPE register to ISR_FETCH and transits back to Fetch state.

Since the OFF state requires a lower level abstraction to be implemented, the state is defined but not implemented in this project. OFF should be used to cut the power-supply path from the processor, turning a large part of CPU circuit off in this process.

The Control Unit is directly tied to a State Controller block which provides clock signal. The Control Unit can only be halted if the State Controller blocks the clock signal to change, making the FSM transition impossible.

Considering the machine structure presented in Figure 3.9, it is possible to derive the exact amount of clock cycles required by each possible instruction execution mode. The entries in Table 3.2 represent the total number of clock-cycles in order to complete the instruction execution.

Table 3.2: Control Unit Instruction Execution modes timing

<table>
<thead>
<tr>
<th>Destination → Source ↓</th>
<th>Register-Direct</th>
<th>Indexed</th>
<th>Absolute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register-Direct</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Immediate</td>
<td>6</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Indexed</td>
<td>8</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Register-Indirect</td>
<td>6</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Register-Indirect Autoinc.</td>
<td>6</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Absolute</td>
<td>8</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Symbolic</td>
<td>8</td>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>

In the table, it is possible to see that Register-Direct operands is the fastest addressing mode, requiring only four clock-cycles when both operands are fetched this way. Furthermore, when both operands are fetched in Indexed mode, the fetch overhead reaches its largest value. Table 3.3 shows the timing of other important CPU operations.

The time references presented in both Tables 3.2 and 3.3 are valid references for whatever implementation of the architecture. Both the TLM and the RTL models presented in sections 6.1 and 6.2 respectively, describe their Control Unit as presented in this section.
Table 3.3: CPU operations Timing

<table>
<thead>
<tr>
<th>Operation</th>
<th>Clock-Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt</td>
<td>10</td>
</tr>
<tr>
<td>Idle State</td>
<td>4</td>
</tr>
<tr>
<td>Off State</td>
<td>5</td>
</tr>
<tr>
<td>FPGA Call</td>
<td>5</td>
</tr>
<tr>
<td>FPGA Return</td>
<td>11</td>
</tr>
<tr>
<td>Reset</td>
<td>1</td>
</tr>
<tr>
<td>BRIDGE Read/Write</td>
<td>4</td>
</tr>
</tbody>
</table>

3.2 Alchemy FPGA Architecture

Alchemy FPGA is a SRAM configurable device based on Xilinx XC2000 architecture, which enables the user to define and embed new device functionalities during the system startup time and use them as standard logic circuits, without the necessity of expensive mask/layout fabrications. The FPGA architecture was designed to provide feasible logic capabilities, being equivalent to about 6000 logic gates depending on the user configuration [16]. This logic capacity was chosen thinking of the balance of area and delay, already seen in other Xilinx products. Figure 3.10 summarizes Alchemy FPGA architecture.

![Alchemy FPGA architecture](image)

Figure 3.10: Alchemy FPGA architecture.

More than a sole reconfigurable matrix, Alchemy FPGA includes a configuration controller, a simplified memory controller and a BRIDGE interface bus. The Configuration Controller is a state-machine driven programmer, that requests bitstreams from the Configuration Memory and forwards them to the configurable elements in the FPGA. The Memory Controller implements a direct communication mechanism with the Configuration Memory, based on requests from the Configuration Controller. It is directly tied to the global Data and Address buses.

FB Bus connects the FPGA to the BRIDGE so that data and control signals can be
exchanged with the CPU. Most of the elements in the FPGA are only used during the system start-up and become idle afterwards.

### 3.2. FPGA Matrix Organization

Alchemy FPGA matrix consists of an Island-Style device which is a mesh interconnection of Logic Blocks, Configurable Interconnects and I/O Blocks. According to [46], Island-Style FPGAs are highly *tileable* devices, which means that hardware modules can be easily deployed to the mesh and replicated through the structure, easily supporting re-use based design approaches. Furthermore, CAD tools can be much easily developed for this kind of FPGAs. The high regularity of the structure allows simpler technology mapping, placement and routing and configuration bitstream generation.

The term "Island-Style" comes from the typical physical organization of the device. By analyzing the diagram shown in Figure 3.11, it can be seen that the Logic Blocks resemble *islands* surrounded by a *sea* of interconnects. In the figure, 1 shows a Logic Block and its surrounding routing elements. 2 shows an I/O Block and an interconnect. The FPGA matrix depicted in the figure intends to illustrate the FPGA matrix structure and is not an accurate representation of its organization.

In Island-Style FPGAs, each logic block is classified according to a 2D coordinate system and is directly tied to a surrounding set of interconnects. Conversely, each I/O Block is classified in an 1D ordinate system, organized in four sectors: *EAST*, *NORTH*, *WEST* and *SOUTH*. The interconnects are spread through the four Logic Block sides (classified as up, down, left and right). The user configuration enables or disables switches between the Logic Block pins and the interconnects in order to configure a given functionality.

Despite providing important advantages over other architectural approaches, Island-Style FPGAs are usually prone to describe routing-hungry configurations and can suffer from delays due to the vast interconnects. Alchemy FPGA deals with this routing architecture drawbacks by making it available a high amount of pins per logic block, placing 33 different pins throughout its structure. This enables a large number of routing possibilities for a given design.

The interconnect *sea* is composed of evenly distributed horizontal and vertical single-length wire segments interleaved with Switch-Matrices. Each interconnect forms a routing channel that
can have up to four wires carrying signals in both directions. Each of these wires holds two Connection Boxes, which are groups of four switches that connect the Logic Block pins to the actual interconnect.

The connectivity between the Logic Block input pins and the Connection Boxes is called $F_c(in)$ while the connectivity of output pins is called $F_c(out)$. An $F_c(in)$ equal to 1.0 means that all the input pins of a given side of a Logic Block are connected to all wires composing the adjacent routing channel. Likewise, $F_c(out)$ equal 1.0 means that all the output pins of a given side of a Logic Block are connected to all the wires composing the adjacent routing channel. Alchemy’s $F_c(in)$ and $F_c(out)$ are equal to 1.0.

Horizontal and vertical wire segments intersect one another by means of a Switch-Matrix. Each Switch-Matrix is a set of configurable switches that directly tie different interconnect channels. Alchemy implements a Subset Switch-Matrix architecture that allows signals traverse through both horizontal and vertical directions, by preserving the signal channel number. This means that a signal coming from a given channel (said $ch_i$) can only be forwarded to a segments in the channel $i$ in the remaining segments.

There are two sets of global-range interconnect channels used to control the Logic Blocks. A global reset channel broadcasts reset signals to all FPGA registers requesting the flush of registers values. Clock signals are also organized in global interconnect and are evenly distributed through all Logic Blocks.

Island-Style FPGAs circuit area is determined by the size of its routing resource. Configurable interconnects usually occupies 80%-90% of the total circuit area, while only 10%-20% of the area is used for actual logic implementation [43].

At the corners of the structure are the I/O Blocks. They are directly connected to interconnects through Connection Boxes and also tied to registers in the BRIDGE. Its primary function is to allow the FPGA to interface with the CPU and the Shared Memory. The FPGA surroundings are filled with I/O Blocks in all four directions. This setup allows better routing capabilities once circuit routes can widely spread through the routing resources.

### 3.2.2 FPGA Modes

From the system startup, the FPGA can be configured to be in one out of four possible states. These states describe how the device is meant to interact with the CPU. When running in Master Mode, the CPU is not allowed to run and the FPGA has full control over the system.

When set to be the CPU partner (Partner Mode), the FPGA is configured in the system startup time and the CPU will summon it on demand. This creates a processor + co-processor relationship where the FPGA is sleeping most of the time and the CPU wakes it up as needed. Despite having no control over the wake up time, the FPGA can decide when it will return to sleep by emitting Return interrupts.

In Independent Mode, the FPGA and the CPU are set to run independently from one another. The FPGA is triggered from the system startup, right before the configuration phase. From that point the CPU and the FPGA continue their execution as two independent blocks, continuing as so until the system is shut down.

Off state denies the FPGA execution. This mode potentially improves the system start up time since no FPGA configuration is performed. Both the FPGA circuitry and the Configuration Memory are shut down. This mode is useful when no application-specific operations are needed according to the user requirements.

Memory accesses are not prioritized. This means that neither the FPGA nor the CPU have privileged memory access and no hierarchical memory structures are defined. Memory accesses are queued so that the access policy is based on the first-in first-out principle. Furthermore,
3.2. ALCHEMY FPGA ARCHITECTURE

the FPGA does not have its internal configurable logic categorized according to access levels. This means that no special privileges are needed to use the full configurable resource.

The FPGA includes a simple controller called State Controller which enables or disables the configurable unit depending on the user requirements. Two bits are used to encode the FPGA states, as presented in Table 3.4.

<table>
<thead>
<tr>
<th>Binary Code</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master Mode</td>
<td>11</td>
</tr>
<tr>
<td>Partner Mode</td>
<td>01</td>
</tr>
<tr>
<td>Independent</td>
<td>10</td>
</tr>
<tr>
<td>Off</td>
<td>00</td>
</tr>
</tbody>
</table>

Table 3.4: FPGA modes and corresponding State Controller encoding.

A pseudo-mode called Slave Mode can be derived from Partner Mode, by implementing a simple control logic in both the CPU and the FPGA. When running in Slave Mode, the FPGA runs freely while the CPU is in a busy-wait loop. The FPGA requests the CPU to process specific data (passed through BRIDGE registers) on demand, regaining the system control afterwards. The FPGA calls the CPU by means of return interrupts signals.

At the system startup, the CPU emits a parallel or sequential FPGA call and waits for FPGA return interrupts. When the interrupt is detected, the CPU executes the ISR by copying the BRIDGE registers and performing the specific computations defined by the user. It then writes the results back to the BRIDGE registers.

After returning from the ISR, the CPU resumes the FPGA execution and return to the busy-wait state, until the FPGA emits another request. The FPGA halts its execution while the CPU is processing the requested data. The FPGA must read the BRIDGE registers in order to use the results processed by the CPU.

3.2.3 FPGA Signal Model

Data and control operations can be performed in the FPGA by using signals. Signals are single bit values that are in each interconnect, Connection Box, Switch-Matrix and Logic Block, at any given time.

There are defined two signal types: Simple Signals and Signal Clusters. Simple Signals are formed up by a single binary value. A Simple Signal can assume either 0 or 1 and can be written in any FPGA component. High level hardware description languages define Simple Signals as either wire, signal or reg variables, depending on the intended functionality.

Signal Clusters are groups of signals formed by Simple Signals. In an HDL context, Signal Clusters are arrays of basic types such as wire. Signal Clusters have both length and value. A Signal Cluster length is the number of Simple Signals that form the cluster. The Signal Cluster value is the binary number encoded by its Simple Signals. It can assume any value from 0 to $2^N - 1$, where N is the number of Simple Signals in the cluster. Each Signal Clusters is used as a high-level abstraction for user configuration and have no physical implications on the FPGA hardware. Each Simple Signal in a cluster is labelled by an unique index that informs its position in the binary number.

Logic Blocks and I/O Blocks are often the source of signals in the FPGA. Logic Block change signal values by outputting binary data to interconnect, according to its inputs, register values and internal logic. I/O Blocks change signals when configured to be Input Blocks. In this case, it receive external binary signals and forwards them to the FPGA.
Interconnects, Connection Boxes and Switch-Matrices are the elements in charge to move signals through the FPGA. When in an interconnect, the signal is sent to all directions and switches connected to it. Signals are not allowed to be forwarded to closed switches.

Signals are not considered to have specific data types. This means that a signal can either be carrying an integer and a literal value at the same time. Signals are contextual entities in the sense the FPGA configuration determines their functionality. The FPGA may use signals to implement control logic, to hold values and to store temporary data.

### 3.2.4 FPGA Coordinate Systems

In order to correctly address its matrix elements, Alchemy FPGA is classified according to four different coordinate systems. Each CLB, I/O Block, Vertical Interconnects and Horizontal Interconnects are associated to an unique coordinate pairs \((X,Y)\), where \(X\) is the column number and \(Y\) is the row number. Figure 3.12 shows a sample FPGA coordinate systems organization.

CLBs are organized in a 2D matrix of elements in which each one occupies a single matrix entry. The CLB Matrix is a square matrix with 16 rows and 16 columns, resulting in a 256 entries matrix. The CLB matrix coordinate system is organized so that rows come from 1 to 16 and columns come from 1 to 16, as depicted in the figure.

Horizontal Interconnects are organized in a 2D, non-square matrix with 16 columns and 17 rows. The coordinate system convention states that rows are numbered from 0 to 16 while columns range from 1 to 16. Similarly, Vertical Interconnects are organized in another 2D matrix with 17 columns and 16 rows. Columns are addressed with coordinates ranging form 0 to 16 while rows are numbered from 1 to 16.

I/O Blocks are classified in unidimensional coordinate systems. In fact, they are part of a
2D system in which either the column or the row never change its value. I/O Blocks in the **EAST** sector have their rows varying from 1 to 16 while the column number is kept in 16. Similarly, I/O Blocks in the **WEST** sector have their rows ranging from 1 to 16 while the column is kept in 0.

For those I/O Blocks in **SOUTH** sectors, the row number is kept steady while the column must change. For both, column numbers vary from 1 to 16. **NORTH** sector I/O Blocks have their row number fixed in 17 while **SOUTH** I/O Blocks have rows fixed in 0.

Despite not depicted in the Figure, Switch Matrices are also classified in a 17 by 17 matrix with both coordinates ranging form 0 to 16. They are not depicted once most of the time they are implied form the coordinates of other FPGA elements, and cannot be directly addressed with Alchemy's Module Assembly language.

### 3.2.5 Configurable Logic Block

Configurable Logic Blocks (CLB) are the basic functional structures in Alchemy FPGA architecture, based on Xilinx XC2000 architecture. The Logic Blocks were designed to be area efficient while delivering high logic capacity for the user custom functionalities. A Logic Block is intended to contain very basic digital logic and storage elements which are meant to be combined with routing resources to provide actual logic capabilities. Figure 3.13 shows the basic Alchemy CLB structure.

Alchemy CLB is physically organized as a block with sectorial inputs and outputs pins. These pins are arranged in four distinct groups, in which each group has one input and one output. There are four local inputs, four local outputs and two global inputs.

The local inputs are labeled as A0, B1, C2 and D3, while the local outputs are labeled as either O1 or O2. The outputs labels follow internal CLB organization which will be presented in the following sections. The global inputs are SYN, which is a synchronization signal (formally a clock signal) and CLEAR, used to reset internal registers.

Despite considered fine-grained, Alchemy CLB was designed to attain a balance between
CHAPTER 3. ALCHEMY ARCHITECTURE

Figure 3.14: Alchemy Configurable Logic Block detailed architecture.

area and logic capability. Henceforth, the CLB is a LUT-based device which provides a good tradeoff between area, delay and logic capacity.

A LUT or Look-up Table is a SRAM block which encodes possible outputs for logic functions. It uses a set of $X$ inputs and $2^X$ memory bits to encode any $X$ variable logic function. The greatest advantages of LUTs are the reduced area required for its implementation and the constant delay to compute outputs for any given $X$ variable logic function. Figure 3.14 shows Alchemy CLB architecture:

The figure shows the basic structure of Alchemy CLB. The Logic Block has several configurable components which can be freely set during configuration time. These components are 27 multiplexers, two demultiplexers, two three-input look-up tables and one register.

I/O Interfaces

Each CLB input is tied to one multiplexer. INPUT_A, INPUT_B, INPUT_C and INPUT_D are four input multiplexers, which allow the selection of any CLB input to any LUT input. This first input selection layer adds more configuration possibilities for a single input than most of standard LUTs like those from [57].

Following the input selection stage, a set of four two-input multiplexers allows the user configuration to choose between the actual CLB input values and the internal CLB register value. This selection layer allows the implementation of sequential logic as well as basic storage capabilities.

An extra sub-selection layer allows a given input to be pointed to the next one, by implementing a chain-like connection structure. This allows an input to be synchronized with the one immediately following it, providing even more logic capabilities. The multiplexers responsible for this kind of connection are $I_{11}$ to $I_{18}$.

Multiplexers $I_3$ and $I_7$ are special multiplexers that can also be synchronized by the CLB
register itself. Multiplexers I4 and I8 are called ENABLE_MUXES once their primary functionality is to synchronize the LUT enable signal with either a constant 1 value (always enabled) or the value from multiplexer DMUX. I4 differs from I8 in the structure of its first input: it has an inverter to allow mutually exclusive enable behavior in the case in which I4 and I8 follow DMUX. This is fundamental when working with 4-input LUTs.

An extra enable option implemented with S1 and S2 allows the LUTs to be selected by the register, bypassing the mutually exclusive functionality provided by I4 and I8. The input multiplexers require 24 bits to be configured by the Configuration Controller.

Alchemy CLBs have two four-input configurable multiplexers placed in its outputs. The outputs of these multiplexers are replicated to address all CLB output pins. OUTMUX_1 is tied to outputs O1 and OUTMUX_2 is tied to outputs O2. Both multiplexers have their inputs tied to the same group of signals so that the synthesizer can choose between either synchronize all CLB outputs or provide different output behavior depending on the LUT configuration. The internal register can also be used as an output of the CLB so that its content can directly feed other blocks, like a storage element. A fourth and not depicted input pulls down the multiplexer output, avoiding unnecessary signal transitions, in the case a given output should not change.

### Look-Up Tables

LUTs can be used in either three- or four-input modes. Three-input mode can also be set to work statically or dynamically, depending on the configuration provided to multiplexers M2 and M3. The four-input LUT requires the two three-input LUTs to be configured complementarily so that the first half of the 16-bits of the expected outputs of the desired four-variable logic function is loaded into the first LUT, and the second half is loaded into the second LUT.

The toggle behavior provided by multiplexers I4 and I8 should mandatorily be used with four-input LUTs. This LUT setup creates the illusion of a single virtual four-input LUT for the user configuration. Four-input LUTs are, according to [35] the optimal LUT size for LUT-based FPGAs. They provide a better balance between area, propagation delay and logic capabilities and should ideally be used by FPGA designers.

Internally, LUTs are small memory resources of one byte in which each bit is addressable by an unique three-bit address. Each bit of this address represents one input of a three variable logic function. The memory must hold the expected output for each possible input combination, so that the LUT behaves exactly like any standard combinational circuit. Figure 3.15 shows possible LUT configurations provided by Alchemy CLB.

![Diagram](image)

Figure 3.15: Four-Input LUT setup.

Four-input LUTs have virtually four inputs and one output in which the fourth input can be
directly tied to the internal CLB register. Each input is copied to each three-input LUT input and the fourth input (enable input) is configured to work as an internal mutually exclusive pair. In this case, the LUTs will have their outputs mirrored.

When configured to work as two independent three-input LUTs, the internal structure of the selection multiplexers is arranged so that the synthesizer can assign different input variables to each LUT inputs. In this case, the fourth CLB input can also be used as the third variable, once the enable signal is fixed in one. For this setup, the outputs X and Y are different from one another, as seen in Figure 3.16.

![Figure 3.16: Independent Three-Input LUT setup.](image)

When using Independent three-input LUTs, the CLB is meant to provide different output pairs. In this case, the output multiplexer OUTMUX1 receives the value of the output of the first LUT (OUTPUT_X), while the multiplexer OUTMUX2 receives the output of the second LUT (OUTPUT_Y). The interconnects in the top and right sides of the CLB will in turn receive the content of OUTMUX1 while the left and bottom sides will receive the content of OUTMUX2. This setup can be changed by providing different configurations for the output multiplexers.

Yet another LUT setup is the Dynamically Selectable LUT pair. In this setup, both LUTs are enabled while the actual LUT outputs are mirrored. At any given time, only one of the two possible LUT outputs is forwarded to the CLB. The mirrored output is dynamically selected by one of the CLB inputs which is tied to a special output multiplexer. The Two LUTs receive the same inputs and their content must be different in order to allow this setup to be different from the Independent LUT setup. Figure 3.17 shows the Dynamic setup.

![Figure 3.17: Dynamically Selectable Three-Input LUT setup.](image)
3.2. ALCHEMY FPGA ARCHITECTURE

**CLB Register**

CLB Registers are simple edge-sensitive latches used to store a single bit in the CLB. They can be chained together (with proper interconnect interface) to provide internal memory capabilities to the FPGA. Their primary function however, is to allow the implementation of sequential logic alongside with the LUT.

In order to allow the synthesizer to control how the register will change its content, a set of internal clock-control multiplexers allows positive and negative clock edge sensitivity. It is also possible to completely disable clock changes by tying the syn input to the ground. In such a case, the register is not meant to be used as a memory bit, but as a constant value driver. The clock-control multiplexers also allows the clock signal come from different sources such as the global clock signal, the LUT and input \( C_2 \).

The CLB Register also provides a pair of asynchronous inputs that allow internal CLB logic elements to *force* the value stored in the register. Input \( \text{set} \) puts a logic 1 in the register whenever it is equal to 1. This input is in turn tied to a multiplexer that allows the synthesizer to choose between the LUT, input \( A_0 \) or the ground to drive \( \text{set} \). Likewise, input \( \text{clear} \) writes a logic 0 in the register whenever it is equal to 1. This input is driven by the LUT, input \( C_2 \) and the ground.

The register do have two complementary outputs. As a design directive, the negated output is ignored by the CLB and only the original output is forwarded to the output multiplexers. Figure 3.18 shows the Register simplified structure.

![CLB Register Diagram](image)

Figure 3.18: CLB Register. Input \( D \) is the synchronous data input, input \( S \) is the asynchronous set input, input \( R \) is the asynchronous clear input and \( Q \) is the register output.

3.2.6 FPGA Interconnects

Interconnects are the core routing element in the FPGA Architecture. They are bilateral channels used to traffic information between the FPGA compounding elements. Interconnects are placed between CLBs pairs or between a CLB and an I/O Block. There exists two types of interconnects that determine its location (or *domain*) in the FPGA Matrix.
Both Horizontal and Vertical Interconnects have a set of four inputs and outputs said to be *Lateral Switches*. These *Lateral Switches* are configurable tri-state buffers used to allow or disallow a CLB or I/O Block to either drive or be driven by an interconnect. *Lateral Switches* are labeled as SW0 and SW2 (for incoming tri-state buffers) while SW1 and SW3 are the outgoing tri-state buffers.

