phoeniX: A RISC-V Platform for Approximate Computing

Version 0.1

Arvin Delavari, Faraz Ghoreishy,

Hadi Shahriar Shahhosseini, Sattar Mirzakuchaki

arvin_delavari@elec.iust.ac.ir, faraz_ghoreishy@elec.iust.ac.ir

shahhosseini@iust.ac.ir, m_kuchaki@iust.ac.ir

Summer 2023

Iran University of Science and Technology, Electronics Research Center

Digital Design Research Lab – Super Computing and Networking (SCaN) Research Lab

This page was intentionally left blank.

Preface:

The phoeniX is a 5-stage pipelined 32-bit RISC-V processor written in Verilog HDL. This project was started in summer of 2023 with collaboration of two research laboratories; Digital Design Research Laboratory and Super Computing and Networking (SCaN) Research Laboratory of Electronics Research Center at Iran University of Science and Technology. This document contains technical specifications and user manual for version 0.1 of the phoeniX processor. The version of the processor which is explained in this document (version 0.1) is able to execute RV32I (integer operations extension) instructions of official RISC-V instruction set manual [1]. Other extensions including new and unique features will be supported by the core in the future updates.

The designed core is capable of execution of assembly and C codes using iverilog simulator. Instructions for the flow of code execution on this processor is included in this document. For execution of C codes, the flow will use RISC-V GCC compiler toolchain and for assembly codes, you can use GCC compiler or Venus simulator which is a free extension for the recognized code editor, Microsoft Visual Studio Code. Further descriptions are included in relative sections of this document.

This project is an open-source CPU under the GNU V3.0 license and is free to use. You can find source codes and documentations in the following GitHub repository:

https://github.com/phoeniX-Digital-Design/phoeniX

The list of contributors and members of the research team of phoeniX project:

- Arvin Delavari, arvin_delavari@elec.iust.ac.ir
- Faraz Ghoreishy, faraz_ghoreishy@elec.iust.ac.ir
- Hadi Shahriar Shahhosseini PhD, shahhosseini@iust.ac.ir
- Sattar Mirzakuchaki PhD, m_kuchaki@iust.ac.ir

Table of Contents

List of Figures

List of Tables

Chapter 1

1 Introduction

"Crafting a memorable introduction is an art form that invites readers to embark on a journey they won't soon forget."

Lisa Wilson

The phoeniX project was initiated in summer of 2023 in Electronics Research Center at Iran University of Science and Technology. This RISC-V processor was designed in order to be an original base processor with acceptable performance and specification which aims to lay a solid groundwork and foundation for future computer architecture research and development endeavors especially in the growing field of approximate computing.

The processor uses RISC-V open-source Instruction Set Architecture (ISA) with a custom designed microarchitecture. The core is written in Verilog HDL and it is synthesizable for both FPGA and ASIC targets. Building blocks of this core were all written in separate modules which would help developers to add features and test computer architecture techniques on the base core. This modular design leads to a simpler way of debugging and also extending the core.

In the following version (version 0.1), phoeniX is a 5-stage pipelined RV32I processor which supports “I-Extension” of RISC-V ISA. Other extensions will be added to the core in the upcoming updates soon. The key goal of phoeniX project, is to be a unique and global platform for approximate computing in RISC-V embedded processors.

The phoeniX processor stands out with its user-friendly and straightforward flow for simulating and executing C and assembly codes on the processor. Unlike many other open-source processors, phoeniX offers a simplified approach that is easier to follow. The repository of this project provides comprehensive documentation, offering step-by-step guidance on executing codes on the core. This resource is highly accessible and designed to facilitate a seamless development experience in computer architecture and digital design field.

1.1 The RISC-V Instruction Set Architecture

ISA stands for Instruction Set Architecture. In computer architecture, an ISA refers to the set of instructions that an Instruction Set Processor (ISP) can execute. It defines the interface between hardware and software, serving as a bridge for communication and coordination between the two.

The ISA specifies the available instructions, their formats, and the behavior of the processor when executing each instruction. It defines the operations that can be performed by the processor, data types supported, addressing modes, and the control flow instructions. The ISA also encompasses the registers, memory organization, and input/output mechanisms.

The ISA provides standardized and consistent attributes for software as perceived by compiler and developers to create programs that can run on different hardware platforms. It abstracts the underlying complexities of the hardware implementation, allowing software to be written at a higher level of abstraction.

Different computer architectures can have different ISAs. Common examples include x86, ARM, MIPS, and RISC-V. Each ISA has its own unique set of instructions and design principles, tailored for specific purposes, performance goals, and target applications.

The ISA serves as a fundamental aspect of computer architecture, as it influences the overall performance, compatibility, and flexibility of a computing system. It forms the basis for software development, compiler design, and system-level optimizations.

Before discussing the RISC-V ISA, it is important to know a very important concept in computer organization and architecture, which is the difference of RISC and CISC machines.