Interconnects are also tied to Switch Matrix in their both extremes. These extremes are tied to *Edge Switches*, that are two configurable tri-state buffers. There is not exception to this rule, and that is the only way to allow signals to come from a *horizontal domain* to a *vertical domain*, and vice-versa. Signals can only turn the corner by using Switch Matrices and *Edge Switches* to do that. Figure 3.19 shows a general Interconnect structure.

![Diagram of General Interconnect Design](image)

**Figure 3.19:** General Interconnect design, regardless its domain.

In the figure, it can be seen that the Interconnect provides an input and an output for each element tied in its *Lateral Switches*. The Synthesizer can chose to allow an Interconnect to provide content to several elements (either CLBs, I/O Blocks and Switch matrices), by enabling outgoing tri-state buffers, but only one of these elements can drive the interconnect. If the interconnect is being used in the user configuration, at least one of the incoming tri-state buffers will be enabled.

Interconnects are also arranged in small groups of four Interconnects, entitled Channel. In fact, all the routing elements in Alchemy FPGA architecture are presented in groups of four, so that routing becomes more feasible for large configurations. The Synthesizer can choose any of the compounding interconnects in a route. Each Interconnect in a channel is called *Track* and they are numbered from 0 to 3. Figure 3.20 shows the basic organization of a Channel.

![Diagram of Channel Organization](image)

**As it can be seen in the figure, Channels are made up of four interconnects as those seen in 3.19. A channel can have any number of interconnects enabled at the same time, and they are configured individually by the Configuration Controller.**

Apart from the buffers, Interconnects are mostly passive components that only *store* a logic value to be forwarded to the next element in the route. As soon as this logic value changes in its source, the interconnect will react to this change, storing immediately, the updated value.
3.2. ALCHEMY FPGA ARCHITECTURE

Figure 3.20: Channel basic organization.

![Channel basic organization](image)

Figure 3.21: Alchemy Subset Switch Matrix architecture.

![Alchemy Subset Switch Matrix architecture](image)

For simplicity, the interconnect can be understood as a simple wire.

### 3.2.7 Switch Matrix

Switch Matrices are important routing element used to tie interconnects together and to allow signals to change its domain. Its primary function is to provide a medium through which signals coming from one interconnect can reach another. It also allows signals coming from vertical (or horizontal) interconnects turn to horizontal (or vertical) interconnects. This is a fundamental feature for high-capacity FPGAs, that is widely used in market products and is expected by most of the Synthesizers to date.

Alchemy Switch Matrix architecture is based on Subset Model, in which for each track coming to the Switch matrix, there exist three possible output tracks to be chosen. Alchemy Switch Matrix is internally composed of four Switch Boxes that are tied to each track in an Interconnect channel. This topology states that each track \( i \) can be connected to any outgoing track also numbered \( i \). Figure 3.21 shows a basic Switch Matrix architecture.

As it can be seen in the figure, for a signal coming from input 1 in the North section of the Switch Matrix (highlighted with the arrow), there will always exist three possible routes to be taken. Note that all possible routes have the same track number of the input and that this organization is replicated for the other inputs.

Internally, each Switch Box is composed of a series of six switches tied to configuration bits. These buffers determine which outputs will receive a given input signal. Each track is tied to
Figure 3.22: Alchemy Switch Box architecture.

a single Switch Box. Figure 3.22 shows a high-level schematic of a Switch Box.
3.2.8 I/O Blocks

The FPGA internal logic is meant to interact with the outside world and other architecture elements such as the CPU and the Memory. In order to allow such a behavior, a set of frontier elements called I/O Blocks provide interface capabilities as well as single bit storage for incoming and outgoing FPGA signals.

I/O Blocks are buffers that can both receive signals from the BRIDGE bits (directly tied to them) or provide content to these bits (with signals coming from the FPGA). As it will be presented in the next section, BRIDGE Bits are BRIDGE entities used to provide communication with the CPU and the Memory. Each I/O Block is composed of a single bit latch, a 2-input multiplexer and a 3-output demultiplexer. The input multiplexer is responsible to determine which architecture element will drive the latch. One input is tied to a BRIDGE bit, while another is tied to an internal interconnect in the FPGA. Figure 3.23 shows the basic structure of an I/O Block.

![Figure 3.23: I/O Block general structure.](image)

The output demultiplexer determines which element will be driven by the I/O Block, or if it will drive no element. It is tied to the BRIDGE, to an FPGA interconnect or to the ground, usually following the opposite behavior of the input multiplexer. The Synthesizer is in charge to determine which will be the selection bit for the Multiplexers and Demultiplexers. This indirectly determines if the I/O Block is an input or output. Input I/O Blocks have their multiplexer configured to point to the BRIDGE bit they are tied to, while demultiplexer is configured to point to internal FPGA interconnect. Likewise, Output I/O Blocks have their multiplexer configured to point to the internal FPGA interconnect, while the demultiplexer is pointed to the BRIDGE.

Unlike internal CLB register, I/O Blocks are built upon a level sensitive D latch, with its clock input always set to 1. This means it is always in transparent state and changes in the input signal are immediately sensed in the output, without any considerable delay. The I/O Block is a passive component that mostly provides interface capabilities and do not provide any change in the signal they receive in their inputs.

Once configured, I/O Blocks will keep their behavior unchanged. Whenever an I/O Block is not used by the user configuration, it has its input multiplexer configured to point to the internal interconnect and the output demultiplexer is set to point to the ground. This ensures that the I/O Block does not impose any change in the BRIDGE or in the FPGA internal logic.

3.2.9 FPGA Configuration Bitstream

This section presents the organization of the configuration bitstream used to configure the FPGA internal configurable logic. Despite presented as a bitstream, the configuration bits are individually read from memory as 16-bit words that are in turn configured in the FPGA
elements by the Configuration Controller. The Configuration Bitstream is generated by the Synthesizer, based on the user module. Once stored in the Configuration Memory, the Bitstream is considered to be immutable.

Figure 3.24 shows the Configuration Bitstream, as it is stored in the Configuration Memory. Following the sequence from the lower addresses to the higher, the bits stored in the Configuration Memory are related to the following FPGA elements: First the Switch Matrices, second the Interconnects, third the CLBs, fourth the I/O Blocks and fifth the BRIDGE bits. The Configuration Controller configures the FPGA elements in the described order.

Switch Matrix bits are grouped in blocks of 32-bits in which each 8-bits half is used to configure a specific Switch Box in a Switch Matrix. Following the coordinate system presented in Figure 3.12, Switch Matrices are organized following their index in the FPGA Matrix, first by lines and then by columns.

At each group of 8 bits, 6 bits are used to configure the switches presented in Figure 3.22. The higher 2 bits are zeroed and not used during configuration. Figure 3.25 shows the Switch Matrix bit organization in the Configuration Bitstream.

There exist a total of 289 Switch Matrices in Alchemy FPGA, composed of four Switch Boxes each. These boxes in turn, require four words to be configured (following the memory alignment) which results in exactly 2312 bits to fully configure all Switch Matrices.

Interconnects are the second large configurable resource in the FPGA. In the configuration bitstream, there are reserved bits to configure each one of the four track of the interconnect independently. Each interconnect requires exactly a 16-bit word in the Configuration Memory. Each nibble of this word is used to configure a track in the interconnect. There exist a total of 544 interconnects in the current Alchemy FPGA implementation (both horizontal and vertical interconnects). Figure 3.26 shows the Interconnect configuration bitstream organization.

The function of each bit on each set of four bits is the following: the first and third bits are incoming signals from CLB or I/O Block output terminals. The second and fourth bits are regarded to signals that goes to CLB or I/O Block input terminals. A total of 8704 bits are
3.2. ALCHEMY FPGA ARCHITECTURE

SWITCH MATRIX CONFIGURATION BITSTREAM

<table>
<thead>
<tr>
<th>2 Bits</th>
<th>1 Bit</th>
<th>1 Bit</th>
<th>1 Bit</th>
<th>1 Bit</th>
<th>1 Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO USE</td>
<td>SW6</td>
<td>SW5</td>
<td>SW4</td>
<td>SW3</td>
<td>SW2</td>
</tr>
<tr>
<td>2 Bits</td>
<td>1 Bit</td>
<td>1 Bit</td>
<td>1 Bit</td>
<td>1 Bit</td>
<td>1 Bit</td>
</tr>
<tr>
<td>NO USE</td>
<td>SW6</td>
<td>SW5</td>
<td>SW4</td>
<td>SW3</td>
<td>SW2</td>
</tr>
<tr>
<td>2 Bits</td>
<td>1 Bit</td>
<td>1 Bit</td>
<td>1 Bit</td>
<td>1 Bit</td>
<td>1 Bit</td>
</tr>
<tr>
<td>NO USE</td>
<td>SW6</td>
<td>SW5</td>
<td>SW4</td>
<td>SW3</td>
<td>SW2</td>
</tr>
</tbody>
</table>

Figure 3.25: Switch Matrices configuration Bitstream.

INTERCONNECT CONFIGURATION BITSTREAM

\[
\begin{array}{c|cccc|c|cccc|c|cccc|c|cccc}
\text{M[i]} & \text{BIT} & \text{BIT} & \text{BIT} & \text{BIT} & \text{BIT} & \text{BIT} & \text{BIT} & \text{BIT} & \text{BIT} & \text{BIT} & \text{BIT} & \text{BIT} & \text{BIT} & \text{BIT} & \text{BIT} \\
\text{Track 00} & 4 \text{B} & 4 \text{B} & 4 \text{B} & 4 \text{B} & 4 \text{B} & 4 \text{B} & 4 \text{B} & 4 \text{B} & 4 \text{B} & 4 \text{B} & 4 \text{B} & 4 \text{B} & 4 \text{B} & 4 \text{B} & 4 \text{B} \\
\text{Track 01} & 1 \text{B} & 1 \text{B} & 1 \text{B} & 1 \text{B} & 1 \text{B} & 1 \text{B} & 1 \text{B} & 1 \text{B} & 1 \text{B} & 1 \text{B} & 1 \text{B} & 1 \text{B} & 1 \text{B} & 1 \text{B} & 1 \text{B} \\
\text{Track 02} & \text{IN} & \text{OUT} & \text{IN} & \text{OUT} & \text{IN} & \text{OUT} & \text{IN} & \text{OUT} & \text{IN} & \text{OUT} & \text{IN} & \text{OUT} & \text{IN} & \text{OUT} & \text{IN} & \text{OUT} \\
\text{Track 03} & \text{CLB0} & \text{CLB1} & \text{CLB0} & \text{CLB1} & \text{CLB0} & \text{CLB1} & \text{CLB0} & \text{CLB1} & \text{CLB0} & \text{CLB1} & \text{CLB0} & \text{CLB1} & \text{CLB0} & \text{CLB1} & \text{CLB0} & \text{CLB1} \\
\end{array}
\]

Figure 3.26: Interconnect Configuration Bitstream.

required to configure the Interconnect sea in Alchemy FPGA.

CLBs are the largest FPGA elements in terms of configuration bitstream length. Each CLB requires exactly 48 bits to be fully configured and each bit has a specific functionality in the element. Following the organization proposed in Figure 3.14, 256 CLBs have their internal logic configured based on the following Bitstream organization, presented in Figure 3.27.

OUTMUX2, OUTMUX1, KMUX, PMUX, RMUX, SMUX, DMUX, CMUX, BMUX and AMUX are selected by the order shown in the CLB connection diagram, presented in Figure 3.13. 00 selects its first input, 01 selects its second input and 10 or 11 selects its third input. The two 8-bit segment bytes for the LUT content are organized such as the LSB addresses the first bit of the LUT function and so forth. Every other multiplexer (M2 down to I1) is configured according to the following order: 0 selects its first input and 1 selects the second one.

CLBs bits are organized following the coordinate system presented in Figure 3.12, considering first lines and then columns. A total of 12288 bits (exactly 12 KB) are required to fully configure the CLBs.

Following the CLBs, there is the I/O Blocks Configuration bitstream. Each I/O block requires a total of two bits to be configured. These bits do not provide any specific configuration for the I/O Block multiplexer and demultiplexer, but rather are used to indirectly inform the Configuration Controller which will be their configuration. Figure 3.28 shows the Configuration bitstream for a sample I/O Block.

As it can be seen in the figure, the higher order bit is used to determine if the I/O Block is enabled or disabled (the demultiplexer is pointed to ground and the input multiplexer pointed to the FPGA Interconnect. The lower order bit informs if the I/O Block is an input or output.
The Configuration controller will provide the correct configuration bits accordingly. In order to configure all the I/O Blocks in the FPGA, it is required a total of 128 bits, organized in 8 16-bit words.

$$M[i] \times 64$$

As it will be presented in the next section, the BRIDGE requires a few configuration bits and therefore it has a thin slice in the configuration bitstream. It is required exactly 280 bits to configure the BDIGE, which will define the behavior of the BDIGE Bits and some internal features.
3.3 **BRIDGE**

Simplicity is the key element in Alchemy Architecture. Despite being a low-end architecture, the system management details can easily become overwhelming for newcomers to the architecture, due to its complexity. In order to support users and make the architecture less complex to be managed, a hardware controller was designed to handle the communication between the CPU and the FPGA. The BRIDGE can be divided into five main blocks, as depicted in Figure 3.29.

![Figure 3.29: BRIDGE internal structure.](image)

BRIDGE stands for *Bilateral Register-based Interface for Data and Guidance Exchange*. As its name suggests, it is a cross-domain interface controller, which implements the communication protocol between the CPU and the FPGA. The BRIDGE provides a reduced amount of storage resource called *BRIDGE Registers* that are the data exchange medium defined alongside a communication protocol. It is also works as a memory-proxy for the FPGA so that users do not have to include complex logic in their modules.

The BRIDGE can emit a single interrupt signal to the interrupt controller in order to inform that the FPGA has finished its execution. It is directly tied to the **CB Bus** and **FB Bus**. With **CB Bus** the BRIDGE communicates with the CPU. It reserves 2 lines for register addresses, 4 lines for control and 16 lines for data. With **FB Bus**, the BRIDGE allocates 64 lines for data (each one for a I/O Block in the FPGA) and one line for control. A subset of **FB Bus**, multiplexes 35 data lines (16 for memory addresses, 16 for data and 3 for control) for memory access functionalities and one extra line for return signal monitoring.

The BRIDGE is also in charge to control the FPGA clock signal. It can enable or disable the **clk** global interconnect route in the FPGA so that its execution flow can be controlled by CPU requests. There is no direct way to control the BRIDGE by means of programming/description.
languages and its functionality is defined during compilation/synthesis or indirectly during runtime. The BRIDGE internal architecture can widely vary depending on the architecture model (TLM or RTL) and further details are presented in Section 6.1 and Section 6.2.

3.3.1 BRIDGE Protocol

The BRIDGE communication protocol is a data/control signal exchange convention that allows interaction between the CPU and the FPGA. The protocol can be divided into three simple steps that enable content exchange (prologue), formalize the transaction (execution) and finalize (epilogue) it, without the need of hand-shake or pairing conventions. The protocol was designed to impose low latency between the communicating pairs and provides memory access functionalities to the FPGA.

The protocol was designed because a minimal coordination mechanism should exist between the CPU and the FPGA. The hardware is minimally designed after these conventions and the protocol itself minimizes the amount of work user must do in order to establish a cross-domain communication.

The BRIDGE protocol can be considered low-overhead because it does not require complex synchronization steps and does not require expensive control. This simplicity comes from the proximity of the CPU and the FPGA. The protocol can be easily expanded by further abstraction layers that can be implemented with libraries defied by users, so new functionalities and even improvements can be attached to it.

The BRIDGE Protocol defines that each member of a transaction is an entity. There can be at least (and at most) a pair of members in a transaction and they do not interact simultaneously. That means that only one of the entities can perform protocol operations at a time. Each member assumes a role in the transaction depending on its intentions. If a member wants to start a transaction, it is said that its role is the requester. The member that is the target of the requester (the one with which the requester wants to establish a communication) is called the receiver. Either the CPU, the FPGA or the Memory can be assigned to receiver role, but only the CPU and the FPGA can become requester.

The content to be exchanged between the transaction pair is called message. The message can be of at most 64-bits and is always exchanged through BRIDGE Registers. They are analogous to network packets, but no formal structure (like headers) are required. Its content is used as the inputs or outputs depending on the transaction goals. The BRIDGE will always store a message until a new request is received. Messages only have a semantic context in two cases. If a message holds a return interrupt bit, it is said to be a return message. Likewise, if a message is used to perform memory accesses (from the FPGA), it is called memory message. Messages only formally exist during the prologue and the epilogue.

In the prologue stage, the requester provides the content to be used as a message for the interaction. If the requester is the CPU, it should write BRIDGE Registers accordingly. The prologue does not have an exact number of machine-cycles to be performed in, but it usually takes from 1 up to 4 cycles. If the FPGA starts the transaction, the message is directly derived from its output I/O Blocks which requires no machine-cycles to be performed.

In the execution stage, there exists three cases based on the requester and the interaction type. The first one is when the CPU requests a transaction to the FPGA (in Partner Mode). In this case, considering that the prologue has been successfully completed, the CPU emits an FPGA Cross-Procedure call request (with instructions par or seq) and continues its execution, depending on the request type. The BRIDGE is only in charge to forward the CPU inputs to the FPGA and allow its clock signal to change. The execution stage can happen for any amount of time.
3.3. BRIDGE

If the FPGA starts an execution stage (in Slave Mode), it must emit a Return interrupt to the CPU and the Return ISR must be prepared to handle the FPGA request (as presented in Section 3.2.2). After the CPU is done processing the request, it will emit another FPGA call request in order to put the system in its previous state, allowing the FPGA to continue its previous execution. Like in the first case, there is no time limit for this kind of execution.

When the execution is meant to interact with memory, the requester is always the FPGA. The BRIDGE watches for I/O Blocks with memory-based I/O and uses them as a message to the memory. The memory is considered the receiver and it will expect the BRIDGE to first send a memory read/write request and will wait for the acknowledge signal from the memory. After that, the BRIDGE puts the requested address in the Global Address Bus and wait for the memory to confirm the address reception. Once the address is successfully read by the memory, if the access consists of a read, the requested address content will be available in the Global Data Bus so the BRIDGE can copy its content and makes it available to the FPGA I/O Blocks. It also rises data_ready signal, so that the FPGA can store the incoming information. If the access is a write, the BRIDGE will wait for the memory to acknowledge the address reception and, then will inform the FPGA data_ready input that the memory is ready to receive the content to be written in the requested address. The BRIDGE will wait for two clock-cycles before copying the content of the FPGA I/O Blocks assigned to memory_data_out to the Global Data Bus. Once the memory writes the requested information it will acknowledge back the operation result to the BRIDGE, so it can report the status to the FPGA (with data_ready).

Once a transaction finishes its execution it enters in epilogue stage. There can be two types of epilogue depending on the transaction requester. If the CPU started the transaction, the epilogue begins when the BRIDGE identifies a rise in the return signal in the FPGA. It will forward an interrupt signal which will prompt the CPU to the Return ISR so that it can read the content of the BRIDGE Registers.

If the transaction was started by the FPGA, the epilogue consists of the emission of a FPGA call from the CPU, in the return ISR. In that case, the output message is readily available in the I/O Blocks, so the epilogue is shorter. After the epilogue is finished, no more transactions exist and the BRIDGE is ready to begin a new one. Figure 3.30 shows a simplified diagram of the BRIDGE protocol.
Figure 3.30: BRIDGE Protocol timing diagram. 1: Full transaction with CPU as requester and FPGA as receiver. 2: Fill transaction with FPGA as requester and CPU as receiver. 3: Full memory read transaction from the FPGA. 4: Full memory write transaction performed by the FPGA.
3.3. BRIDGE

3.3.2 BRIDGE Schedule Register

The Schedule Register is a 16-bit register tied to the CB Bus, used to store the number of clock cycles the FPGA Cross-Procedure should execute, before emitting a Return interrupt. The CPU can only write content to the Schedule Registers, but never read it.

The Schedule Register is decremented whenever its value is non-zero. The BRIDGE Control Unit is in charge to update its value once at each clock-cycle. Note that Value stored in the Schedule Register is always considered to be unsigned, so negative values passed to the register will be converted to their unsigned counterparts.

3.3.3 BRIDGE Control and Monitoring Unit

The BRIDGE Control and Monitoring Unit is a collection of finite state machines that are in charge to emit FPGA call requests and to watch FPGA I/O signals so that Return interrupts can be issued and memory accesses can be performed from the FPGA. It can be divided in two parts: the BRIDGE Control Unit and the Monitoring Unit. The Control Unit is directly tied to the CPU and will identify requests from the CPU and control the FPGA clock signal accordingly. As a Monitoring Unit, it uses the BRIDGE Selection Plane to watch I/O Blocks and perform certain tasks according to FPGA requests.

The BRIDGE Control Unit receives FPGA call requests and enable or disable the FPGA clock signal. It is not in charge to control FPGA reads and writes once this operation is directly performed in the CPU or the FPGA. The BRIDGE Control Unit state machine can be seen in Figure 3.31.

The BRIDGE Control Unit usually stays in an IDLE state, in which no work is done, and it constantly checks the ins control lines to determine if the CPU requested any FPGA call or
The BRIDGE scheduling operation. If a **dst** (presented in Section 4.4) instruction is detected, with non-null register value ($N > 0$), the BRIDGE will be prone to the **SCHEDULE** state in which the content of the Schedule Register is updated with the content provided by the CPU. At this point, null schedules are ignored by the BRIDGE. This instruction also informs the BRIDGE that the Schedule register must be updated when running FPGA calls (with **DEC** register). After updating the register, the BRIDGE is brought back to **IDLE** state.

If a **par** (presented in Section 4.4) instruction is detected (10) the Control Unit is prompted to the **PARALLEL** state in which the FPGA Clock signal is enabled to change. Similarly, if a sequential request is detected (01) the controller transits to **SEQUENTIAL** state, also enabling the FPGA clock signal. In both cases the controller enters in a **WAIT** state, if **DEC** is equal to zero, or will be prompted to **DECREMENT** state, when a schedule has already been registered. In both cases, the FPGA Cross-Procedure call is executed.

There exists two ways to leave **WAIT** state. The first one is when the FPGA module naturally emits a Return interrupt. In this case, the BRIDGE rises the **return** line in CB Bus, informing the CPU that the FPGA finished its execution. The Control Unit transits back to the **IDLE** state, becoming available for new requests.

When in **WAIT** state, the Controller continues to watch the **ins** lines, waiting for a **dst** instruction (with null register value), which forces the BRIDGE to block the FPGA execution and issue a Return Interrupt. In this case, the Control Unit goes first to **ABORT** stage and then leaves it, rising the **return** line.

Once in **DECREMENT** state, the BRIDGE will continuously decrement the Schedule Register until its value is not null. In this case the FPGA clock signal changes freely. If zero is reached, the Control unit is prompted to **ABORT** State, which will emit a Return interrupt to the CPU. When in **DECREMENT** state, the BRIDGE also watches the **ins** lines in order to determine if a **dst** instruction is emitted with null schedule. If so, the Control Unit is also prompted to **ABORT** state, and the Schedule Register is cleaned afterwards.

Note that whenever in **WAIT** or **DECREMENT** states, no new FPGA call requests are accepted from the CPU. It is only allowed to abort an FPGA call, if required. **PARALLEL** and **SEQUENTIAL** states do not exists in the actual implementations of Alchemy, but are depicted in the figure for better understanding.

Aside from the Control Unit, there exist the Monitoring Unit. It watches I/O Blocks to determine if a return interrupt has been issued by the FPGA and to handle FPGA memory accesses requests. It was designed to simplify the device management tasks for the user so that complex tasks such as memory reads and writes can be performed with minimal effort.

When waiting for **return** signal, the Monitoring Unit only checks a single I/O Block. It is tied to the Selection Plane and whenever the content of the **return** line changes to 1, an acknowledge signal is sent to the BRIDGE Control Unit informing the occurrence of a Return interrupt (during **EMIT** state). After that, the Monitoring unit enter in an idle state (**RESUME** state), waiting for the return signal to change back to 0. It is important to realize that two natural interrupts can only be emitted if the watched I/O Block changes from 0 to 1, and then changes back to 0. Otherwise, the Monitoring Unit will ignore interrupts form the FPGA. This is an important convention that shall be considered when designing Verilog modules. Figure 3.32 shows the Monitoring Unit state machine.

---

6those directly emitted by the FPGA
Figure 3.32: Monitoring Unit state machine.

The figure also shows states used by the Monitoring Unit to perform memory accesses as a proxy for the FPGA. The state machine implements the conventions defined by the BRIDGE Protocol. Detailed information about the memory access sequence is presented in Section 3.3.2.

### 3.3.4 BRIDGE Selection Plane

The BRIDGE Selection Plane is a set of configurable interconnects that provide access to specific FPGA I/O Blocks to the BRIDGE. It allows user defined signals like `return` or `memory_data_ready` to be directly forwarded to the BRIDGE without any intermediate stage. Physically, the Selection Plane consists of a series of 36 parallel lines which 17 of them are output lines and the remaining are input lines. I/O Blocks can have either their outputs or inputs tied to them, depending on its type.

Each line is tied to the 64 I/O Blocks. That means that there exist 64 tri-state buffers, one for each I/O Block, that are in turn connected to a configurable bit. Only one tri-state buffer shall be enabled per line. The Compiler/Synthesizer uses the Verilog Module I/O pins to determine which I/O Block will drive each line. It can only be configured once by the Configuration Controller, at the system startup time.

Lines are categorized according to their types (outputs or inputs) and also organized in numbered sectors. All output lines range from O0 to O15, while input lines range from I0 to I35. The Selection Plane directly drives the Control and Monitoring Unit and the Memory Interface Unit.
3.3.5 BRIDGE Register Bank

The BRIDGE Register Bank is a set of four 16-bit registers used to store inputs and outputs from either the CPU and the FPGA. Each register in the Bank is called a BRIDGE Register and they are clusters of single bit structures, called BRIDGE Bits, with two inputs and two outputs. Each one of the four BRIDGE Registers are identified by an unique address and a name. B0 is the East Register (address 00), B1 is the North Register (address 01), B2 is the West Register (address 10) and B3 is the South Register (address 11).

The CPU considers each BRIDGE Register as a single 16-bit storage element. It cannot read nor write individual bits, but always address the whole register content. The FPGA treats each bit individually and the Verilog modules have fine control over them (even though not explicitly).

BRIDGE Registers are essentially the medium through which signals can be exchanged from the CPU to the FPGA and vice-versa. One could argue that direct interconnects could reduce delay and provide a direct communication between the modules. However, there are a few functionalities that are required for the BRIDGE to provide, that could not be implemented without BRIDGE Registers. For example, the BRIDGE can delay FPGA call requests (based upon certain conditions) and the inputs provided by the CPU should be stored somewhere to be forwarded in a more appropriate time. The BRIDGE Registers are also useful to store addresses and data from/to memory when requested by the FPGA. Note that the FPGA inputs need a constant driver and its outputs can be stored for later use in the CPU.

A BRIDGE bit is designed to support requests from both the CPU and FPGA, while keeping consistency on accesses of bit values. A BRIDGE bit can only be read or written by one work-modules at a time and this is decided during build time. Physically, a BRIDGE Bit is an edge-sensitive latch tied to a two input multiplexer and a two output demultiplexer. These multiplexers and demultiplexers selectively define if the CPU or the FPGA is the source of its data and which will receive it during runtime. The latch also has its clock input tied to a multiplexer, which will determine when the bit value will change. Figure 3.33 shows a BRIDGE bit. In the figure, BCB1, BCB2 and BCB3, are configuration bits tied to the multiplexers/demultiplexer to enable the correct functionality in the Bit.

The input multiplexers are used to determine which module will write content into the
3.3. BRIDGE

BRIDGE. It is tied to the CB Bus data lines and to the FB Bus I/O lines. Once the multiplexer is a mutually exclusive logic entity, only one of the busses is meant to provide data to the BRIDGE bit data input. At the CB Bus side, one of the multiplexer input is tied to a four output demultiplexer. This is called source selection demux and its select signal comes from the address lines in the CB Bus. This demux is used to determine which one of the four registers will be written, based on the CPU request (and its two bit address). The BRIDGE Bit will only receive content from the CPU if it belongs to the selected register in the write operation. At the FB Bus side, the multiplexer remaining input is tied to the I/O Block that is in the matching side in the FPGA. That means that, for example, a BRIDGE Bit i in East Register (B0) will be tied to the I/O Block i in the East I/O Vector in the FPGA.

The output demultiplexers are two outputs demuxes in charge to select which work-module will receive the content of the BRIDGE Bit. They have its outputs tied to the CB bus and FB bus. In the CPU side, each bit is tied to a four input multiplexer which is directly connected to the data lines in the CB bus. There are 16 multiplexers which have their select input tied to the address line in CB bus, so that the CPU can select which BRIDGE Register will be read. In the FB bus side, an output of the demultiplexer is tied to the corresponding I/O Block in the FPGA I/O.

BRIDGE bit functionality is defined after the configuration of the I/O Block it is tied to. If an I/O Block IOB_i is configured as an input, that means that the corresponding BRIDGE Bit BB_i will have its content provided by the CPU. The input multiplexer is set to point to the CB Bus and the output demultiplexer is set to forward content to the FPGA. Likewise, if an I/O Block IOB_j is configured as an output, its corresponding BRIDGE Bit BB_j will be driven by the FPGA while CPU can only read its content.

3.3.6 BRIDGE Memory Interface Unit

The Memory Interface Unit is a small functional block that forwards memory access requests to the Memory Management Unit. It takes inputs and outputs from Control Unit, Monitoring Unit, and the Selection Plane and simply copies them to the inputs of the MMU. Likewise, it is tied to the Memory Controller outputs so that they can be directly forwarded to the BRIDGE.
3.4 Memory Organization

Memory is one of the pillars of any computer system. Memory resources are used to store information used by the system to perform its tasks. Alchemy is no exception for this rule and provides three memory resources used to store Program Instructions, Configuration bitstreams and runtime data.

There exist two read-only memory resources (one tied to the CPU and other to the FPGA) and one read-and-write memory block, used by the whole system (and I/O devices) to access run-time data. This memory organization intends to implement a more efficient memory subsystem in which memory accesses overhead is reduced by reducing the traffic in the system shared busses.

Each Memory block is organized in cluster-like sectors that hold 256 memory cells of 8-bits each. These clusters are called Banks. Each memory block has 256 Banks that are enabled in a mutually exclusive way. If Bank $N$ is enabled, any other Bank is disabled. Whenever a memory access requests interaction with a Bank other than $N$, it will be disabled and the requested Bank enabled.

This memory architecture is useful to reduce static power consumption [15]. Alchemy memory can also be optimized by Compilation/Synthesis techniques that can schedule memory access to avoid enabling multiple Banks, improving the memory system as a whole. Figure 3.34 shows Alchemy Memory Architecture.

![Memory Block and its Bank architecture.](image)

The address decoding takes three stages where Banks are first selected in terms of rows and columns. The eight most significant bits in an address are used to select the desired Bank, while the remaining eight least significant address bits are used to select a Bank cell. Alchemy memory has two access ports which allows two reads or a read and a write to be performed simultaneously. This is useful to improve the memory performance, once it is required two access to handle a full 16-bit word.

The CPU can have direct accesses to two different memory resources. A static memory block called Program Memory stores immutable data that primarily represents constants and program instructions. It can only be read during run-time. Another memory resource is
the Shared Memory. It is used to store program data, and can be either read or written during run-time. Like-wise, the FPGA have direct accesses to a static memory resource called Configuration Memory, that stores the configuration bitstream used to define the state of each FPGA configurable element. It also has accesses to the Shared Memory by means of the BRIDGE. The FPGA uses this memory resource to store and retrieve run-time data.

Alchemy CPU deals with two distinct address spaces, being one for the Program Memory and another for the Shared Memory. This memory organization is usual for Harvard architectures [30], in which data and program memories are physically divided into two blocks. Harvard architectures usually plot better performance on memory accesses, when compared to Von Neumann, since they provide two separate memory busses. Alchemy implements a Harvard memory architecture to improve its memory throughput. The CPU can access any address between $0$ to $2^{16} - 33$ in the Program Memory and any address between $0$ to $2^{16} - 1$ in the Shared Memory. The CPU reserves 32 addresses in the Program Memory for interrupt management that are only accessed by the interrupt controller.

Like defined for the CPU, the FPGA also distinguishes two distinct address spaces of 64 Kb each, being one of them for the Configuration Memory and another for the Shared Memory. The BRIDGE has no accesses right to the Configuration Memory, which is reserved to the Configuration Controller. Similarly, the Configuration Controller cannot perform any accesses in the Shared Memory addressing space. Either the Configuration Controller and the BRIDGE can accesses any address between $0$ to $2^{16} - 1$ in the Configuration Memory and in the Shared Memory respectively.

Each address makes direct reference to an 8-bit word. Accesses are usually performed in pairs, once Alchemy’s primary data and address bus length is 16-bits. For that reason, the address space is regarded as consisting of $2^{15} - 1$ words, being each one of them aligned.

The alignment states that memory access can only be performed in even addresses once the memory block will automatically provide the following odd address to compose a single 16-bit word. The CPU Control Unit and the BRIDGE Control Unit expects memory access to be aligned. In fact, when running the CPU in byte mode or performing Memory accesses from the FPGA with only 8-bit data busses, unaligned memory accesses can be performed coherently.

In the CPU context, instructions must be fetched in aligned addresses. The Program Counter always holds even addresses to ensure alignment and odd addresses may cause execution hazards to the CPU. Once branch instructions can only run in word mode, the Program Memory is always read in an aligned addresses. The CPU will not notify the use of unaligned accesses during run-time, once they are allowed in byte-mode. Henceforth, it is the user responsibility to ensure consistency in address alignment.

In the FPGA context, parts of the configuration bit-stream is fetched in aligned addresses. The configuration stream is organized so that configuration bits are padded in one byte. Once memory accesses are automatically performed by the Configuration Controller, no unaligned accesses are prone to happen during the FPGA configuration.

Memory accesses are managed by the Memory Management Unit (MMU). The Memory Manager Unit deals with both Shared and Program Memory access requests. The MMU is a special controller that provides a conversion table and an address translation functionalities to transform a plain virtual address into a real address that selects memory banks and inner cells in the Shared and Program memories. It is directly tied to the Global Address Bus and also handles memory access timing and permission. It was designed to match the memory architecture and to forward signals from both sides, without directly needing of external intervention.

The MMU will send MEMORY_READ_REQUEST signals whenever reads are requested by the CPU or the BRIDGE. Likewise, MEMORY_WRITE_REQUEST signals are emitted whenever writes are requested. The MMU deals with memory CHIP_ENABLE signal to properly enable the memory
resource and also watches MEMORY_DATA_READY signals as the memory operation is successfully completed. If the CPU or the BRIDGE requests a forbidden or out-of-range address, the MMU will emit a memory error without forwarding the address request to the memory, ensuring access consistency during run time. Figure 3.35 shows the memory blocks alongside the MMU.

As it can be seen in the figure, the MMU receives a virtual address as one of its input. This address is a plain 16-bit address ranging from 0 to 65535. The Address Translator first breaks this address into two halves and sends the real address (memory row, column and bank address) to the memory block. It will wait for the operation to be completed afterwards.

Once a read is completed, the MMU will inform the operation status to the Control Unit or the BRIDGE Monitoring Unit, and resume its execution. The read data is forwarded to the Global Data Bus, by the memory block itself, and then read by the requester. If the operation consists of a write, the control unit waits for the memory acknowledge signal and informs the requester the operation status. The data to be written in the memory is put in the Global Data Bus by the requester.

### 3.5 Minor Architecture Elements

#### 3.5.1 Interrupt Controller

The Interrupt Controller is in charge to detect incoming interrupt signals and perform the interrupt start-up sequence, by fetching the correct ISR to the CPU. Its internal structure is largely based on an address decoder and simple control logic. Figure 3.36 shows the Interrupt Controller internal structure.

As it can be seen in the figure, there exist a set of 16 inputs, that represent the interrupt sources. Each device that wants to provide external interrupt signals must be attached to one of these lines. In fact, three of them are not available to the outside world once they are tied to internal non-maskable interrupt sources. The interrupt controller was designed to accept a single interrupt at any time. That means that in case of two interrupts happening at the same time, only one of them will be served.
The Interrupt controller passively provides the ISR address based solely on its inputs. The address is directly forwarded to the Memory Management unit (overriding other address sources), through output pins o0 to o15, and a control signal is sent to the CPU, so that it can start the ISR call routine at the next Fetch stage. At this point, the CPU acknowledges the Interrupt Controller back, which will suppress the interrupt signal, until a new one is detected at its inputs. Due to its simplicity, the Interrupt Controller is usually implemented within the Control Unit.

### 3.5.2 Configuration Controller

The Configuration Controller is small Control Unit used to configure the FPGA components. It was designed to be directly tied to the Configuration Memory (through a specific data and address bus) so that memory reads are performed efficiently. The Configuration Controller is composed of 10 states FSM, an address decoder and an address counter. The State machine basic structure can be seen in Figure 3.37:

![Configuration Controller finite state machine](image)

The States are:

- **OFF**: Initial state.
- **S1**: State 1.
- **S2**: State 2.
- **S3**: State 3.
- **S4**: State 4.
- **S5**: State 5.
- **S6**: State 6.
- **S7**: State 7.
- **S8**: State 8.
- **S9**: State 9.
- **S10**: State 10.

There is an exact number of clock cycle the FSM will stay in each state. Implicitly, each
state represents a class of FPGA components to be configured. Once the creation of a state for each FPGA component would be impractical, the FSM is controlled by a counter. The counter is increased after the configuration of each component (also called an iteration), and the FSM transits to the next state if a specific amount of configurations is performed.

The Configuration Controller is directly tied to a set of enable signals that are connected to each configurable point in the FPGA. It selects which point will be configured by means of a decoder designed after a standard memory decoder. Each FPGA component is chosen with an 11-bit address that is accessed sequentially as the FSM progresses through the configuration process. Note that for each component, the amount of bits to be written in the configurable points can widely vary. This greatly simplifies the configuration controller logic by improving its regularity. Furthermore, the configuration bits organization is defined in the FPGA component, so that the Configuration Controller has nothing to do with the specific configuration logic.

Following the FSM structure, the Configuration Controller first configures the CLBs. The FPGA component address ranges from 0x0 (0) to 0xFF (255) at this stage, and each CLB is configured by performing three Configuration Memory access. After 256 CLB configurations, the Configuration Controller transits to state S2 where Switch Matrices are handled. At each iteration, the four Switch Boxes in a Switch Matrix are configured. This requires two memory accesses, performed from addresses 0x100 (256) to 0x221 (545). The FSM stays in state S2 for 289 iterations until changing to state S3. At this point, only Vertical Interconnects are configured. There are 272 Interconnects accessible from address 0x222 (546) to 0x331 (817). Each one of them is configured with a single memory access, handling the four tracks simultaneously. Likewise, at state S4, Vertical Interconnects are configured with the same amount of bits, requiring the same number of memory accesses, performed from address 0x332 (818) to 0x441 (1089). The four Interconnect tracks are also configured simultaneously.

At states S5.1, S5.2, S5.3 and S5.4 the I/O Blocks are configured based on their position in the FPGA. Once there are 16 I/O Blocks per side, each side is fully configured in one of the S5.X state. The Configuration Controller accesses the FPGA component addresses 0x442 (1090), 0x443 (1091), 0x444 (1092) and 0x445 (109) to configure the I/O Blocks. The BRIDGE bits are configured in the final state S6, where the Selection Plane, and other small features are configured. There are required 98 iterations to fully configure the BRIDGE.

After the configuration is completed, the Configuration Controller issues an Acknowledge signal to the CPU Control Unit which will become operational to start the execution of user programs. The configuration Controller is shut off at this point and will only become operation again if a reset interrupt is detected. The Configuration Controller Configures the FPGA with approximately 12,200 simulation-cycles.

### 3.5.3 System Busses

There are defined two types of system-wide busses, organized according to their proximity to the BRIDGE. These busses are the Local Busses and the Global Busses. Local Busses are used to directly connect the CPU to the BRIDGE and the FPGA to the BRIDGE. Similarly, Global busses are used to connect the CPU, the FPGA and the BRIDGE to the global memory resources and minor control elements such as the Configuration Controller and the Interrupt Controller. This section presents a detailed description of the system busses, according to the stated groups.
Local Busses

Local Busses are small and fast wires used to connect the CPU to the BRIDGE and the FPGA to the BRIDGE. The bus is formed by 201 lines that are actually a set of two non-synchronized local busses, with a different number of lines, that serves to multiple purposes in the BRIDGE Protocol. The local busses are called CB Bus as an acronym for CPU-BRIDGE Bus and FB Bus as an acronym for FPGA-BRIDGE Bus.

CB Bus is mainly used by the CPU to control the BRIDGE during a Cross-Procedure call. The BRIDGE also uses CB Bus to its report status back to the CPU. The bus also provides a set of data and address lines to allow content exchange between the CPU and the BRIDGE Registers.

CB Bus is formed up by 44 lines. These lines are categorized as control, data and addresses lines. 32 of these lines are used to exchange data between the CPU and the BRIDGE. 16 of these come from the CPU to the BRIDGE (more specifically from the ALU output), and the remaining 16 lines come from the BRIDGE to the Register File data input.

There are 4 address lines used to inform which BRIDGE Register is meant to be written or read. There are 8 control lines used to inform the BRIDGE operation type and the register to be used in a Cross-Procedure call. The CB Bus does not define the concept of ownership once there are well defined lines owners.

The second bus is the FB Bus. This bus is a more diffuse Local Bus used to connect the FPGA I/O Blocks to the BRIDGE, in order to implement the part of the communication protocol defined in Section 3.3.1.

FB Bus is formed by 165 lines. These lines are categorized as Control lines, I/O lines and Memory lines. They are mostly connected to the FPGA I/O Blocks and to the BRIDGE Memory Interface Unit. The BRIDGE Selection Plane is another important element used to organize the FB Bus depending on the user configuration.

Control lines are the group that holds the FPGA Clock line (from the BRIDGE), and the Memory control lines. The Memory control lines are selected during configuration, with the BRIDGE Selection Plane. Likewise, Memory lines are 32 lines used to send and receive full words from memory. Like the Memory control lines, Memory lines are defined during configuration. In both cases, these lines are directly tied to the FPGA I/O Blocks.

The largest part of FB Bus is composed of I/O lines. These are 128 wires (64 for input and 64 for output) that directly connects I/O Blocks to BRIDGE Registers. There is an 1:1 relationship between each I/O Block and an Input/Output line pair in the bus. Like in CB Bus, there is not a coordination mechanism for this bus. The bus is driven by both the BRIDGE and the FPGA, according to the conventions to presented in Section 3.3.1.

Global Bus

The Global Bus is a more general bus used to bring together all architecture elements in a single functional unit. This includes Configuration lines and Memory accesses busses, and other minor interconnects.

Global Bus lines are categorized into control, data, address and configuration lines. Control lines are those used in transactions with the CPU, Memory (Shared, Program and Configuration), Memory Management Unit, FPGA, BRIDGE, Configuration Controller and Interrupt Controller. They are mainly in charge to enable the architecture blocks, to send control signals (such as operation types and target registers), change control state (in control units such those in the CPU, BRIDGE and Configuration Controller) among several other functionalities.