RISC (Reduced Instruction Set Computer) processors have a simplified and streamlined instruction set design. They employ a small number of simple instructions, each performing a single operation. This simplicity enables easier decoding and execution, resulting in faster and more efficient processing. RISC architectures often use a load/store architecture, where memory access is limited to specific load and store instructions. Arithmetic and logical operations are performed only on registers, improving efficiency and reducing memory traffic. RISC processors are well-suited for pipelining, a technique that divides instruction execution into multiple stages,

allowing for parallel processing and improved performance. They also work closely with optimizing compilers, facilitating efficient code generation and optimization techniques.

CISC (Complex Instruction Set Computer) processors have a more extensive and complex instruction set. They feature a large number of complex instructions that can perform multiple operations in a single instruction. CISC instructions often include memory access, arithmetic operations, and control flow operations. CISC processors may allow direct memory access from memory to memory, reducing the need for explicit load and store instructions. CISC architectures aim to provide powerful instructions that can handle complex tasks directly in hardware, reducing reliance on software and potentially improving execution time for certain operations. CISC instructions can vary in length, which adds complexity to the decoding process but allows for packing more functionality into individual instructions.

It's worth noting that the distinction between RISC and CISC has become less clear in modern processors. Many contemporary processors employ hybrid approaches that combine features from both design philosophies. The choice between RISC and CISC depends on factors such as performance goals, power efficiency, instruction density, and the target applications of the processor.

The RISC-V ISA (Instruction Set Architecture) is an open-source instruction set architecture that was designed to be simple, modular, and extensible. It was developed by researchers at the University of California, Berkeley in 2010 and has gained significant traction and popularity in both academic and industrial communities.

The RISC-V ISA follows the Reduced Instruction Set Computer (RISC) design philosophy, emphasizing simplicity and efficiency. It provides a standard set of instructions that a RISC-V processor can execute. The instruction formats are designed to be compact and easy to decode, enabling efficient instruction fetch and execution.

One of the notable features of the RISC-V ISA is its modular design. It offers a base instruction set that provides essential instructions for general-purpose computing. In addition to the base instruction set, RISC-V supports various optional instruction extensions, allowing designers to customize and tailor the ISA to specific application domains or performance requirements. These

extension modules include support for floating-point operations, atomic operations, vector instructions, and more.

The RISC-V ISA is designed to be portable and platform-independent, enabling easy migration of software across different implementations. It supports 32-bit, 64-bit and 128-bit instruction widths, providing flexibility for different computing environments.

Being an open-source ISA, the RISC-V architecture encourages collaboration, innovation, and customization. It has gained widespread adoption not only in academia but also in industry, with several companies developing RISC-V-based processors, systems-on-chip (SoCs), and development tools.

Overall, the RISC-V ISA offers a flexible, open, and scalable architecture that promotes innovation and fosters a diverse ecosystem of RISC-V-based hardware and software solutions.

RISC-V stands out among other ISAs due to the following advantages:

- Open Source: RISC-V is an open-source ISA, fostering collaboration, innovation, and customization, Modularity and Extensibility: Its modular design allows for optional instruction extensions, tailoring the ISA to specific application needs.
- Simplified Instruction Set: RISC-V follows a streamlined design, enhancing decode and execution efficiency while reducing complexity.
- Portability: RISC-V is platform-independent, enabling seamless software migration across different implementations.
- Education and Research: RISC-V's open nature makes it ideal for educational institutions and research projects, promoting exploration and experimentation.
- Industry Adoption: RISC-V has gained significant industry traction, with companies leveraging it for custom processors and specialized accelerators.

These advantages make RISC-V an attractive choice, offering flexibility, customization, and cost-effectiveness in various computing applications.

RISC-V has gained notable adoption in both established companies and innovative projects across various industries. Here are a few examples:

- **SiFive:** SiFive, a leading RISC-V company, offers RISC-V-based processors and development platforms. They provide customizable and scalable solutions for a wide range of applications, including AI, IoT, and edge computing.
- **Western Digital:** Western Digital has embraced RISC-V for its storage products. They utilize RISC-V cores in their controllers and firmware, leveraging the openness and flexibility of the ISA.
- **NVIDIA:** NVIDIA has integrated RISC-V cores into their graphics processing units (GPUs) to enhance security and enable custom functionality in their hardware platforms.
- **Esperanto Technologies:** Esperanto is developing high-performance RISC-V-based processors for AI and machine learning workloads. They aim to deliver energy-efficient solutions for data centers and edge devices.
- **Google:** Google has expressed interest in RISC-V for various projects. They have been exploring the use of RISC-V in their data centers, potentially leveraging its flexibility and efficiency for specialized applications.
- **Andes Technology:** Andes provides high-performance RISC-V processor IP cores for embedded applications. Their solutions target areas such as IoT, automotive, and industrial automation.

Additionally, there are numerous open-source projects and research initiatives utilizing RISC-V. For instance, the CHIPS Alliance fosters collaboration on open-source hardware, including RISC-V designs. The PULP Platform focuses on developing energy-efficient RISC-V cores for IoT devices. The RISC-V International association, formerly the RISC-V Foundation, promotes the advancement and adoption of the RISC-V ISA.