Data lines are solely used to exchange words, bytes and bits among the architecture elements. Non-conventional data lengths are also exchanged in this bus, specially for branch
targets (10-bit offsets). Address lines are in companion with data bus and usually are used to inform memory lines, BRIDGE registers and CPU registers.

Most of the lines in Global Bus are part of the Configuration Controller. They are tied to individual element in the FPGA and the BRIDGE and are only used during the system configuration (at startup). The lines are organized in groups depending on the elements to be configured. The data to be sent to this elements have non-conventional lengths, used to simplify and optimize the configuration process.
Chapter 4

Alchemy Applications

Applications are the logic part of each computer architectures. Applications are collections of concepts used to guide the hardware towards a intended functionality, usually implemented in software. When running an application, the computer architecture is being used to perform a specific task. Applications are designed using high- or low-level programming languages and generally employ abstract logic concepts on its implementation.

Alchemy requires programmers to work with two classes of languages (programming and description) and also introduces concepts such as cross-procedures and flows. This Chapter describes the abstractions related to the application development with Alchemy, including logic concepts and development languages.

4.1 Alchemy Application Organization

An Alchemy application can be divided into a Program and a Module. A Program consists of a set of machine-readable instructions that is meant to run in the CPU. The Program can perform any kind of arithmetic, logic and control flow operations strictly defined in the processor ISA. The Program is stored in the Program Memory. A Module is a set of structural or behavioral statements that explicitly describes digital circuits and are embedded to the FPGA. They can virtually implement any logic function, and should be stored in the Configuration Memory.

From the developer perspective, applications are collections of source code files. These source files are succinct textual description of intended functionalities, organized into software and hardware parts. Softwares consists of programs designed with C language. Similarly, hardwares consists of modules designed with Verilog. These languages were carefully chosen to match the Compiler/Synthesizer requirements, presented in Chapter 5.

A C program is composed of one or more source files (.c files). Each source file can implement any number of functions, define complex types and include headers files (.h files) that in turn may define their functions and types to be implemented by other .c files. C programs may also include pre-compiled libraries, user defined constants and macros. The compiler is in charge to gather all the information defined in the collection of .c and .h source and build an executable program as output. Multiple .c files are composed in a compilation technique called Single Compilation Unit, in which multiple .c files are linked together to build a single executable unit.

The compiler works with the concept of entry-point, function that must be included in a single .c source file, to be the function through which the program starts its execution. The
entry-point function must be included in the main .c source file and should mandatorily be called main. The function do not include any input parameter by default.

A Verilog module is composed of one or more source files (.v files). Modules can be considered a basic building block of Verilog code. It defines interface ports (for I/O operations), registers, wires, and combinational/sequential digital logic blocks. Modules are hierarchically composed of instances of other modules that may or may not be defined by the user. The synthesizer gathers the multiple modules information and turns them into a single configuration bitstream as output.

It is required a top-level module described in a single .v file. This module gathers all lower-level modules into a single entity that must mandatorily be called main. The main module is used as starting point by the synthesizer. It can have any number of instances and at most 64 inputs/outputs. Figure describes the Alchemy application organization as a collection of source files.

![Alchemy Application Organization](image)

Figure 4.1: Alchemy Application Organization

The figure depicts Alchemy Application organization in two different levels. Users can also develop applications in a lower level using assembly languages. Alchemy programs can be described with an assembly language (written in a .s file) that encodes machine instructions with mnemonics. These mnemonics may use parameters to directly deal with registers and memory. Alchemy modules can be developed with another assembly language that describes the FPGA internal organization. It encodes the device placement (with .place file), its internal routes (with .route file) and the internal CLB logic (with .blif files).

User can choose any of these abstraction levels to design their applications. The following sections describe Alchemy high-level and low-level development languages and their non-standard structures.

### 4.1.1 Alchemy Cross-Procedures

A Cross-Procedure is a high-level software abstraction concept that characterizes FPGA execution requests from a C program. Cross-Procedures are function-like abstractions that accept input parameters and may indirectly provide returns.

Cross-Procedures are represented with special C syntactic elements developed for Alchemy, that imply in requesting the BRIDGE to send inputs to the FPGA, requests its execution and writes back results. Cross-Procedures try to abstract the overhead of the call initialization, development and finalization by hiding from users the machine-specific tasks involved in an
actual call. In a lower level, Cross-Procedures leads to the concept of Flows that are presented in the following sections.

4.2 Alchemy Programming Model

Determine a suitable computation model for reconfigurable architectures is useful to support users to design better applications. One could think of many-core architectures programming model, in which kernels are delivered to the many computation units in the device in the form of instruction packets, that provides local and global storage capabilities alongside some hierarchic organization. However, Alchemy could not benefit from this computation model for three reasons. First, the FPGA do not formally execute instructions; second, computation models such those provided by OpenCL [34] or CUDA [36] require complex setup procedures; and third, these computation models are designed for heavily tiled and pipelined SIMD systems, which is not the case of the proposed FPGA architecture.

In a practical multithreaded environment, the elementary way to deliver concurrent work units to the CPU is by creating threads that may run concurrently (and not necessarily in parallel) with other threads. This programming model loosely resembles real-life parallelism, but works as a friendly allegory for developers to explore the underlying concurrency characteristics of their operating systems and their hardware.

Taking threaded environments as reference, it was adapted a programming model approach to aid users to abstract Alchemy characteristics and provide simpler and more natural programming abstraction. This can be mostly used as a didactic approach to understand the architecture model. That is what was intended by defining the concept of Cross-Procedures and Flows.

4.2.1 Alchemy Flows

Flows are execution units as seen from the FPGA standpoint. In essence, Flows are threads: they do execute in parallel with other processing units and can be created, destroyed, awaited and so forth. A Flow is an elementary execution seam branched from the CPU’s execution flow. They cannot be scheduled nor re-spawned and are designed to be easily manageable. It is defined a high-level Flow management mechanisms to aid developers to ensure runtime correctness when using the CPU and the FPGA in their applications.

The main difference between threads and Flows lies in their physical representation on behalf of the hardware. Threads comes with contextual information, run over a common addressing space and execute instructions. Flows are simpler in terms of protocolar conventions and one could even say that they are more primitives than threads. Flows do not execute formal instructions once they are abstractions of an FPGA call\(^1\).

Regardless the task a Flow is meant to perform, it is not described as a formal (and sequential) set of instructions to be ran in a CPU. There are no clearly defined registers nor machine cycle and no conventions at all. This looseness in terms of protocol is the price paid to attain design flexibility. Designers could not rely on much formalism in the FPGA architecture, otherwise it could become not as generic as it was meant to be, and the hardware could dangerously become general purpose rather than application specific.

Another fundamental difference is the creation method. Flows are unitary. There are no two Flows running at the same time. The concurrency in a Flowed environment falls between the CPU and the FPGA, rather than two CPUs or multiple execution flows in the same CPU.

\(^1\)FPGA calls are abstractions of the event of allowing the FPGA global clock signal to change.
Once there is only one FPGA in Alchemy architecture, Alchemy limits the Flow amount to one. Threads in other hand, can be spawned in whichever number the designer wants. There can be as many threads as the number of cores (or processors) are there in a computer system, or there can be even theoretically an infinite number of threads, that are managed by an Operating System to be scheduled over time so that all threads have a chance to run.

Threads are highly flexible in terms of what they are meant to execute: taking POSIX Threads [38] as an example, a thread can be assigned to different procedures by providing a function pointer as a parameter at the moment it is created. This allows threads to potentially execute any piece of code in the program addressing space, giving high flexibility to designer to divide work between several thread. Flows however, are not driven by execution entry-points (like function pointers). The FPGA configuration are digital circuits designed by the user and it is what the Flow understands as workload. The Flow ultimately will perform the task described by the high-level HDL module. Obviously, the module can determine different tasks to be performed, given an input provided by the CPU, but the Flow is not aware of that. It will always run based on the FPGA configuration and there are no such things as entry-points or procedures to be called.

Despite the stated differences between Flows and threads, it is possible to highlight a few similarities. First of all, both threads and Flows can make use of shared variables, or common memory locations to all processing units running in parallel. In fact, the FPGA and the CPU can read and write the same memory locations (Shared Memory resource), like any two threads can. Alchemy Flows also do resemble threads when it comes to parallelism. Hardware threads can run at the same time in different CPUs, and so can an Alchemy CPU thread and an FPGA Flow.

Flows can obviously be compared with multiple running peripherals in a computer systems in respect to the CPU. Each peripheral has its own execution pace and internal architecture so the CPU must adapt its execution flow to establish a relationship with them. That relationship is what a Flows abstract for programmers, once the CPU is tightly close to the FPGA and there is no substantial performance difference between them (both the CPU and the FPGA run in the same clock frequency^2^).

Flows do not require anything more than a set of input information passed from the CPU to the FPGA through the BRIDGE. Furthermore, threads can spawn new threads while Flows are sole at any given time (due to the unicity of the FPGA). T threads can also be assigned to different pieces of program when created, but Flows are always bound by the FPGA configuration. If by any chance the CPU requests the creation of a second Flow while the first one is still active, the BRIDGE will silently ignore the request.

### 4.2.2 Flows Life-cycle

Like any thread, a Flow is assigned to different states during its life cycle. A Flow can be at any one of four states: starting, running, holding or ending. The Flow can only be in one of these states at any given time and they mandatorily follow the logic sequence of 'life-and-death': a Flow should first explicitly be created, then put in a running state (and eventually put to hold for the CPU) to finally be destroyed after its execution is complete.

An implicit waiting state can be derived from ending when the FPGA module is resumed before the return interrupt is issued. This would make the FPGA to continue running from the last state before the ending state is reached, essentially resuming the previous execution context. Figure 4.2 shows the life-cycle of a Flow through its possible states.

^2^In the current Alchemy implementation, only a single clock source is described in the process core, which is used by both the CPU and the FPGA. This decision was made to favor ease of implementation,
4.2. ALCHEMY PROGRAMMING MODEL

**Figure 4.2:** Flow life-cycle diagram.

starting comprehends the Flow initialization stage. As it can be seen in Section 5.1, the compiler abstracts this stage from the user code so that the Flow creation is performed under the hood. starting consists of initializing the BRIDGE registers and submitting an FPGA call request. It is during starting state that the Flow is set to run as blocking or non-blocking.

Blocking Flows run while the CPU waits. That means that the CPU stalls until the Flow reaches the ending state. This effectively puts the FPGA into the CPU pipeline and makes the Flow behave like a CPU instruction.

Non-blocking Flows run in parallel to the CPU. They make the system like any embedded system endowed with peripheral devices. The Flow will keep active until a Return interrupt is issued by the FPGA module.

Non-blocking Flows can also become blocking Flows in two situations. The active Flow becomes blocking, once the CPU is not performing any work until a Return interrupt is issued. The Flow can also become blocking for a defined amount of time. In this case, the CPU will wait for N clock cycles before forcing the Flow to enter ending state. Note that the Flow only becomes non-blocking when N is greater than 0.

The running state puts the Flow in execution. The FPGA clock signal is put to change and the logic circuit configured in the FPGA can work over its inputs to produce outputs. The Flow continues in that state regardless being blocking or non-blocking. The Flow can possibly exist for an infinite time period. It is the programmer’s responsibility to ensure a finite execution time for Flows, by properly describing this in the Verilog modules.

holding state puts the Flow in a constant wait for the CPU. This is basically synchronization state that can be directly implemented in the Verilog module. Finally, ending is the Flow’s final state, in which its life-cycle ends. At this point, the Return interrupt is issued by the BRIDGE and the CPU performs the tasks needed to harvest the results obtained from the FPGA.
4.2.3 Shared-memory Flowed environment

Threads can be considered small instances of programs. For that reason, they usually have their own private storage space (represented with a stack) and also can have access to global memory locations used by all threads. This is what is usually called a shared-memory environment: threads run in a common addressing spaces and their data can be directly shared among other threads by means of memory.

Once there is always one Flow at any given time, Alchemy shared-memory environment is based on the data exchange between the CPU and the FPGA, by means of the Shared-Memory resource (Section 3.4). Flows have the right to perform reads and writes in memory (by using the BRIDGE, as seen in Section 3.3.1), but they do no have formal private data storage as threads do. A typical workaround to provide private storage for Flows is the definition of FPGA only memory areas, created with compilation-time memory fences. This establishes virtual memory sections in which the C/Assembly program may not access, but do not ensure full run-time consistency. The FPGA module can also define certain CLBs to work as internal storage elements.

At any given time, when the CPU and the FPGA are running in parallel, concurrent access may be performed in memory. Alchemy Shared-Memory block allows only one read and one write at the same time so inconsistency may raise from local copies of shared data (stored in both the CPU or the FPGA). It is not defined any mutually exclusive access mechanism and this is specially dangerous for Flows because they do not have any way to establish locks, so the CPU is in charge to ensure accesses correctness for them.

Note that hazardous memory access is only prone to happen when the Flow is cerated as a non-blocking Flow. Blocking Flows will ensure that only single memory accesses are performed during execution and only requires the user to watch for volatile registers or memory locations in between FPGA calls.

Despite not defined nor implemented, C and Verilog libraries can be designers and embedded to the user application to implement Mutexes, Semaphores, Barriers and other mechanisms to ensure consistent memory handling during execution.

4.2.4 Synchronization with Flows

Synchronization is a term only applied to non-blocking Flows once they are usually running alongside with the C program. There are defined two synchronization methods for Flows, as it will be presented in Section 4.3.1.

Flows in turn, can also be synchronized with the CPU, and it can be done in two different ways. Both of them are not explicit hardware implementation of synchronization methods but rather are high-level conventions implemented by developers in their Verilog modules. This is called "manual synchronization".

Verilog modules can be designed to wait for a given input (indefinitely or for a certain number of clock cycles) to perform certain actions. This input can be either from memory or from the BRIDGE registers and the synchronization is achieved by the fact that the FPGA cannot execute its natural flow, unless this input changes to a given value.

Verilog designers usually implement this approach with state machines and that is exactly what is proposed as a synchronization mechanism performed by the Flows themselves. Figure 4.3 shows a manual synchronization example.

As it can be seen in the figure, the state machine forces the system to not deliver any output while the input in1 is not equal 1. This input is directly controlled by the CPU, so that it is possible to establish interesting communication methods between CPU and FPGA. Note that this is the only way in which the Flow is put in holding state.
4.2. ALCHEMY PROGRAMMING MODEL

... always @ (posedge clk)
begin
  state <= next_state;
  ...
  if (state == 2'b10)
  begin
    //Synchronization Block
    if (in1 == 1'b1)
    begin
      next_state <= 2'b11;
    end
    else
    begin
      next_state <= 2'b10;
    end
    //end: Synchronization Block
  end
  else
  begin
    return <= 1'b1;
  end
  
end
...

Figure 4.3: Manual Synchronization method in Verilog Modules.

4.2.5 Considerations

Alchemy programming model and the concept of Flows are high-level abstractions to support programmers to design their applications taking most out of the architecture. Developers shall consider the concepts presented in this Section when designing their applications once the Flow model catches most of the characteristics of the heterogeneous architecture implemented by Alchemy.

It is important to realize that the CPU itself can also define several threads to concur with an active Flow. They could be implemented with a time multiplexing method, that executes threads concurrently (but not in parallel), generally managed by an Operating System. In this case, the considerations presented in this Section are still valid, but the CPU must be treated as multiple CPUs, and the synchronization mechanisms for threads must be applied locally to ensure CPU execution correctness.
4.3 Alchemy High-Level Development Languages

The C language is a general-purpose programming language, designed by Dennis Ritchie in 1973 at AT&T Bell-labs, for the development of the Unix operating system. This computer language became officially standardized in 1980 by the ANSI X3J11 committee, and today it is among the most popular computer languages in the computer history [44].

This is a flexible, multi-platform, imperative computer language which is strongly typed, supports structured programming, lexical variable scope and recursion, while a static type system prevents many unintended operations. C also includes powerful pointer manipulation capabilities and bit-wise operators. It became specially popular among embedded systems design once it provides closer proximity to the hardware. This usually cannot be provided by high-level languages such as Java, C# or Python.

Alchemy C is a non-ANSI C once it defines a subset of syntactic elements that provides FPGA Cross-Procedure control/synchronization capabilities. It also supports most of the MSP430 standard libraries.

Verilog is a Hardware description language designed in early 1980 by Prabhu Goel and Phill Moorby at Automated Integrated Design Systems, which was purchased by Cadence Design Systems in 1990. It is mostly used as a design verification language of digital circuits (also used in the verification of analog and mixed-signal circuits). Verilog is currently a non-proprietary, public domain language which is standardized by the IEEE Std. 1364-1995 and has been improved and revised since then.

This HDL was designed to make easier the hardware description, turning it very similar to the software development. Verilog is also used for simulation of digital circuits and low level hardware (transistors and its interconnections) description of sequential and combinational logic circuits.

Alchemy Verilog does not heavily extends the language syntactic structures, but instead, defines a set of conventions that allows the communication between the CPU, FPGA and the BRIDGE. Those conventions are intended to provide higher flexibility in terms of design of applications.

4.3.1 C Non-Standard Language Statements

In order to hide from users details regarding the BRIDGE protocol as it is meant to be used from C programs, it is defined a set of non-standard C language statements. This approach was chosen to simplify the development process, since users would not have to install and learn extensive libraries to develop their applications for Alchemy. They are used to control Cross-Procedure calls and their intrinsic management requirements. Every non-standard statements in C start with @ prefix (except for the Acknowledge Statement). This Section provides a comprehensive overview over the syntactic and semantic characteristics of these statements.

Cross-Procedure Call Request Statements

When it is required to request the FPGA to execute whichever Cross-Procedure, users can make use of two special syntactic statements specially designed for that purpose. @parallel.call() and @sequential.call() can directly request the FPGA to run a Cross-Procedure, over a set of provided input parameters and implicitly providing multiple returns for the call. Figure 4.4 shows the structure of Call Request Statements:

In the figure, both parallel and sequential variations of the statement have the same syntax. @parallel or @sequential informs the type of the call, or more specifically, how the Flow
associated to the call will behave during runtime. Both versions also accept a list of parameters
that can have 0 up to 64 entries.

It is important to notice that a Cross-Procedure parameter can be an input or an output
parameter, and that \texttt{@parallel.call} and \texttt{@sequential.call} provides \texttt{void} returns for attribu-
tions. The type of the parameter is determined by the type of its related pin in the Verilog
Module. Any I/O signal in the module must be addressed as a parameter in the program. If
the pin is an Input pin, the related parameter, in the Cross-Procedure Call, will drive this pin.
In other words, the parameter value is copied to the Module input pin. Likewise, output pins
have their content copied to the their related parameter.

The parameter type is determined by the pin declaration order in the Verilog Module. More
than that, the parameter type (as a C variable) must fit the Related pin size. This is crucial
when the pin is a signal cluster. If a parameter is smaller than its related pin, it will generate a
compilation error. If a second \texttt{@parallel} statement is called while a first one is still being run
by the FPGA, the second request is ignored by the BRIDGE (this essentially tries to create a
new \texttt{Flow}, which is a not legal operation).

Returns are automatically implied from output parameters. They are made available when
the Cross-Procedure execution is finished. Using these parameters during the procedure exe-
cution will provide its value as it was prior to the procedure call.

The parameter list is meant to be separated with coma, like any function parameter list.
No numeric constants nor pointer types are allowed as parameters and the \texttt{.call} suffix is
mandatory.

Cross-Procedure Synchronization Barrier

When running a Cross-Procedure in parallel to the CPU, users may want to hold the CPU
execution until the FPGA quits running the requested procedure. That is exactly the function
of \texttt{@wait} statement. The CPU will wait for the FPGA until a Return Interrupt is issued by
the FPGA. \texttt{@wait} accepts no input parameters and provides no returns. Figure 4.5 shows the
statement usage.

\begin{verbatim}
1: \texttt{@parallel.call(<parameter1>,<parameter2>,...);}
2: \texttt{@wait;}
\end{verbatim}

\textbf{Figure 4.5:} \texttt{@wait} statement usage.

The statement cannot be aborted after invoked by the user code. It essentially converts a
non-blocking \texttt{Flow} into a blocking \texttt{Flow}. Furthermore, if the CPU executes an \texttt{@wait} statement
when the FPGA is idle, it will cause no effect in the CPU. Hence, it never affects sequential
Cross-Procedure calls.

An \texttt{@wait} can be emitted at any point during the execution of a Cross-Procedure. The
CPU may have to wait for the same number of machine cycles as required by the FPGA to run
Cross-Procedure Scheduling Statement

FPGA calls can run for virtually an infinite number of clock-cycles before issuing a Return interrupt in the CPU. Users are allowed to control the number of clock-cycles the FPGA is meant to run a Cross-Procedure by defining \( @\text{stall}[N] \) statement. The user can provide a 16-bit constant \( N \) representing the number of clock cycles the BRIDGE will enable the clock signal to change.

\( @\text{stall} \) syntax requires the number of clock-cycles to be a constant, and it must be placed between \( [ \) and \( ] \). The BRIDGE is in charge to count the clock cycles before the FPGA is blocked. When blocked, the BRIDGE issues a return interrupt to the CPU, so that the call output parameters can be harvested in the C program. Figure 4.6 shows \( @\text{stall} \) statement usage.

```
1: @parallel.call(<parameter1>,<parameter2>,...);
2: @stall[65000];
```

Figure 4.6: \( @\text{stall} \) statement usage.

The scheduling statement is important to ensure that the FPGA will not take too much time running, specially when the Cross-Procedure is requested in sequential mode. It is not as critical for parallel calls, once \( @\text{destroy} \) statement can abort FPGA Cross-Procedure.

Cross-Procedure Abortion Statement

In order to force the BRIDGE to stop the FPGA from running, the user may use \( @\text{destroy} \) statement. This extinguishes the current Flow in the system, invoking a Return Interrupt afterwards. Figure 4.7 shows \( @\text{destroy} \) statement.

```
1: @parallel.call(<parameter1>,<parameter2>,...);
2: ...
3: @destroy
```

Figure 4.7: \( @\text{destroy} \) statement usage.

Abortion is useful to control the FPGA pace, depending on certain conditions reached in the C Program. For example, it can be used in shutdown routines to Stop the FPGA and save its outputs in memory, before effectively turning the system OFF.

Differently to \( @\text{stall} \) statements, \( @\text{destroy} \) clears the scheduling count and stops the FPGA with a single machine cycle, and no further setup steps are needed other than the Return Interrupt itself. Once destroyed, Cross-Procedure calls can be normally emitted again.
#reach Acknowledgement Statement

In order to provide visibility of the intended top-level Verilog Module in the C program, it is defined an Acknowledge statement that informs the Compiler where to look for Verilog source code in the current compilation directory. This statement is only required in the .c file in which the main function is declared. The syntax of this element is similar to a standard C #include. The user only needs to inform the name of the top-level Verilog file (followed by its extension) surrounded by ".". Figure 4.8 shows #reach acknowledge statement usage.

```
1: #include <Alchemy.h>
2: #reach "module.v"
```

Figure 4.8: #reach statement usage.

4.3.2 Verilog Non-Standard Language Statements

In order to simplify the task of interacting Verilog Modules, there are defined a few non-standard statements and conventions to guide the development of correct modules in Verilog. Users may mandatorily use the conventions when implementing Memory interface from the FPGA (for example), otherwise the Synthesizer and the Preprocessing tool will not identify them as so.

Verilog has a very lean syntax so that Alchemy do not heavily load it with new elements. There exist only a single new statement for the language, and the rest of this section presents naming conventions and their functionality in the module.

Clock Signal

Most of Verilog modules describe combinational and sequential logic in their structures. It is well know that a basic requisite for sequential logic is the existence of a clock signal, to allow transition of states in the digital logic.

Users can ideally define any source, in their module hierarchy to provide clock signals. However, users are only allowed to use its internal clock source. In order to simplify the access, it is required any reference to clock to be made through a standard input signal called clk. Figure 4.9 shows the declaration and an usage example of clk signal.

```
1: input clk;
2: always @(posedge clk)
3: begin
4:  ...
5: end
```

Figure 4.9: clk signal declaration and usage on always block.

As it can be seen in the figure, clk signal must mandatorily be declared as an input signal with unitary length. The signal can be included in any position in the module definition list, but its name must be typed exactly as depicted. The absent of clk signal will raise a synthesis
error. This convention is useful to optimize the FPGA Matrix usage, once the signal source lies in the BRIDGE.

The Synthesizer automatically identifies the CLBs in which clock signals are required and directly tie them to the global clock source, asserting the register edge mode accordingly.

Return Signal

Return signal is one of the most explicit BRIDGE Protocol conventions that must be included in Verilog modules. It allows users to effectively manage Cross-Procedure execution flow, by giving the control back to the CPU. Like clk signal, return must mandatorily be included in the user module as an unitary length output signal. Figure 4.10 shows return signal declaration and usage.

```
1: output return
2: always @ (...)
3: begin
4: ...
5: return <= 1'b1;
6: end
```

Figure 4.10: return signal declaration and usage on always block.

return will only change the system state if it is assigned to a logic 1. This informs the BRIDGE Monitoring Unit that the FPGA Cross-Procedure is not meant to keep running and will stop the FPGA Clock Signal. Users must be careful when using return once it must be set to zero right after a new FPGA call is issued.

The Synthesizer identifies the return signal and directly ties its assigned I/O Block to the BRIDGE Monitoring Unit. Users do not need to worry about anything other than changing the signal, which highly simplifies the development and debugging processes. Users are also able to assign other signals to return to perform an indirect signal manipulation.

Memory Manipulation Conventions

In order to allow memory manipulation from the FPGA, users are required to define standardized I/O signals and implement a simple convention to coordinate read and write operations. It is not defined any new syntactic element in the language for that purpose, so that users can have more flexibility when including memory based operations in their modules.

In order to send and receive content to/from memory, there are defined two data/address busses. These are called memory_out_bus and memory_in_bus. Both busses are 16-bit wide, and memory_out_bus is used in to send both addresses and data to be written in the memory. Figure 4.11 shows the declaration of the busses.

```
1: output [15:0] memory_in_bus;
2: input [15:0] memory_out_bus;
```

Figure 4.11: Memory busses declaration.
Three control signals are used to manage the memory operations. It is defined a two-line operation-control bus that informs the memory operation. `memory_read_request` and `memory_write_request` inform the BRIDGE the operation type by setting one of the lines to 1 exclusively. If `memory_read_request` is equal to 1 and `memory_write_request` is equal to 0 then a memory read operation is performed. Likewise, if `memory_write_request` is equal to 1 and `memory_read_request` is equal to 0, a memory write is performed.

`memory_data_ready` is a flag signal used to inform the memory control internal state machine the memory status. Like the operation-control bus, `memory_data_ready` is a single bit signal that is directly tied to the BRIDGE. Figure 4.12 shows the declaration of these signals.

```
1: output memory_data_ready;
2: input memory_read_request;
3: input memory_write_request;
```

Figure 4.12: Memory management signals declaration.

Memory accesses are performed using a simple state machine that is synchronized with the BRIDGE protocol. It is required to define two state control for current and next stage, as well as a single `@always` block for this.

In order to read content from memory, it is first required to write the address to be read in `memory_in_bus`, transitioning to the next machine state. After that, `memory_read_request` is set to 1 and `memory_write_request` is set to 0. At this point, `memory_read_request` is set back to 0 and the machine enters in a waiting state, in which `memory_data_ready` is expected to change to 1. After that, the read value will be available in `memory_out_bus`.

Memory writes are similar to read operations but, when informing the operation type, `memory_write_request` must be set to 1 and `memory_read_request` have to be set to 0. The content to be written must be put in `memory_in_bus`, right after `memory_data_ready` changes to 1. The change in the data bus will make `memory_data_ready` change back to 0 and the state machine enters in an idle state, waiting for the memory to acknowledge the write operation, by setting `memory_data_ready` to 1. Figure 4.13 shows examples of memory read and memory write operations as the should be described in Verilog.

`reach Acknowledgement Statement`

In order to aid the Synthesizer to locate the `.c` source file with the entry-point function in the current directory, users must mandatorily include the `reach` keyword in their top-level source file. Only the top-level module must include this statement and its syntax consists of providing the C file name, with extension, between quotation marks. The Acknowledge Statement syntax can be seen in Figure 4.14.

4.4 Alchemy Custom Assembly Structures

Assembly Languages are usually classified as low-level development languages due to its proximity to the machine architecture. The minimal abstractions provided by these languages does not hide intrinsic architectural features from users, making software and hardware development more expensive (in terms of complexity and time) but allowing a better exploration of implementation possibilities, potentially improving performance.
Memory handling state machines, written in Verilog. The addresses and the content to be written in memory are placeholders.

```
1: \`include "Alchemy.v"
2: \`reach "main.c"
```

Figure 4.14: ‘reach keyword usage.

There are defined two different Assembly languages, one for the CPU programming, called Program Assembly, and other for the FPGA configuration, called Module Assembly. Program Assembly language inherits most of MSP430 Assembly structure, including syntactic and semantic, while including six new instructions. Module Assembly is based on three different language, defined by version 7.0 of Verilog-to-Route [13] project. No new structures are defined in this language. The complete description of these languages is out of the scope of this document, but a short description of the new structures is given in this section.

The new BRIDGE instructions are used to allow Cross-Procedure calls and BRIDGE protocol management from C programs. The name of the instruction is presented with its base opcode as presented in section 3.3.1. The instructions are presented next:

- **stb [0000-0-11]**: Store in BRIDGE Register. Alchemy custom instruction designed to allow C programs to send input parameters to the BRIDGE, prior to a Cross-Procedure Call. The operation can only be executed in word mode and requires a source operand, residing either in registers or memory (accessible with any addressing mode) and a target BRIDGE register, represented by a two-bit value. Bits related to output I/O Blocks in the FPGA are not written by this instruction.

- **ldb [0000-0-00]**: Load from BRIDGE Register. Custom instruction used to load BRIDGE register into a destination operand, being either a register or memory location. Used mainly to recover the return values from a Cross-Procedure Call. The source operand is one of the BRIDGE registers (addressed with two bits). Bits written by stb are also restored with ldb instruction.

- **dst [111-10]**: Destroy Call [after]. Aborts a Cross-Procedure call. It essentially destroys the active Flow, forcing the emission of a Return Interrupt even though the BRIDGE
Monitoring Unit did not detect it. This stops the FPGA clock signal (it will not be allowed to change until another Cross-Procedure call is requested). This instruction accepts a single input parameter that is mandatorily stored in the next word following the instruction’s. This parameter is the number of clock cycles used to schedule the Cross-Procedure abortion. This 16-bit unsigned number is directly written in the BRIDGE Schedule Register, and will abort the call immediately if and only if it is equal to zero. Otherwise, the BRIDGE will decrement the register until it reaches zero, before aborting the call. This instruction can only be used in word mode.

- **seq [111-00]**: Sequential Cross-Procedure Call. Creates a blocking Flow in the FPGA. It essentially puts the CPU in a wait state and allows the FPGA clock signal to change. The call persists until a Return Interrupter is emitted. This instruction requires a four bit Register Map, informing which BRIDGE registers will be forwarded as input to the FPGA. This bits are organized as following: SWNE, where the least significant bit E informs that the EAST (B0) is copied to their related FPGA I/O Blocks, if it is set to 1. Likewise, N means NORTH or B1 register, W means WEST or B2 and S means SOUTH or B3 and they are only forwarded to the FPGA if their related bits are set to 1. The instruction allows any bit to be set, so zero up to the four registers can be forwarded to the FPGA, with a single call. The assembler automatically ignores masks larger than 15, casting the value down to four bits. This instruction can only run in byte mode.

- **par [111-01]**: Parallel Cross-Procedure Call. Creates a non-blocking Flow in the FPGA, following the calling of a parameter convention defined for seq instruction. This instruction can only run in byte mode.

- **abt [Virtual Instruction]**: Abort Cross-Procedure Call. Equivalent to dst #0. Immediately destroys the current Flow and emit a Cross-Procedure Call. Requires no input parameter and can only execute in word mode.
Chapter 5

Transmute Toolchain

In order to support programmers to develop applications for a given platform, a set of tools must be developed to aid the design process. These tools are IDEs, SDKs and debuggers. Yet, a very basic and important tool is the Compiler.

Despite being able to fully develop applications with only assembly languages, users usually tend to make use of high-level programming languages or hardware description languages when developing for CPUs or FPGAs. This abstraction layer generally increases application complexity, without drastically increasing the development difficulty, allowing users to take advantage of the target platform.

Alchemy is no exception. Unlike most of low-end processor architectures, Alchemy crosses two hardware domains, which results in four different assembly languages (as presented in Chapter 4). This can highly discourage new users and must ideally be addressed with a compilation/synthesis toolchain. Henceforth, it was necessary to develop new tools taking into account Alchemy architectural singularities. These tools were called Transmute, and were developed as a command line utility built with C/C++, Python and Perl, for Linux.

Alchemy Transmute toolchain was designed to ideally simplify the development process by allowing users to write their programs with high-level languages. Transmute requires users to use C language for software programming and Verilog language for hardware description. The compiler/synthesizer can directly translate high-level C programs into executable softwares in the CPU, and high-level Verilog modules into hardware configurations in the FPGA. Furthermore, Transmute can handle and translate cross-domain constructions, allowing CPU-FPGA interaction. Low-level Program Assembly and Module Assembly development is also available.

This chapter presents the basic Alchemy build flow as well as the various build tools. The term build is a general term used to refer to both compilation and synthesis as performed for Alchemy applications. The build process outputs two memory files for program and configuration bitstream.

5.1 Transmute Toolchain

Transmute means "change into another nature, substance, form, or condition". In this project context, this definition is accurate to describe Transmute toolchain functionality. It was designed to take two (or optionally one) user source code, written in C and Verilog, and convert them into memory files to be embedded in the Program Memory and Configuration Memory.

1Alchemy term for compilation or synthesis is build.
Transmute builds applications for Alchemy processor by providing a series of wrapper applications that surround the C compiler and the Verilog synthesizer. The C compiler is a branch of Texas' Instruments MSP430-GCC project [29] and the Verilog Synthesizer is a branch of version 7.0 of Verilog-to-Route (VTR) [33] project. Both the compiler and the synthesizer source codes were not modified to fit in Transmute Toolchain. A simplified compilation/synthesis flow-diagram is presented in Figure 5.1.

The compilation/synthesis flow was designed to allow fast and coherent application build. After an initial preprocessing, the hardware is fully synthesized and packed before the software part gets compiled. This order is also required due to the expressive data dependency between the building steps.

The wrapper applications were strictly developed with C++ and implement the cross-domain layer that allows C programs to interact with Verilog modules. There was implemented three applications: Alkahest, Azoth and Archeus. Alkahest is a high-level parser, presented in Section 5.2.1. Azoth is an Alchemy-FPGA technology mapper, presented in Section 5.2.3. Archeus is a cross-domain memory patcher, presented in Section 5.2.6. A Python front-end was also implemented to handle user input and file-system interactions.

Transmute tries to simplify the development process by explicitly requiring few options to build programs for Alchemy architecture. Transmute can be executed by running $\texttt{transmute}$ or $\texttt{tmute}$.

(1) $\texttt{transmute}$ \texttt{software.c} and \texttt{hardware.v} into output
(2) $\texttt{tmute}$ only \texttt{software.c} into output

Figure 5.2: Transmute command examples.

The command shown in Figure 5.2 (1) asks Transmute to compile/synthesize a complete Alchemy application, including C programs and Verilog modules. These top-level entities can optionally include C headers and Verilog modules to implement their functionalities. The and keyword informs that both top-level source codes are going to be used in the compilation and synthesis processes. The order in which the source files appear is not important for Transmute. The into keyword informs that the word following it will hold the output prefix used in the resulting files.

Figure 5.2 (2) asks Transmute to compile or synthesize a single source file, depending on its type. Transmute will automatically generate 'dummy' C programs or 'null' Verilog modules, taking the processor CPU or FPGA to master mode, depending on the provided source code. The keyword only is required to run Transmute in this mode.

5.2 Transmute Compilation and Synthesis Flow

Transmute builds process is bound by several intermediate stages that feed the compilation and synthesis flows, implementing cross-domain structures to allow C and Verilog interaction.

The Controller is essentially the Transmute user interface controller and the build flow coordinator. It is the tool summoned when the user types $\texttt{transmute}$ or $\texttt{tmute}$. It is in charge to parse the commands and adjust the compilation flow. Controller also exports environment variables as well as creates and deletes directories and files.

As a flow coordinator, Controller emits and checks the outputs of each tool required to take
5.2. TRANSMUTE COMPILATION AND SYNTHESIS FLOW

Figure 5.1: Transmute Toolchain Compilation/Synthesis Flow.
part in the build processes. It is also responsible to actively emit commands to execute these tools. Controller is implemented with Python.

5.2.1 Source Code Preprocessing with Alkahest

The build process is started by calling Alkahest. Alkahest is a preprocessing tool in the Transmute Toolchain. Its objective is to identify, parse, analyze and adjust Alchemy’s nonstandard language structures (those presented in Chapter 4). Alkahest requires a .c file containing the C source and a .v file with a Verilog module (main module). An output prefix is also required.

Alkahest will always process Verilog files before processing C files. Once C programs can contain direct FPGA calls which requires argument passing (to be written in the BRIDGE), the Verilog source must be firstly classified in terms of its inputs and output pins. Alkahest identifies and records the type (input or output), the dimension (simple signal or signal cluster) and the name of each I/O pin declared in the Verilog module.

In Verilog, pins can be either declared directly with the module name or within the module body. Alkahest can handle these two syntaxes and also identify inconsistencies between the definition and declaration. Based on this signal analysis, Alkahest builds a signal table. This table will be used in further build stages that will match @parallel.call or @sequential.call structures with the actual FPGA I/O setup.

In the Verilog source file, Alkahest checks for ‘reach and ‘include "Alchemy.v" statements, and will make sure that the Verilog file is acknowledged in the C file. Any inconsistency in those structures will raise an error informing the line number and its nature. All non-standard Verilog language structures are converted into valid structures in this stage.

When processing C files, Alkahest identifies each @parallel.call and @sequential.call included in the source code. It parses the calls and checks their syntax. After that, Alkahest analyses each parameter passed in the call, matching them with C variables. Note that the parameters must be explicitly declared as variables in the call context. The C variable declaration syntax is also checked in this stage.

After that, Alkahest checks if the parameters passed to the call are in match with the FPGA I/O pins. It should guarantee that each parameter type is of at least the size of the I/O pin it is related to. This is done by comparing each parameter type, with the size of the pins extracted from the Verilog source file. The parameter order in the call is also taken into account during this stage. Parameters are matched with module pins by considering their relative position in the call list and in the module pin definition list (pin names declared with the module name, excluding return, clk, memory_data_ready, memory_read_request, memory_write_request, memory_in_bus and memory_out_bus). Alkahest also makes sure that both lists have the same number of entries.

FPGA calls can hold parameters of at most 62 bits (which fits in a long long variable). If a parameter passed in the call is shorter than its related Verilog module pin, Alkahest will raise an error. Figure 5.3 shows a parameter-to-pin relationship diagram.

In the figure, it can be seen the relationship between calling parameters and module pin. __param1 is related to pin1 once their positions are the same in the parameter list and in the pin declaration list. pin1 is the first in the pin declaration list, despite the fact that it was declared after clk pin.

The figure also depicts the relationship between the parameters and the pins. Taking the pair __param2 and pin2 as example, it can be seen that part of __param2 variable is not used, once it is a char variable and the pin2 is a signal cluster of 3 bits. When mapping the parameter to the pin, its least significant bits are those that will be written in the BRIDGE. The remaining bits are ignored. Table 5.1 shows parameter sizes in terms of pin sizes.
5.2. TRANSMUTE COMPILATION AND SYNTHESIS FLOW

Figure 5.3: Alkahest parameter-to-pin relationship.

Table 5.1: Parameter size related to pin size.

<table>
<thead>
<tr>
<th>Pin Size</th>
<th>Parameter Type</th>
<th>Length (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>from [0:0] to [7:0]</td>
<td>char</td>
<td>8</td>
</tr>
<tr>
<td>from [8:0] to [15:0]</td>
<td>short</td>
<td>16</td>
</tr>
<tr>
<td>from [16:0] to [31:0]</td>
<td>long</td>
<td>32</td>
</tr>
<tr>
<td>from [31:0] to [61:0]</td>
<td>long long</td>
<td>64 (only 62 are visible)</td>
</tr>
</tbody>
</table>

Alkahest handles nonstandard C structures defined by Alchemy. #include <Alchemy> statement is resolved and replaced by standard C includes and the #reach statement is checked to determine if the Verilog Module acknowledges the C program.

Alkahest is also responsible to prepare the interface between the C program and the BRIDGE. Considering the type of each parameter and its related I/O pins, Alkahest emits the content of GET. GET stands for Global Exchange Table and its entries are used to store pointers to each parameter used in @parallel.call or @sequential.call throughout the code. GET can be virtually of any size and resides in the lower parts of the Shared Memory and its allocation is responsibility of the C compiler. The number of GET entries is defined after the number of I/O pins in the Verilog Module. Alkahest only allocates space for it, declaring each entry as a global variable. The Global Exchange Table is the an auto-generated element inserted in the user code, and cannot be modified by the user.

GET is necessary once Alkahest treats each @parallel.call and @sequential. call as a standard function call. In fact, an auto-generated function is embedded in the C code and every FPGA call is replaced by a call for this function. Alkahest calls it "Cross-Procedure Request Routine" and only part of its content is implemented at this stage. The Cross-Procedure Request Routine is a void-typed function and its name is composed by the _alchemy_ prefix, followed by a 112 bits sequence of random character. This naming system intends to avoid collisions with user-defined functions. Figure 5.4 shows GET structure and dynamics.
At the beginning of the Cross-Procedure Request Routine, the GET entries are updated to store the address of the current call parameters. GET entries must be updated whenever an FPGA Cross-Procedure call is performed, once the return values are meant to be stored back in the variables passed as requested call parameters. The Cross-Procedure Request Routine receives a reference of each parameter passed to @parallel.call or @sequential.call statements, and stores them in the corresponding GET slots.

Alkahest also identifies the FPGA Cross-Procedure call type and emits correct Cross-Procedure Request Routine calls, according to the call type. An extra parameter informs if the call is parallel or sequential. The Cross-Procedure Request Routine implements a different execution flows for each method using if-else block. These blocks are filled with dummy code to be replaced by actual BRIDGE instructions. Alkahest also handles @wait synchronization statements, replacing them with busy wait while loops that constantly checks a global integer constant called __FREEZE__. This constant should also be equal to 1, except when returning from a Return ISR, in which its value is equal to 0. The next instruction right after the while sets __FREEZE__ back to 1, so the system can wait for other parallel calls, if needed. Other syntactic element is the @stall statement which is converted into a low level instruction set with a dadd #0, r3 and a nop instruction, which will be replaced by valid BRIDGE instructions later on.

A replacement flag, coded as !ARCHEUS! is included in the beginning of the Cross-Procedure Request Routine to allow code insertion in further processing stages. A final processing stage defines and does not implement the FPGA Return ISR, which will be implemented by Archeus (Section 5.2.6). Alkahest only inserts the ?ARCHEUS? so that Archeus can embedded code for the ISR. The ISR naming system follows the Cross-Procedure Request Routine standard but the suffix __return_isr is appended to the function name.

Alkahest will create a new version of both C source code and Verilog source code, appending
5.2. TRANSMUTE COMPILATION AND SYNTHESIS FLOW

the temp. prefix to their names. These files will be forwarded to other tools to continue the build process. Alkahest also creates a signal table file, with the extension .sig.

At this point, Transmute starts the hardware synthesis flow, working mainly on the Verilog module. The software compilation flow is put into a hold, while other intermediate results are created by the Synthesizer.

5.2.2 Verilog Module Synthesis, Place and Route with VTR

The CAD flow for Alchemy was designed to be like any other FPGA platform. Users design circuits with a high-level hardware description language, which abstracts the FPGA platform singularities and is used as an input for a synthesis, optimization and packing tool that generates the configuration bit-flow. VTR is usually referred as AVTR (Alchemy-VTR).

Instead of creating its own synthesizer, Alchemy makes use of Verilog-to-Route tool-chain which integrates an elaboration and synthesis tool (ODIN II), a logic synthesizer and FPGA technology mapping tool (ABC) and an FPGA packing, placement and routing tool (VPR). Besides being a full CAD flow tool, VTR also provides a benchmark tool and can be used to perform complex FPGA architecture researchers.

One of the greatest advantages of VTR is the possibility of describing an FPGA architecture with a simple and comprehensive XML format, which allows external projects to seamlessly bind their custom FPGA architectures to the CAD flow.

Transmute does exactly that: a custom FPGA architecture description was developed to represent Alchemy FPGA and the design flow is used to synthesize, place and route custom user circuits. VTR execution flow can be seen in Figure 5.5

![VTR execution flow diagram.](image)

VTR is essentially a set of Perl scripts that coordinate the flow of the synthesis process. Like Transmute, these scripts handle task emissions, file creation and deletion among several other things. There are three intermediate files of major importance for Transmute: a placement file (.place file), a route file (.route file) and a logic model circuit coded in BLIF format (.pre-vpr.blif file).

ODIN II is an open-source tool used as the front-end Verilog language parser, used to convert the user-defined circuit into a flattened netlist representation which maps structural elements to the target FPGA building blocks. ODIN II takes as input the XML architecture description format and performs inferencing and partial mapping during which there are identified hard
circuits in the Verilog design and then determined how to map them to the hard circuits on the FPGA architecture [33].

ODIN II analyses the circuit by converting the Verilog source code into an Abstract Syntax Tree (AST) and a netlist format, which provide hierarchical information to allow the identification of logical structures in the user circuit. ODIN II internal organization and functionality is beyond the scope of this document. Further information can be found in [33]. ODIN II results are purely intermediate in the build process surrounded by Transmute. Its outputs are used as input to the next stage.

ABC follows ODIN II in the synthesis flow. ABC combines scalable logic optimization based on And-Inverter Graphs (AIGs), optimal-delay DAG-based technology mapping for look-up tables and standard cells, and algorithms for sequential synthesis and verification. It is used in the logic optimization and technology mapping stages [1].

The last tool in VTR flow is VPR. VPR implements placement and routing algorithms for target FPGA platforms described in XML format. It takes as input a technology-mapped user circuit, preprocessed with ABC. VPR outputs a couple of configuration files that describe the placement and routing of the circuit in the target FPGA platform, as well as a set of statistics regarding the circuit parameters.

The results generated with VTR are used as input of further build steps by Transmute. Transmute will pack the output files and forward them as input for the next synthesis stage, performed by Azoth.

### 5.2.3 Device Packing with Azoth

Azoth is the tool responsible to generate the configuration memory output file used by the Configuration Controller to configure the FPGA circuitry. It parses the logic model, the placement and the routing files created by VTR, converting them into configuration bitstreams.

Azoth inputs are .pre-vpr.blif, .place, .route files and an output prefix. Its outputs are a BRIDGE map (.map file) and a hardware image memory file (.hmg file). The BRIDGE map consists of a list of I/O Blocks (organized according to the description presented in Figure 3.24) followed by the Verilog module pin bits attached to them. The hardware image memory file holds a list of bits representing the desired state of each configurable spot in the FPGA.

Azoth will first parse the placement file and analyze its content to identify each CLB and I/O Block used by the user circuit. These blocks are then classified in terms of their relative position on its proper coordinate system. Azoth determines if a block is an input or output, according to information parsed from the .place file.

A second parse stage consists of analyzing the content of .route files. Azoth will extract and assemble each route described in the route file and store them. The routes are presented as sequences of wires with a begin and an end in CLBs or I/O Blocks. Each wire in the route has an ID and a coordinate pair in the segment coordinate system.

The .blif parsing is the last input analysis stage performed by Azoth. It consists of analyzing each node in the file, extracting their name and their logic structure in single-output-cover format. Azoth also converts the logic function to conjunctive normal form for simpler manipulation in later stages.

VTR will identify each node in the .blif file with its relative index in the CLB matrix or I/O Block list. Azoth takes this information to correctly map each logic functions to its corresponding CLB 4-LUT configuration and to map interconnects to the I/O Blocks. A special list of registers is also included in the file and is used by Azoth to determine whether the internal CLB latch will be used and which will be its initial value.

Once gathered the synthesized Verilog module information, Azoth will start to assemble
the configuration bit-stream that will be used to configure the FPGA. The tool was to support virtually any FPGA matrix size, so that future versions of Alchemy (with potentially larger FPGA matrix) can be defined. The only parameter required by Azoth aside from the input files is the FPGA matrix size. For the current Architecture version, the Matrix size was defined as 16.

The FPGA is packed by first building the routes described in the .route file. This is done by following the nodes (I/O Blocks or CLBs) in each route listed in the file and inferring the interconnects used to follow this path. The route file does not provide any information regarding interconnects, so that Azoth must convert the CLBs and I/O Blocks coordinate system into the actual interconnect coordinates.

Once the selected interconnects are identified, Azoth maps each switch matrix in between interconnects, and specifies the configuration of its internal tri-state buffers. Azoth will also map the tri-state buffers tied to the CLBs to allow communication with other CLBs throughout the route. This procedure consists of analyzing the pin number attribute listed in the .route file, taking into account the position of this pin in relation to the CLB itself. This will determine if a tri-state buffer will be tied to an input or an output and also which track will be used to forward the signal.

Another packing state defines the functionality of each I/O Block. This consists of analyzing the routing data extracted from the .route file and determine the netlist end-points.

CLBs functionalities are set according to the logic functions described in the .blif file. The first element to be analyzed is LUT. Its internal bits are set according to the logic function extracted from the .blif file. The internal CLB structure is defined by determining whether the register is enabled or not. If enabled, Azoth will determine the configuration bits for the internal multiplexers to allow communication between the register, LUT and the external FPGA circuitry. If the register is not meant to be used, the LUT outputs are directly tied to the CLB outputs. Minor internal CLB configuration bit can define the clock-edge type for the register, the LUT input order, asynchronous register set and reset and LUT modes (as described in Section 3.2.5).

Finally, Azoth generates the bitstream to configure the BRIDGE. As presented in Section 3.3.4, the BRIDGE has a set of multiplexers and demultiplexers that define the signal source for the BRIDGE bits. Azoth generates the configuration bits for the multiplexers and demultiplexers on each bit, by analyzing the I/O block types inferred from the .route file.

Whenever memory interface signals are detected in the module, the BRIDGE will have its memory proxy functionality enabled to watch I/O Blocks as inputs for memory access (either data/address signals or status signals) or outputs from the memory (data or status signals). The BRIDGE also is meant to watch the return signal in the FPGA. Azoth finds the position of this signal in the FPGA I/O Blocks and determine the configuration bits in the return-multiplexer so that the CPU can receive the interrupt request.

A final processing stage consists of the generation of the bit-stream to be written in the configuration memory. Azoth will align the stream bits so they fit in a byte each. Following the Configuration Controller requirements, Azoth stores CLB configuration bits in the lowest memory words. Switch matrices and interconnect (tri-state buffers) configuration bits are written after the CLB configuration, followed by the BRIDGE configuration bits. Azoth outputs a hardware-image memory file (.hmg file) and a BRIDGE bits map (.map file). The .map file relates each BRIDGE bit to the I/O signals in the Verilog module, to be used in further build stages.
5.2.4 FPGA Cross-Procedure call request emission with Archeus

FPGA Cross-Procedure call requests are highly dependent on the BRIDGE organization. In order to keep it abstract for users (when calling the FPGA from high-level languages), the input parameters passed to @parallel or @sequential statements must be treated as a single contiguous data resource. In fact, VTR does break Verilog I/O signals throughout the FPGA surrounding I/O Blocks and the BRIDGE registers are configured to follow this organization. Henceforth it is required to break the bits of each parameter in order forward them to the correct I/O Blocks in the FPGA. This is the first operation performed by Archeus when called right after Azoth.

Archeus takes as inputs the preprocessed C source file (generated with Alkahest), the BRIDGE map (generated with Azoth) and a signal list extracted from the Verilog module (.sig) file. It will first identify the position of each bit of each I/O signal in the Verilog module and generate specific bit manipulation instructions to break and organize the bits of the call parameters in their right position in the BRIDGE.

Once the BRIDGE classifies the FPGA I/O Blocks into four classes, represented by East, North, West and South registers, Azoth will provide code to fill the required bits in each one of them. The resulting code is placed in the Cross-Procedure Request Routine so that the parameters can be treated independently from the Cross-Procedure call.

Archeus generates code to update the GET table entries so the parameters passed in a Cross-Procedure call can be tracked after the request return. Once the FPGA finishes its execution, Archeus disassembles the BRIDGE registers so that the output parameters can be sent back to the C program as a single monolithic memory resource. The disassembled code is included in the Return ISR and it basically consists of performing bit manipulation over 16-bit data read from the BRIDGE registers. The GET table is used in this stage to provide the correct reference for each call parameter. Figure 5.6 shows sample code generated by Archeus:
As shown in the figure, Archeus includes code at the lines marked by Alkahest with !ARCHEUS! and ?ARCHEUS? directives, so that they do not appear in further build stages. Archeus will include several dummy instructions like `nop` or `dadd #0, r3/r4/r5` so that they can be replaced by valid BRIDGE instructions during further build stages. These dummy instructions are only intended to reserve room in the memory for the BRIDGE instructions, and shall not be optimized out by the compiler. At this point, the C program is ready to be compiled by GCC, in the next build stage.

### 5.2.5 C Programs Compilation, Assembly and Linking with GCC

In order to provide high-quality program compilation and maintain compatibility with Texas Instrument’s MSP430 compliant code, Alchemy uses MSP430-GCC project as its C language compiler. MSP430-GCC consists of a port of GNU GCC 3.2.2 that supports a broad range of processors of MSP430 device family. The compiler comes with several of header files for the processor and a basic libc library [29].

Like any traditional C compiler, MSP430-GCC uses several tools in at least four compilation stages which are pre-processing, compilation, assembly and linking. AGCC will provide an executable file at the end of the compilation process that will be patched by Archeus in order to embed Alchemy BRIDGE instructions in the program binary code. In fact, Alchemy compilation process uses the executable file to obtain an instruction dump file. Figure 5.7 shows the basic compilation flow of MSP430-GCC.

![Figure 5.7: MSP430-GCC compilation flow.](image)

CPP stands for C preprocessor and is the tool responsible to expand macros and include header files into C programs. It also deals with conditional compilation and line control over input source code files. CPP works over a already preprocessed C source (generated with Alkahest and Azoth).

GCC is the GNU C Compiler and it is in charge to compile the preprocessed source code. This stage consists in translate the C language source code into architecture-specific assembly language. MSP430-GCC will generate assembly code for MSP430 architecture and may also perform optimizations in the code, if requested by the user. The output of this stage is an assembly code.

The Assembler (AS) converts the assembly code into an object-file that holds the binary values for each instruction in the user program. AS will create the object-file with relative code offsets so that the Linker (LD) can produce the final executable file. It takes one or more object files or libraries as input and combines them to produce a single file. In doing so, it resolves
references to external symbols, assigns final addresses to procedures/functions and variables, and revises code and data to reflect new addresses in a process called relocation.

As a final compilation stage, Objdump is used to provide a disassembler view over the executable file the next build stage. Its output is a .dump file that holds the relative position of each instruction in the executable file as well as their binary code and corresponding assembly language representation. The binary .o file is ignored by Transmute.

5.2.6 Memory Patching with Archeus

The final step of compilation is the memory patching stage. At this point, Archeus is used to analyze the pre-compile dump file and convert it into actual software image. This stage is also characterized by instruction patching (nop and dadd #0, r3/r4/r5) which will be converted into actual BRIDGE instructions, according to their position in the binary file.

Archeus firstly parses the dump file generated by MSP430-GCC. The parses procedure consists of extracting the hexadecimal value of each instruction in the dump file, converting them to their binary representation and storing it in a sequential array of instructions.

The patching procedure is responsible to locate sequences of four nop and a dadd #0, r4/r5 instructions and converts them into BRIDGE instructions. The first four nop are replaced by four stb instructions with different target BRIDGE register. Archeus converts dadd #0, r4 or dadd #0, r5 instructions into par or seq instructions, respectively, forwarding all BRIDGE registers to the FPGA.

Whenever a dadd #0, r3 instruction is found in the code, it is replaced by a dst instruction. Note that a nop must be included in the word following the one of dadd #0, r3. Archeus will store the number of clock cycles used to schedule the Cross-Procedure abortion in the position kept by the nop.

After patching the instruction array, the software-image file (.smg files) is generated, holding the binary code of each instruction in the user program. With this last procedure, Alchemy applications can be directly deployed to the system simulator or to the hardware model.
Chapter 6

Architecture Implementation and Validation

Computer architecture exploration researches, like this project, tend to test proposed ideas and architectural approaches in several ways. Up to this point, this document has presented a description of Alchemy architecture features, and this chapter is focused on practical implementations.

Alchemy was initially developed as a high-level model that loosely outlined the CPU, FPGA, BRIDGE, Memory, conventions, protocols, tools and languages. Later on a lower-level architecture model was developed. Successive refinements were widely used to make sure each architecture element and their relationships are in match with the initial requirements.

Aside from that, in order to ensure that the models were correctly implemented, a set of validation techniques were used to test the correctness of the models. This Chapter presents a detail description of Alchemy architecture modules, as well as the validation procedures performed to ensure they were implemented accordingly.

6.1 System-C Transaction Level Model

Transaction-Level Models (TLM) tries to catch the architecture features by expressing them as software blocks that exchange information with transactions, rather than wires or channels. This modeling approach makes it feasible to translate requisites into actual executable code, once the implementation is guided by the relationship between the model elements instead of digital logic. In many situations, TLM models can express the architecture model as it was designed in the requisites documents, making it more readable and highly improving the design quality.

TLM was chosen to implement the first Alchemy model for several reasons. Firstly, it is a software-level model. That means that the model is implemented in a higher-level and hence some implementation details are left apart. These details are meant to be further refined in a lower-level model, that will closely resemble the final architecture hardware. Second, TLM models are faster in terms of design time and simulation time. In fact, the TLM model took about 2 months to be fully implemented and could run up to 300 thousand instructions in about 40 minutes of simulation. That makes the model (and the architecture) design much more iterative and less prone to error, once implementation decisions can be readily tested.

By designing a first model in TLM, Alchemy earns a high-level software model to be used
as an architecture simulator. This is very interesting for users once they can test their applications in a relatively accurate system, before actually deploying it to the final device. Another important advantage provided by TLM models comes into play during the implementation of lower-level models. After properly tested and validated, the TLM model can be used as a Golden Reference. That means that its results can be compared with the lower-level model in order to ensure its correctness.

6.1.1 Implementation

Once TLM takes part of System-C architecture, and System-C itself is a set of C++ language libraries, the Alchemy's TLM model is built upon a set of .h System-C modules, that are in essence C++ classes. These classes define most of the functional blocks in the architecture, as seen in Figure 3.1. There exist a single top-level module that is in charge to integrate these modules with sockets which will in turn exchange payload objects during runtime. A single .c file creates an instance of the top-level module and starts the simulation when the TLM model is started.

There are two clearly distinct implementation approaches in Alchemy TLM Model. The CPU is said to be implemented in a static way while the FPGA is said to be dynamically implemented. The static implementation was used to describe the steady CPU structure, in which most of its logic is explicitly declared in its compounding modules. Dynamic implementation takes a set of parameters such as MATRIX_WIDTH and MATRIX_LENGTH and builds the model during the system startup time. That means that the declaration, binding and the device configuration is performed while the module is actually running.

Static modules describe the CPU Control Unit, the BRIDGE, the ALU, the Register File, the Memory Management Unit, the three memory Resources (Shared, Program and Configuration), the Interrupt and Configuration Controller and the Instruction Decoder. The bus hierarchy is implicitly implemented by means of the modules transactions.

It can be said that the FPGA is the dynamic module. It declares and ties the CLBs, Interconnects, I/O Blocks and Switch Matrices with a set of specific C++ functions. This dynamic approach was useful to simplify the FPGA implementation once most of the module consisted of highly regular structures.

In terms of lower-granularity modules, the TLM model defines some general structures such as multiplexers, registers and memory blocks. Most of the elementary functionalities, such as logic and arithmetic operations come directly from the C++ language rather than from digital circuit primitives.

Most of the Control-driven modules such as the Configuration Controller, the Control Unit and part of the BRIDGE, are implemented using loop statements that provides a clock-like execution pace for the loosely-timed System-C + TLM module. The loop iterations can be considered as clock-cycles, without the notion of time attached to them.

6.2 Alchemy Register-Transfer Level Model

Based on the executable TLM model description, it was possible to derive a more detailed and hardware-driven implementation of the architecture. RTL models abstract digital circuit in terms of the exchange flow of signals among registers and the logical operations applied over these signals.

Despite describing the same functionalities as presented in the initial architecture specification documents, Alchemy RTL model tries to catch the processor characteristics by using a finer granularity on design partition as well as in the functionality implementation. The RTL
6.3. ARCHITECTURE VALIDATION

description code is built upon a collection of behavioral and structural VHDL modules that give an implementation that is very close to the actual device hardware. Furthermore, the VHDL code was designed thinking more in synthesis than in simulation. Therefore, most of the design short-cuts provided by VHDL libraries and high-level behavioral structures were not used in the final code.

VHDL was chosen as a design language for convenience. The models described in this level of abstraction are usually not suitable for simulations so that users shall not consider this for program debug purposes. Due to its high complexity VHDL models are slower than System-C models, which gives less efficient simulations and somehow cluttered simulation logs. In other hand, VHDL models can be used to directly synthesize digital hardware layouts.

6.2.1 Implementation

Hierarchically, the RTL model defines a top-level entity (MSXP4200 module) which in turn is composed of two smaller mid-level modules, MSP430 and XC2000. The former implements the CPU, the Program Memory, the Shared Memory and the Memory Management Unit. The latter implements the FPGA and the BRIDGE modules, alongside with the Shared Memory and the Configuration Controller.

In the CPU scope, MSP430 is formed by four main modules: Register_File, Instruction_Register, Control_Unit, Instruction_Decoder and Arithmetic_Logic_Unit. The Shared_Memory, Program_Memory and the MMU are merged into a larger Module called Memory. As for the inputs and outputs in the CPU module, clock and 83 inputs/outputs signals provide the communication of the CPU with the outside world.

The XC2000 scope defines the BRIDGE module, alongside the FPGA. This is an entirely structural module where all Interconnects, Switch Matrices, CLBs and I/O Blocks are declared and tied together. The Configuration_Controller and the Configuration_Memory are also included in the FPGA Module. Aside from the clock signal, XC2000 defines 85 inputs and outputs for communication with the rest of the architecture modules.

When it comes to building block modules, the RTL model defines some very low-level structures such as individual multiplexers, demultiplexers, Interconnects, tri-state buffers, BRIDGE registers, standard flip-flops and latches, along side some higher level structures such as full 8- and 16-bit registers, memory cells (and memory banks) and LUTs. Most of the finer grain modules such as wires and logic gates are directly implemented using VHDL language statements. These modules were generally integrated by using a structural approach.

Most of the behavioral modeling was applied to the control units and to the control modules. These modules were described using finite state machines to represent state persistence and transition. Most of them are synchronized by a global clock signal. Behavioral modeling was also widely used in the ALU module, which describes complex logic and mathematical operations over registers.

6.3 Architecture Validation

Validation comprehends a complex set of technique used to ensure that a given design satisfies some criteria of acceptance, by proving mathematical correctness. Therefore, validation techniques can verify the overall design exhaustively. In practice, models are usually validated by using logic simulation.

Alchemy was tested in a simulation-based environment considering two different validation techniques. One of them is the Exhaustive Verification to a bounded Depth in which a model is thoroughly verified with simulation. This can also be applied to two designs potentially in
different abstraction levels rather than a non-functional specification or an implementation. Dynamic Property Checking approaches determine if the design satisfies a given model specification in terms of its logic. Figure 6.1 shows the overall Validation procedure performed to test Alchemy.

In both verification approaches, output logs were used to translate the execution results into a human-readable piece of information. System-C CPU TLM model defines a cycle-based log structure in which each machine-cycle is divided into at least four parts (depending on the instruction addressing mode). Each of these parts describe the information flow through the CPU modules involved in the transaction. For the FPGA model, each clock-cycle consists of a log section, and each section presents the FPGA elements that had its value changed. The VHDL RTL model has the same logging structure, but waveforms were also widely used, specially during Dynamic Property Checking stage.

### 6.4 Dynamic Property Checking

One of the most used validation method is the Dynamic Property Checking, which consists of a model exploration by means of interactive simulation. In this context, a model is taken as a reference to be compared with an executable model, seeking for conformity. Alchemy mathematical formulation comes from the initial architecture description, which states what exactly each block is meant to do. One of the greatest advantage of Dynamic Property Checking is that it can often be performed automatically. This can be translated as testbench routines used to verify the models as they are being implemented.

Alchemy architecture modules (both in TLM and RTL models) were individually validated with custom testbenches. This simpler yet powerful verification mechanism was used to guarantee that the architecture features are functional and to make evident bugs in the module code. There was built about 70 different testbenches to verify the System-C model and about 50 testbenches for the VHDL model.

The architecture modules were slightly designed focusing on approaches such as design for testability once the tests performed during design time were driven by a white-box model. This makes it trivial to compare different representations to be used in Dynamic Property Checking approach. Furthermore, design for testability is a very complex subject which was not in the scope of this project. Henceforth, testability features were included in the module as they were needed.

Dynamic Property Checking was applied by using individual testbenches to test each part of the model, intending to verify if it satisfies the behavior established in the architecture definition state. If the satisfaction cannot be attained, it is not provided a counter example (as presented in [17]), but the model is changed to become consonant with the formulation.

During implementation stages, the testbenches were designed to verify a single module at a time. Despite providing local correctness, these testbenches were important to simplify further testing routines, during the integration stage. A common metric to evaluate the efficiency of a testbench is the statement code coverage. Alchemy modules were tested to reach nearly 100% statement coverage, which ensures that all code lines were checked during simulation. If the testbench cannot achieve such a full coverage, it is most likely to be ignoring some situations that may potentially hide bugs and unwanted exceptions. In practice this perfect coverage is not always achievable. The average coverage attained with the testbenches was about 81%.

Once each module has an unique internal architecture, it was not possible to use a single testbench to verify the whole processor. However, a basic template was designed to simplify the creation of complex testbenches during the implementation. This template is based on a
Figure 6.1: Alchemy Formal Verification Flow Diagram.
set of simple rules that describe what ideally a testbench should provide. These rules can be stated as following:

1. Testbenches must have as much inputs and outputs as are there in the modules they are meant to test.

2. All inputs and outputs must be used in the test routine, and they should ideally change between the acceptable range of possible values. The module architecture must be prepared to ignore or report wrong or out-of-range inputs. Likewise, the testbench must watch for erroneous outputs provided by the module.

3. The internal logic of the testbench should minimally mimic the expected logic of the real module that will interact with the module under test. The testbench will not provide all the functionalities of these modules, but minimal behavior is expected. For example, when testing the memory blocks, minimal timing must be ensured from the testbench routine, so that the module can be properly tested.

4. All parts of the module should be executed at least once during test, in order to achieve a 100% statement coverage.

As an example, the testbench implemented to check the functionality of a register bit, was designed to provide a clock signal, a set of valid inputs and to watch the register output after each clock-cycle. If the register value is different from the expected for each previous input, the testbench writes an error line in the simulation log, informing the inconsistency in the register operation. Likewise, when testing an FPGA interconnect, the testbench first configures the module’s tri-state buffers. After applying a set of inputs to the Interconnect, if an unexpected value appears in the output tri-state buffers, then it writes an error line in the log file. The interconnect configuration is also changed during the simulation, to ensure that all use cases were tested.

Testbenches should be designed to perform Fault Injection so that the module can be evaluated over unexpected or unwanted situations. The module is expected to provide some mechanism through which these kinds of events are handled. The testbench in turn, must test as much cases as possible, in order to drive the module with realistic test scenarios. As a Fault Injection example, the FPGA Interconnect testbench was designed to configure and drive two interconnect inputs at the same time. The interconnect must ideally ignore one of them and deny the configuration of two simultaneous inputs.

The testbench does not necessarily have to validate the module outputs. In case of simpler logic, output validation code was directly included in the testbench, so that wrong results could be directly reported by the test routine. However, complex validations were manually performed based on the output log generated by the simulation. This procedure intended to ensure that the module is actually performing the functionalities they were designed to.

Once the Architecture modules were individually tested and validated, the Dynamic Property Checking was further expanded to cover the integration stage, in which basic modules are grouped to implement larger functional blocks. At this point, new testbenches were designed to evaluate the interaction between the modules rather then their internal operation, which was already assumed to be correct.

Considering that not all parts of the module are executed when interacting with other modules, a 100% statement coverage test is not mandatory to declare the correctness of integrated modules. In addition, Dynamic Property Checking was still based on the initial definition of the architecture, so the test approach was not changed.

An integration testbench must firstly force the modules perform the desired interaction. For example, when the instruction decoder is to interact with the Register File, a set of configuration
6.5 Exhaustive Verification

Exhaustive Verification is the most common and simple approach on Formal Verification. This is an approach that uses simulation to assert a model correctness. This project uses simulation as an Exhaustive Verification method. It was defined a set of test-vectors used as inputs for the fully integrated models, that generate outputs to be compared with a reference model (or Golden-Reference). These vectors consist of programs and configurations, generated in both Assembly and High-level languages. They were randomly generated with an application developed for this purpose. The Exhaustive Verification technique was exhaustively applied in the processor code. This section presents a detailed description of this verification methods used to test the architecture.

This project implements two different architecture models, in different abstraction levels. Due to the singularities of each implementation approach, the equivalence was checked in parts. The whole verification processes is driven by a single assertion which states that "if the outputs of the two models are not equal for the same set of inputs, this implies that the models are not equivalent". In the case that this assertion fails, the architecture model is submitted to a checking processes in order to identify bugs that eventually differs it from the reference.

According to [18], to prove the equivalence between two designs having \( n \) inputs, the outputs are checked for \( 2^n \) input patterns, which is not practical for large combinational circuits in
terms of simulation time. Similarly, sequential logic is translated into a set of communicating FSM. Given the same input sequence, Exhaustive Verification checks whether two FSMs have the same behavior. In both cases simulation time is subject to combinatorial explosion, so a simpler and cheaper checking technique must be selected to allow a more effective design-space exploration.

This project scales the Exhaustive Verification processes down to a manageable set of tests performed in the both models, followed by a strict equality check between the simulation results. Each test is based on random input vectors that excite internal nodes of each model. The models are said to be equivalent if all similar nodes are equivalent. In order to reduce the search space, the input vectors are limited to the instruction set (in CPU testing) or three possible configurations in the FPGA. These configurations are the basic LUT setups, associated with random routes in the FPGA Matrix.

When simulating Alchemy, either RTL or TLM models were subject to a design partition prior to the validation processes. There was defined three different test groups. The first one consists of CPU tests, the second consists of FPGA tests and the third consists of Architecture Tests. CPU tests are driven by instruction sequences that were randomly generated from high-level C program description and compiled with Transmute Toolchain. Similarly, FPGA tests had their input vectors built with random configurations derived from a high-level Verilog module, synthesized with Transmute Toolchain. Lastly Architecture tests were based on a combination of the two previous approaches.

Once the first implemented architecture model was the TLM model, it was checked against a Golden-Reference Model. For CPU tests, the Golden-Reference was the MSP430 hardware (MSP-EXP430G2 integrated in MSP430 LaunchPad Value Line Development kit). For FPGA tests, the verification process was limited to Verilog simulations running in Altera Modelsim for Quartus II. The third test group was only performed in later development stages when the RTL model was properly verified.

Following the model hierarchy, the lower-level model described in RTL was checked against the TLM model. This was possible once the RTL model was only implemented when the TLM model was fully implemented and verified. In this case, the TLM model is said to be the Golden-Reference for the RTL model. CPU tests and FPGA tests were performed based on the same test vectors used to test the TLM models. This allowed the reduction of tests setup time and did not require another round of simulations with the TLM model.

In order to ensure that both models have implemented the same conventions for cross-domain operations with the BRIDGE, random applications were used as input for the third group of tests. These applications were limited to simple interaction procedures with the custom BRIDGE instructions and Return Interrupts. Exhaustive Verification was useful in this stage, once both implementations could be considered correct if their results are equivalent, even though these tests lacked a Golden-Reference Model. Figure 6.2 shows the basic Exhaustive Verification Test setup developed for Alchemy.

In the figure, an unidirectional arrow means "is a reference to" while a bidirectional arrow means "are reference to one another". The dashed rectangles informs that the same set of inputs are used among the internal test groups. Note that the Golden-References are assumed to be right in any situation. Even though an unexpected behavior is observed in these references, they are meant to exist in the TLM and RTL models. The next sections present the random test vector generation and the Exhaustive Verification procedure for each architecture model.
6.5. EXHAUSTIVE VERIFICATION

6.5.1 Test Vector Generation

Test vectors are the inputs for Exhaustive Verification validation. Each test vector is ultimately a sequence of random bits that have two attributes: a length and a random seed. The length informs how many instructions or bits are in the vector. The length can be interpreted in different ways depending on the vector purpose. The random seed is an unique attribute that is used to initialize a pseudorandom number generator, which is the base to emit bits in the vector. This is a fundamental attribute once it allows one to re-build any test vector, keeping a coherent history of generated tests.

When used to test the CPU, test vectors were composed of random assembly instructions with 16 bits each. The length of a CPU test vector is assumed to be divided by 16. CPU test vectors can have up to 65536 instructions (64 KB) that are stored in the program memory. The test vector is initially zeroed and the instructions are stored in any location, as needed. CPU test vectors are generated by compiling a random program with Transmute Toolchain.

Unlike CPU test vectors, FPGA test vectors are sequences of bits of a predefined length. That means that, regardless the configuration to be stored in the FPGA, the vector will always have the exact number of bits configured on it by the synthesizer. Rather than storing instructions, these test vectors have a configuration bitstream. When the FPGA was being implemented, the BRIDGE had a special debug mode in which its registers could be used to directly provide and store content from the FPGA I/O Blocks, without the needing of a CPU attached to it. The Synthesizer was also in charge to generate this minimal bitstream to configure the BRIDGE Selection Plane and Register Bits. The configuration bitstream have the fixed size of approximately 22 KB. FPGA test vectors are generated by synthesizing a random module with Transmute Toolchain.

In both cases test vectors are generated with a custom application called Aether. Aether provides a friendly graphic user interface in which the test vector parameters can be tuned. Aether generates programs or modules in either Assembly or high-level languages. These programs/modules should in turn be built using Transmute Toolchain. The software/hardware images outputted by Transmute will then become the test vectors used in Exhaustive Verification.
For CPU test vectors, it allows users to select which instructions (in Program Assembly level) or which statements (in C) are going to be part of the test vector. Users can also determine how many instructions (or statements) are included in the vector. For Program Assembly program, Aether allows the selection of which addressing modes will be used with the instructions, which operation mode (byte or word) and how many registers will be made available. The number of subroutine calls and their sizes are also selectable parameters.

For C Programs, Aether provides a wide range of selectable parameters to control the vector generation. These are the number of statements, the number of structs and constants, the number of arrays, the number of variables, how many subroutines are declared and how many will be used through the code, which are the arithmetic and logic operands, how many loops and conditional statements, the maximum nesting level and the number of elements in the right side of an expression.

For FPGA test vectors, Aether provides different parameters for Module Assembly and Verilog. When generating the vector in Module Assembly, users have the option to specify how many logic-blocks are used, which is the maximum length of the routes and which LUT modes are going to be used. It is also possible to determine the number of CLBs that will make use of its internal register, how many inputs and how many outputs and which sides of the FPGA Matrix they will be. Furthermore, users can specify which kind of random logic function will be configured in the LUTs (one can choose to fix some bits and allow other to freely vary).

When using Verilog as the source to generate the test vector, Aether allows users to define how many I/O pins will be made available and which will be their maximum length (in the case of signal clusters). It is also possible to determine how many internal signals are declared in the module (and their maximum length), how many wires and registers (and their maximum length) and how many blocking assignment statements. In terms of assignments, users can choose to have a maximum number of elements in the right side of the expressing, this includes both inputs and signals, as well as numeric constants. The left side can be randomly chosen between outputs or signals. A list of which mathematical and logical operands will be taking part in both conditional and logical statements can also be customized.

Aether provides a limited set of pre-defined structures for sequential logic generation. It allows user to include state machines in their code, with a maximum number of states. This is implemented with an always block and each state always transits to the next in a cyclic manner. Aether also allows users to specify how many if statements will be included in the always block (alongside the number of operands in the condition) and how many non-blocking assignments (and the type and number of elements to compose the assignment expression). Repeat statements are not allowed to be used. Multiple hierarchy between the modules is also not generated by Aether once it does not provide any important runtime implication in the test vector. The random modules generated with Aether must later be Synthesized with Transmute toolchain in order to be configured in the architecture model as the test vector. Aether user interface is presented in Figure 6.3.

Aether was designed to allow users to easily manage the content of their test vector. The application allows users to create multiple vectors and provides an unique random seed to each one. In fact CPU test vectors and FPGA test vectors can share a same seed once they provide completely different test vectors. The seed generation is controlled internally by Aether so the user does not have to worry about their consistency across multiple vector generations.

Aether was an important step towards test automation once custom designed C programs and Verilog modules, alongside with their Assembly counterparts, can take much time to be made by humans. Aether allows a better exploration of the design and test space once its random combination of statements is not biased by any application domain or specific goals. The programs and modules do not necessarily have to make any sense, but rather are meant to
6.5. EXHAUSTIVE VERIFICATION

113

Figure 6.3: Aether user interface.

excite the CPU and the FPGA as much as possible, by combining statements and operations in different ways. Aether was used in each one of the million test cases used on Alchemy Exhaustive Verification.

6.5.2 Exhaustive Verification in CPU and FPGA

Exhaustive Verification is performed by comparing different models outputs, for the same set of inputs test vectors produced with Aether. The Exhaustive Verification took place in two different development stages. The first one was after the Dynamic Property Checking validation with the TLM Model. At this point, the Golden-Reference Model for CPU tests was the MSP-EXP430G2 device, while the Golden-Reference Model for FPGA tests was Altera’s Modelsim. The second Exhaustive Verification was performed right after the implementation of the RTL model. In this case the CPU and FPGA Golden-Reference model was the TLM model.

For CPU tests, the procedure was performed as following: For each test vector, stored in the CPU Program Memory, run the program until the OFF bit in the Status Register is set to 1. This means that the CPU test will be executed until the CPU is shut down. The testing procedure does not state any duration, so programs can potentially run for an undefined period of time. Any test can be aborted during execution so that infinite loops (for example) can be stopped. Note that even though endless loops are executed, the test results can still be used.

FPGA tests in turn, were performed considering a maximum test limit. In order to keep the duration independent from clock frequencies, tests duration was measured in clock-cycles rather than in seconds. These tests were executed as following: For each test vector stored in the FPGA Configuration Memory, provide a set of dynamic random inputs to the FPGA I/O Blocks, allowing them change freely through clock-cycles. Like CPU tests, FPGA tests can be aborted at any time. The simulation results can still be used in this case.

Exhaustive Verification is a very iterative processes, as implemented to test Alchemy models. It requires the models to strictly perform each operation stored in their memory blocks so that their behavior can be evaluated. In both cases, the test results were log files with hundreds of thousands of annotations regarding the state of the CPU and FPGA internal blocks. Test
files had usually a size of a few hundreds of mega bytes to a few giga bytes.

For CPU test results, the data composing a log file was the following: Register File content, the current instruction information (extracted from Instruction Decoder), the ALU output values and the Shared Memory state. It was not stored any data regarding Control Unit signals or other minor CPU blocks once the Register File, ALU and Memory are central architecture elements. That means that they can only work properly if the other modules are operating correctly and hence they are enough to state the CPU correctness.

For FPGA test results, the information written in the log file was the state of each CLB inputs and outputs, and register content (only for RTL model) and I/O Blocks contents. Note that only inputs and outputs were extracted from the Modelsim simulation. Routing elements did not take part on the log file once the correct operation of CLBs and I/O blocks is directly dependent on the Interconnects and Switch Matrices. Henceforth they are not fundamental to attest correctness of the FPGA model.

Once the Exhaustive Verification of both TLM and RTL models consisted of the same procedures, only a general description will be given. After running the test vectors in the Reference Models and the Architecture Models to be verified (TLM or RTL), for each test vector, there will be outputted two log files. These files are in turn compared with one another, in order to identify divergences on the checking process.

An automatic comparison tool has been developed to simplify Exhaustive Verification. Azure is both a log file comparator and a graphic user interface that displays the test results, highlighting divergences in the verification process. It accepts both CPU log files as well as FPGA log files. Azure user interface is presented in Figure 6.4.

For each test vector, Azure takes two input .deb files, one from the Reference and other from the Architecture Model under verification. Azure will first perform a pairwise comparison, checking each runtime information per clock- or machine-cycle. Whenever a divergence is found, Azure annotates in a third log file (.cmp file) the iteration where the divergence was observed and which modules are not in match. This procedure is executed until there are no more comparisons to be performed.

After that, Azure comparison can be analyzed with a GUI which displays the results for
6.6. VALIDATION CONSIDERATIONS

Each comparison, highlighting the differences alongside their sources. Users can easily navigate through divergences, so that the debugging processes becomes more focused.

Azure was vital to lead the correction of non-obvious bugs in the Architecture models, once for each divergence observed in the log files, it was possible to correct the problem source (in the model source code) and re-run the test with the same test vector. This iterative fix-and-test cycle was fundamental to correct most of the bugs not observed during Dynamic Property Checking stage.

Exhaustive Verification stage could cover about 86.2% of the source code statements of the TLM model and about 89% of the source code statements of the RTL model. These coverage results were considered to be satisfactory for this purpose, considering the scope constraints and time limitations imposed over this project. For CPU tests, Azure performed about 14 million machine cycle comparisons. For FPGA tests 36 million clock cycles were tested.

6.5.3 Exhaustive Verification in BRIDGE

When validating the BRIDGE (integrated with the CPU, FPGA, Memory System and Busses) TLM and RTL models were checked by comparing one to another. This can be considered a final integration stage in which the whole processor is validated to confirm the equivalency of the two models, as well as to ensure that the Architecture concept is functional.

For this validation stage, random Alchemy Applications were built with Aether. The main goal of these applications was to test both Cross-Procedure Prologue and Epilogue. The two models were prone to run the same test vectors so their outputs can be once more compared with Azure. Note that this time, each model outputs two log files, one for the FPGA and other for the CPU. The BRIDGE is not directly observed in this test once it would require the adjustments in Azure and Aether codes and because the BRIDGE behavior directly affects the CPU and FPGA execution (any error in the BRIDGE would cause wrong results in the comparisons performed by Azure).

Once it was needed to test small subset of CPU instructions (5 BRIDGE instructions) and the Return Interrupt, there was performed about 5 million machine cycle comparisons (in the CPU) and 9 million clock cycles (in the FPGA). These tests were important to locate bugs in the BRIDGE code and to test the overall protocol functionalities.

6.6 Validation Considerations

Based on the presented two-layer model validation, the architecture software simulator (TLM model) and the hardware model (RTL model) could be successfully verified. Despite not reaching a 100% code statement coverage during simulation, the amount of test vectors and the number of bugs corrected through the validation process proved that the proposed Validation approach was satisfactory to ensure correctness in the two models.
Chapter 7

Architecture Evaluation

In order to assert Alchemy’s computation power and advantages over other embedded systems processor, this Chapter presents a set of benchmarking tests performed with the architecture. This is fundamental to characterize the effectiveness of Alchemy’s features and their real effect in the overall architecture. This Chapter also presents a single application example, with implementation details. It is intended to provided a general view over the steps required in the design process of Alchemy applications. This is an engineering exercise for those who want to create applications for reconfigurable processors. A simple problem is described, followed by a possible solution using Alchemy capabilities.

7.1 Architecture Benchmarking Tests

To demonstrate the capabilities of Alchemy architecture, a set of tests were designed and executed with the architecture models. These are benchmark-like tests performed in order to evaluate the architecture in terms of the number of clock cycles, memory accesses and BRIDGE protocol latency required to execute certain classes of algorithms. It was intended to determine if the payload inherent to the BRIDGE protocol and the FPGA architectures were in match with the provided performance improvements reportedly observed.

The basic test setup consists on the executions of programs, for two possible scenarios: one using the CPU and the FPGA (Setup 1) and another using only the CPU (Setup 2). The tests were designed to expose the effect of partitioning the workload between the CPU and the FPGA and to determine how the BRIDGE communication protocol can affect the system overall performance. Each test setup was executed with a 25 Mhz clock frequency.

Five programs were selected: Program A: 5,000 8-bit integer multiplication executed within a loop; Program B: 5,000 random 16-bit integer comparisons; Program C: a 50x50 gray-scaled 8-bit image filtering (using a single sample Sobel matrix); Program D: an even parity computation algorithm executed 5,000 times; Program E: an FIR filter applied over 12,000 16-bit integers. The data to be analyzed was selected so that no technology specific parameters could bias the results. It was analyzed the number of clock cycle taken during execution (T1 for Setup 1 and T2 for Setup 2), the number memory accesses performed and the bridge-protocol latency. Since the FPGA configuration is part of the system prologue, it was not accounted in the final results. Likewise, the CPU prologue for stack pointer setup, registers setup, interrupt controller setup among other procedures was not accounted. The system shutdown epilogue was also not accounted for both setups and no scheduled FPGA calls were performed.
Table 7.1 shows the tests results. For program A, Setup 1 could perform considerably better than Setup 2 (about 13-times faster). It can be observed that Setup 2 faced larger overhead due to memory accesses. It can be observed that Setup 1 faced a large overhead executing the bridge protocol (87.5% of its execution time).

For program B, Setup 2 outperformed Setup 1 in about 70%. This was due to the overhead caused by the FPGA calls, which required several immediate BRIDGE register manipulation and causes the overall execution to be more expensive than the simple 'compare' instruction. Setup 1 spent about 91% of its time in protocol execution rather than performing the comparisons. For program C, Setup 1 surpassed Setup 2 performance by a factor of approximately 6. Differently than other algorithms, only an initial BRIDGE protocol overhead is imposed to the FPGA. The memory accesses were directly handled by the FPGA-BRIDGE and no actual protocol overhead is required. Henceforth, the majority of FPGA execution time was demanded by actual computations. Setup 2 had to execute massive memory access to get the pixel values (a set of eight MOV instructions for each pixel).

For program D, Setup 1 performed 6.1 times faster than Setup 2. It is observed that an even parity circuit can explore parallelism in the FPGA, making the computation faster. The main drawback in Setup 2 implementation was the large number of logical operations that requires a large number of individual memory accesses. Only 7% of the FPGA execution time is used for protocol handling. Program E was implemented in Setup 1 using multiple calls rather than direct memory access. This implementation guideline was taken in order to explore different FPGA usages. Setup 1 could perform slightly better than Setup 2, being about 2.5 faster. The main reason for the discrete performance improvement is due to the expensive context switches between subsequent FPGA calls.

In all cases, Alchemy architecture could outperform standard single core embedded processor setup by 2 to 12 times. This performance improvement is important for two main reasons. First, by computing solutions faster, the overall system requires less power to run. According to [13], a processor that is more power-hungry but takes less time may use less energy. For systems which rely on battery, as their primary power source, Alchemy is a better solution when compared with standard CPU cores, even though its inherent parallelism requires more power per work unit.

Furthermore, the performance improvement could be achieved even though each evaluation test implement different functionalities. By considering ASIC platforms, this kind of improvement would be nearly impossible: the architectures would not be adaptable for different applications and hence new ASIC design would be required to fit the needs of each application. Alchemy in other hand, provides adaptability to users, which can explore reconfigurability to

<table>
<thead>
<tr>
<th>Program</th>
<th>Clock Cycles ($10^5$)</th>
<th>Memory Accesses ($10^5$)</th>
<th>Bridge Latency ($10^5$)</th>
<th>Clock Cycles ($10^5$)</th>
<th>Memory Accesses ($10^5$)</th>
<th>Speed-Up ($T_2/T_1$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1.21</td>
<td>0.21</td>
<td>1.05</td>
<td>15.05</td>
<td>57.01</td>
<td>12.43</td>
</tr>
<tr>
<td>B</td>
<td>1.55</td>
<td>0.15</td>
<td>1.40</td>
<td>0.89</td>
<td>0.11</td>
<td>0.57</td>
</tr>
<tr>
<td>C</td>
<td>5379.00</td>
<td>1375.00</td>
<td>0.10</td>
<td>34447.50</td>
<td>10976.50</td>
<td>6.41</td>
</tr>
<tr>
<td>D</td>
<td>1.42</td>
<td>0.39</td>
<td>0.11</td>
<td>8.65</td>
<td>3.30</td>
<td>6.09</td>
</tr>
<tr>
<td>E</td>
<td>0.61</td>
<td>0.14</td>
<td>0.12</td>
<td>1.56</td>
<td>0.19</td>
<td>2.55</td>
</tr>
</tbody>
</table>
improve performance of their applications without recurring to custom and expensive circuitry.

After presenting the advantages of Alchemy in terms of performance improvement, it is intended to provide a general view over the steps required in the design process of applications. The following section presents an engineering exercise for those who want to create applications for reconfigurable processors. A simple problem is described, followed by a possible solution using Alchemy capabilities.

7.2 Alchemy and Wireless Sensor Network

Embedded systems and specially Texas Instruments MSP430 micro-controllers, are widely used in an application filed called Wireless Sensor Networks (WSN). These networks are usually composed of a set of individual nodes that are able to interact with the environment by sensing or controlling it. As a mandatory requirement, wireless communication enables the operation of the nodes to fulfill bigger tasks that a single node could not.

Wireless Sensor Networks are used in several applications such as disaster relief operations, wildfire detections, biodiversity mapping, Intelligent buildings and bridges, precision agriculture, medicine and health care, logistics and many others.

In the general topology of a WSN, there exist two classes of participants. Sources are nodes that sense data in the environment. These Sensors must mandatorily report their status to a single node called Sink. The Sink or Base Station is in charge to receive data from Source nodes. It can be part of the sensor network or possibly can be a heterogeneous device such as a laptop or mobile phone. Sources periodically perform their measurements in the environment and detect events that must be reported back to the Sinks.

One of the most important tasks of Source nodes are the measurements. This procedure consists of constantly measuring an environment characteristic, with an analog sensor, and performing filtering operations on the digitized sensor sample. From a processor (or microcontroller) standpoint, data is constantly made available in memory by the sensor/Analog-to-Digital Converter. Its task is to read this data, process it and send the results to the Sink.

Considering the low-performance requirement imposed by Source nodes (due to its lack of battery resources), computations usually are very simple. Henceforth data processing must be handed to the Sink which can easily become overwhelmed by the amount of requisitions from several Source nodes. Reconfigurable Architectures can be used to perform an efficient preprocessing in the acquired data, relieving the Sink from heavy workload.

In order to demonstrate Alchemy capabilities, a sample Source Node was built. In this scenario, the Source node gets inputs from an A/D Converter (which is in turn receiving data from a sensor) and pass an FIR filter in the inputs. This results in a low-pass filtering operation which removes high-frequency noise from the sampled data. The FIR topology implemented for this purpose was a 3-term moving average filter in the form:

\[
y[n] = \sum_{k=M_2}^{M_2} b_k x[n - k] = \frac{1}{3} (x[n] + x[n - 1] + x[n - 2])
\] (7.1)

Once this operation can be expensive to be performed in low-end architectures, a coprocessor can be used to improve the overall performance of the system. In this example Alchemy FPGA was used to implement an application-specific instruction which implements the FIR filter while the CPU is in charge to interact with the AD converter and store the sampled entries in the memory, in groups of 1024 samples. The FPGA applies the filter to the sampled array, storing the results back in the memory.
The workload partition between the CPU and the FPGA was designed as following: the CPU must initialize the ADC converter and then request it for new samples at a constant pace. This processes are repeated for 1024 samples until the FPGA custom instruction is invoked by the CPU.

The FPGA assumes that the 1024 memory words, starting at 0x3FF0, hold the set of sampled values from the ADC. It will apply the FIR filter presented in Equation 8.1, storing the results back in the same addresses they were loaded. No input parameters are required and the FPGA issues a Return Interrupt as soon as the filtering operation is finished. Appendix A1 shows the implementation of the C Program and the Verilog Module.

All tests are supposed to be performed with Alchemy's software simulator (System-C TLM Model). For that purpose, an ADC model was designed to digitalize samples over an uniform sinusoidal input signal, internally generated by the module itself. This ADC module was designed to resemble the actual functionality of MSP430G2253 10-bit ADC peripheral.

To maintain further compatibility, common MSP430 initialization procedures were also included in the C program. First off, the watch-dog peripheral is disabled. The frequency range of the internal clock is changed (without practical use in Alchemy once only a single clock source is available) and the ADC is set to work in I/O pin P1.3. Finally configure_adc() sets the ADC channels up and additional configuration bits determine the sampling-and-hold period.

__enable_interrupt() function enables general interrupts to be triggered anytime during program execution. This is another common step on MSP430 programs and can be required depending on the ADC functionality. For this specific example, FPGA Return interrupts are issued even though interrupts are not enabled, once it is a non-maskable interrupt.

The number of samples is stored in sample_number variable. The ADC conversions are read in an endless while loop which always begin with a delay of 1000 cycles, which will allow the ADC reference to settle. After this initial idle period, the CPU request a conversion to be performed and stores the outcome into samples array. If 1024 samples are stored in the array, the FPGA is called to filter the data using its FIR implementation. As soon as the Return interrupt is issued, the number of samples is cleared and the conversion are once again started.

The FPGA module is widely driven by state-machine to control the memory interaction through the BRIDGE. There exist two distinct states in the FSM, which are used to load the word from memory, filter it and then store the results back to the memory. The filtering is implemented as presented in Equation 8.1 and the division by three is computed by shifting and addition operations. The FPGA will continue this procedure for 1024 iterations until a Return interrupt is issued to the CPU and the conversion takes place.

The application described in figure was built using Transmute Toolchain, targeted for the Software simulator. The C Program was preprocessed by Alkahest and Archeus, resulting in a 1.2 KB memory file. Regardless the intended configuration, the Verilog module is converted into a 23.5 KB bitstream, taking 89% of the logic resources (CLB) tied by 272 routes and using 64 I/O Blocks of the FPGA. Appendix A2 shows the resource allocation in the FPGA, as outputted by Azoth (.pack log file).

The program was executed considering an ADC internal clock reference of 1/5 of the CPU clock (the ADC module is updated once for each five iterations of the CPU/FPGA modules). A similar version of the program was designed to run only in the CPU without co-processing support, which resulted in a slightly larger program.

The performance improvement provided by Alchemy features was of 2.6 times, which is moderate compared to other applications. It is possible to point out two main reasons for that. First, a large part of the nearly 12000 clock-cycles executed during the simulation were used to handle memory through the BRIDGE (10522 cycles). This was expected once the
FPGA module is designed to interact with memory in all its stages. The filtering operation is spatially computed during the ‘accessing stages’ efficiently using the idle time during memory interaction.

The second reason for the lower performance was the lack of parallelism in the interaction between the CPU and the FPGA. Once the FPGA module is triggered to run sequentially to the CPU, it must stall until the computation is not complete. Parallelism could be explored to allow the FPGA to filter the samples as the CPU store them memory, but Alchemy’s current memory architecture would cause many blocking requests due to concurrent memory accesses, which in turn could reduce the system performance.

Unquestionably, Wireless Sensor Networks could take advantage of Alchemy architecture, due to the faster processing and the ease of implementation. Despite not presented in Figure ??, the C program is meant to send the results to the Sink, which would incur in larger payload in the source code. This procedure could be performed in parallel to the actual filtering, further improving the system overall performance.

It is important to make a few considerations regarding the design process of the application. First off, Transmute toolchain proves itself valuable in design time, once the implementation was carried out with high-level languages such as C and Verilog. Assembly development would be too much complex and time consuming for entry-level designers, which would make the architecture less attractive for mainstream use.

The debugging process with the simulator introduces an unnecessary complexity to designers which must recur to extensive logs and a somewhat simple offline debugger tool to ensure that their programs and modules are correctly implemented. These must be considered possible improvements to the architecture toolchain.

The workload partition between the CPU and the FPGA was easily designed. It is clearly necessary to minimally understand Alchemy internal architecture though. Most of the embedded systems programmers usually have knowledge on computer architectures and internal micro-controller hardware, but this could discourage high-level language programmers due to the overwhelming amount of details regarding the internal functionality of Alchemy.

The hardware development stage is another example of the selectiveness imposed by reconfigurable architectures over programmers. Not all programmers know how to deal with implicit parallelism which would render Verilog modules nearly underused. The architecture tries to keep this design approach for two reasons. The first one is due to the finer control granularity over the logic resource, and second due to the complexity related to the creation of C-to-hardware compilers.

It is considered however, that the performance improvements provided by the architecture for such kind of applications is attractive enough to overcome the implementation challenges.
Chapter 8

Conclusions

This project presents a novel low-end embedded system reconfigurable architecture which integrates a CPU and an FPGA. The Architecture was carefully designed so that it becomes simpler and easier for users to develop and debug applications.

Alchemy was designed considering two different implementation approaches, one from a higher level (using SystemC and TLM) and another from a lower level (RTL with VHDL). These models were fundamental to test and ensure that the concept behind the architecture is fully functional and is affordable when compared to other embedded system architectures.

Alongside the architecture definition and models, a set of tools and programming concepts were created to make the development feasible and meaningful for users: a set of high-level programming and description language statements were developed, as well a new concurrency model, called flow, and a new procedure-like abstraction called cross-procedure.

The architecture was also thoroughly tested and verified to ensure that its models were correctly implemented and to assert if its features are efficient enough to provide performance improvements for user applications.

The following section discuss the Architecture and its consequences in application development and performance. A set of future works are also discussed in the final section.

8.1 Discussion

Alchemy has been designed through a ten month cycle which comprises the initial architecture planning down to the implementation of TLM and RTL models and the Transmute Toolchain. During this development period, the architecture was redesigned and tested through rigorous procedures, which provided a deep understanding of its features, both advantages and fallacies. This Chapter presents a summary of these features in a critical way, followed by a conclusion about the architecture.

Alchemy was initially designed thinking of low-end embedded systems, in which performance is already deeply harvested in coding optimization and ASIC design. These devices are usually highly specified for a given application and their performance is dependent on these scope. They are likely to perform poorly out of their application domain, simply because their hardware was not designed for this purpose.

Embedded software optimization has been proven as a powerful approach to improve performance in embedded systems. Algorithmic optimizations are capable of considerably improve performance of programs in many orders of magnitude, but are already well studied and many of them are directly performed by the compiler, even without the direct intervention of the
CHAPTER 8. CONCLUSIONS

user. Due to that reason, it becomes difficult to design new and better optimizations that bring the performance improvement to other level.

Alchemy was designed with this problem in mind. By providing users the ability to craft their hardware for a given application, both software and hardware can be optimized and further improvements can be achieved. Users are not limited by the machine instruction set nor the platform chipset, and their architectures can be highly specialized towards a given problem. The platform is also highly flexible: new software and hardware can be designed and uploaded to the device, providing an unlimited number of implementation possibilities.

Very expressive performance improvements could be achieved using this architecture, and they are important for two reasons: first, engineering cost can be highly reduced given that the design of new software and hardware takes as much resources as the design of an ordinary software. Second, low-end embedded systems usually lack of widely parallel systems such as GPUs, and by adding the FPGA to the architecture, an interesting SIMD and MIMD processing layer is brought to them. This makes performance gains much larger than simple software optimizations.

Despite providing advantages to low-end embedded systems, it is important to discuss some side effects caused by the architecture. First, the design effort is larger than simple software design flow. First, the software must be designed with the hardware and the communication mechanism in mind. Second, the hardware must be fully planned and designed, and further simulation and validation for both software and hardware must be performed to ensure correctness. It is expected that users have some background knowledge in software and hardware development.

Another side effect comes from the architecture design. In some Verilog Modules, memory throughput can easily become a problem due to the high number of access requests. This causes the overall system to stall until the memory interface deals with request. This can easily be solved with faster and dual-port memory, and some sort of caching system. These solutions were not included in the system due to the overhead involved in their development (for cache systems), in both area and energy consumption.

The BRIDGE itself may also cause side effects in the system performance. The communication protocol centralizes too much tasks on it, making the controller impose some kind of overhead, specially when acting as a memory proxy.

Another side effect caused by the architecture design decisions is the size of the FPGA matrix. It was observed that due to the limited amount of logic resources in the Matrix, sometimes it is required to redesign whole modules so that they fit in the FPGA. In other cases, highly complex modules would not even fit. This, however is not a major problem for the architecture once the models were designed thinking of FPGA matrix scalability and the current technologies make it feasible to implement larger FPGAs along with the CPU in the same die.

When developing solutions with Alchemy, users must keep in mind that performance will only be improved when two problems are correctly managed by their code. First, the workload partition must both evenly divide the work between the FPGA and the CPU and must be designed to deliver the correct tasks for each one of them. CPUs are control-driven blocks and hence are more likely to perform well in these kind of tasks. Conversely, the FPGA is a data-driven device which must be in charge of these tasks.

The second problem is the payload imposed by the architecture internals. In simple terms users must be aware that the solution provided by a CPU and FPGA design must take less or the same time of a solution using only the CPU or only the FPGA to pay off. Otherwise there is no point on using the architecture as so and the system will loose performance instead of improving it.
In terms of design tools, transmute toolchain proved itself worthy of implementing, once the complex device setup is completely abstracted for designers. The tool makes it simple to control the interface between the CPU and the FPGA with indirect commands, which results in simplicity and efficiency in implementation.

Alchemy was initially focused on raw performance improvements for low-end embedded systems, by proposing both non-conventional hardware and software setups. Despite the difficulties imposed by its nature, Alchemy can be considered a powerful tool to make embedded systems more efficient. Users must carefully design their solutions keeping the architecture features and payloads in mind. If they can overcome this challenge, it is most likely that Alchemy will positively improve their system performance and also may impact future projects and solutions.

Considering the recent products and researches in this area, it seems that reconfigurable architectures are to become mainstream as soon as the design tools are in match with the hardware complexity and user requirements. Alchemy is another attempt to make reconfigurable architectures mainstream. It can be said that Alchemy achieved its objectives as well as this project does. Future improvements in the architecture are discussed in the next section.

8.2 Future Works

This section presents a set of ideas for future works to further improve the Alchemy architecture.

8.2.1 Improve FPGA Matrix Size

In order to allow users to implement more complex and powerful hardware accelerators in the FPGA, an obvious improvement would be the size of the FPGA Matrix. 16 by 16 matrix can be further expanded to power-of-two sizes like 32 by 32, 64 by 64 or 128 by 128 matrices. The surrounding I/O Blocks could be accessed in sections of 16-bit per time and new BRIDGE registers would have to be included in the BRIDGE.

Instructions such as stb and ld b could be adapted to support a larger BRIDGE Register Bank, which would impose some overhead to be fully loaded by the CPU during run-time. In 128 by 128 matrices, for example, 8 ld b or st b instructions would be required to handle a single side.

8.2.2 CPU Instruction Override

By expanding the BRIDGE Protocol, it could be implemented an Override Controller which controls user-defined instruction overrides. The user would be allowed to specify newer versions of CPU instructions, by informing the override intention to the CPU and proving a new implementation in the FPGA. The Bridge would be in charge to receive and decode the overridden instruction fields, forward the proper information to the FPGA and trackback the results and status to the CPU.

The CPU Control Unit would have a set of override registers (27 registers) which store the status of each instruction. An instruction set as over would trigger a forward signal in the Instruction Decoder whenever read from memory. The full instruction is then sent to the BRIDGE, which decode its fields a send them to the FPGA. Only the fields defined in standard instructions would be allowed and the FPGA would have to implement special input signals for each field. The CPU would wait in a new FPGA-Execution state constantly checking an Acknowledge signal from the BRIDGE, until the execution result is sent back to the Register File. The execution flow would continue normally after that.
The Verilog non-standard subset would have to be changed so that new control signals are made available. Furthermore, the configuration process would have to properly inform the CPU which instructions are overridden. This allows higher code compatibility and provides more flexibility to designers. The co-processor concept devised to the FPGA is further expanded once Instruction Override capabilities makes the FPGA directly interact with the CPU organization. Users must carefully consider this future feature once its usage can considerably affect program execution.

8.2.3 FPGA in Machine-Cycle

By adapting the Control Unit architecture, it would be possible to allow the FPGA to execute once at each machine-cycle. The FPGA Incursion mechanism would allow the FPGA inputs, outputs and clock signal to change at predefined and regular times, reserving for that, a transition state in the CPU. This blends the FPGA operations with those of the CPU, making it possible to improve the overall system functionalities.

8.2.4 BRIDGE Watch Functionality

The Bridge could use its Monitoring unit to watch for specific I/O Blocks in the FPGA so that interrupts could be issued in the CPU. That could be implemented by creating a reserved keyword in the Verilog module (@sensitive <signal_name> value) in front of the signal name. This could trigger a special configuring stage in Azoth which (based on a sensitivity list file extracted with Alkahest) would be used to configure multiplexers to point to I/O Blocks. If the signal changes to (1 or 0) depending on the configuration provided by the user, a specific user defined interrupt could be used. The @sensitive also could inform the desired value to trigger the interrupt, in case of signal clusters.
Chapter 9

References


127


Appendix

```
#include “Alchemy.v”
#include stdlib.h
#define DATA_ARRAY_BASE 0x3FF0
#define _ARRAY_SIZE_ 1024
void configure_adc ();

int main ()
{
  int * samples = malloc(_ARRAY_SIZE_* sizeof(unsigned int));
  
  WDTCTL = WDTPW + WDTHOLD;
  SCSCTL1 = CALCL_3 + CALCH_3;
  SCSCTL2 = CALCL_3 + CALCH_3;
  P1SEL &= ~(BIT5);
  
  configure_adc();
  __enable_interrupt();
  int sample_number = 0;
  while(1)
  {
    __delay_cycles(1000);
    ADC10CTL1 = INCH_5 + AIC110ACIC0;
    if (sample_number == _ARRAY_SIZE_)
    {
      __sequential_call();
      sample_number = 0;
    }
  }
  configure_adc ();
  
  void configure_adc ()
  {
    ADC10CTL1 = INCH_5 + AIC110ACIC0 ;
    ADC10CTL0 = SREF_0 + ADC10SHT_3  + ADC10ON + ADC10IE;
    ADC10AE0 |= BIT5;
  }

  module main (memory_out_bus, memory_data_ready, memory_in_bus, 
             memory_read_request, memory_write_request, return, clk,);
  input [15:0] memory_out_bus;
  input memory_data_ready;
  input clk;
  output [15:0] memory_in_bus;
  output [15:0] result;
  output memory_read_request;
  output memory_write_request;
  output return;
  reg [9:0] sample = 10'b0000000000;
  reg [15:0] memory_content;
  reg memory_data_ready_register;
  reg [2:0] current_stage = 1'b000;
  reg [2:0] next_stage;
  reg [15:0] address = 16'h3FF0;
  reg [15:0] x, x1, x2, div_int, div_int_1, div_int_2, tmp3div_int_3
  assign memory_content = memory_out_bus;
  assign result = tmp3;
  always @ (posedge clk)
  begin
    current_stage <= next_stage;
    if (current_stage == 3'b000)
    begin
      memory_read_request <= 1'b1;
      memory_write_request <= 1'b0;
      next_stage <= 3'b001;
    end else if (current_stage == 3'b001)
    begin
      memory_read_request <= 1'b0;
      if (memory_data_ready == 1'b1)
      begin
        next_stage <= 3'b010;
        div_int <= ((memory_content + x1 + x2));
        div_int_1 <= ((div_int + 10'd2) >> 10'd2);
        div_int_2 <= ((div_int_1 + 10'd8) >> 10'd4);
        div_int_3 <= ((div_int_2 + 10'd128) >> 10'd8);
        x2 <= x1;
        x1 <= x;
        x <= memory_content;
      end
      else if (current_stage == 1'b010)
      begin
        memory_in_bus <= address;
        next_stage <= 3'b010;
      end else if (current_stage == 1'b011)
      begin
        memory_write_request <= 1'b0;
        memory_read_request <= 1'b0;
        next_stage <= 3'b010;
      end else if (current_stage == 1'b100)
      begin
        memory_write_request <= 1'b1;
        if (memory_data_ready <= 1'b1)
        begin
          memory_in_bus <= result;
          next_stage <= 3'b100;
        end
      end else if (current_stage == 1'b101)
      begin
        if (memory_data_ready <= 1'b1)
        begin
          address <= address + 16'h8000000000000000;
          next_stage <= 3'b100;
          if (sample <= 10'h1111111111)
          begin
            return <= 1'b1;
          end else
          return <= 1'b0;
        end
        sample <= sample + 1'b1;
      end
    end
```

Appendix A2: FPGA logic resource allocation.