It's important to note that RISC-V's adoption continues to grow rapidly, and more companies and projects are actively exploring and implementing RISC-V solutions in their products and research endeavors.

1.2 phoeniX core structure

In the beginning, one of the most specialties of this processor which should be mentioned is the removal of a centralized control unit and not utilizing a solitary control module. This processor employs a "Self Control Logic" which leads to elimination of control unit from the building blocks. In SCL, after decoding the instructions, fields that are related to the production of control signals such as *opcode*, *funct3*, *funct7* and other fields are directly fed to the linked modules, and the process of control signals generation will be taken place in the target module hardware. For example, in a basic processor decoded fields are sent to control unit and inside this module, signals will be generated to define ALU operation, ALU multiplexers (for selection between registers or immediate values), bypassing and forwarding multiplexers and etc. In phoeniX processor, all of these control signals are determined inside the target modules. By sending the relative decoded fields directly to the targets, each module has become self-controllable and when each one is stationed in its own relative stage a Self Control Logic system will form.

The phoeniX is a classic 5-stage pipelined RISC-V processor which includes the following machine cycle stages: Fetch, Decode, Execute, Memory and Write-Back. Building blocks of the processor are mostly designed in separated modules which gives developers freedom for changes and the mentionable benefit of modularity. Registers used for latching data in different stages of the core are not designed as modules and are completely written inside the main module named *phoeniX*. Detailed descriptions about the hardware design of the core and its modules are included Building Blocks (Modules) section of this documents.

Memory interface of the core is completely written and described in testbench file because of the limitations of memory simulation using Hardware Description Languages. Further information about the interface logic is inside Memory Interface section of this document.

The phoeniX core also has a hazard detection unit which gives the core, the ability of performing data-forwarding and bypassing techniques in order to reduce stalls in codes including data dependencies and hazards.

The maximum delay time (critical path) of a phoeniX processor stage analyzed using Yosys [2] open-source synthesis tool and Vesta [3] static time analysis software, in different stages is somewhere around 3800-3900 picoseconds which leads us to a clock cycle width of 4

nanoseconds. This means the phoeniX processor will perform in a frequency rate of nearly 250MHz which is a remarkable number in benchmarking with other embedded processors. Most of the commercial and industrial embedded processors designed by ARM with Cortex M0, M3 and M4 microarchitecture have a lower core frequency. These processors find extensive integration with microcontroller series such as STM32 (by ST) or LPC (by NXP), making them highly prevalent and widely employed in the field of embedded systems.

Another open-source RISC-V embedded processor, PicoRV32 by Claire Wolf [4] achieves a lower frequency than mentioned. PicoRV32, is one of the most recognized open-source processors and even used in fabricated microcontroller chips has a frequency rate ranged in 50 to 100MHz dependent on the technology node. Another well-known RISC-V processor, ibex core by ETH Zurich [5] university which also has a frequency range between 100MHz and 300MHz in max performance configurations.

The phoeniX processor is a reliable and user-friendly solution, offering modularity and easy extensibility for developers and computer architecture researchers. It features a simplified design that is easy to understand, while still delivering competitive performance compared to other processors.

1.3 phoeniX software interface

The phoeniX processor uses standard GCC compiler toolchain [6] which is officially verified by RISC-V organization, in order to run C codes and assembly codes on the processor. These codes are turned into a firmware file including hex format of instructions and addresses by the compiler toolchain. The generated firmware file is given to the testbench of phoeniX CPU as a unified memory module. The processor can simulate the compiled machine code using iverilog [7] version 12 tool. A shell script is provided in the repository which helps users install all software requirements and perquisites for simulating and coding on phoeniX processor.

The simulation and execution flow are simplified by phoeniX code executants which will be described briefly in the upcoming sections of this document. They are straightforward solutions to simulate assembly and C codes on both Windows and Linux systems on phoeniX processor. Linux

systems can use the Makefile in the main directory which will do the complete process of compiling, assembling, generating firmware file and executing on the phoeniX core without any complexity and additional user interference. There is a second solution which can be used for both Windows and Linux systems, but it can only be utilized with simulation of assembly codes on the processor. You can write and simulate your RISC-V assembly code using Venus simulator which is a Microsoft Visual Studio Code extension. This extension has an output file which includes hex instructions of the assembly code. The output will be given to a code executant python script and after that, the desired firmware will be generated and ready to be executed on phoeniX processor.

The phoeniX processor streamlines the simulation and execution process, ensuring a user-friendly and intuitive experience. Its simplified flow enables effortless utilization, making it accessible and straightforward for users of all levels of expertise. Additional details regarding the integration of further descriptions can be found in the Code Executant Software section of this document, providing comprehensive insights into how the phoeniX processor incorporates advanced functionalities and features.

1.4 Required Software

As mentioned before, a shell script is provided integrated in the phoeniX RV32 core repository, which helps users install all of the required software for simulation and execution flow without any problem. In this section there are some additional information about the software used by phoeniX core execution flow.

- **Python3**: Python is also required to be installed on the system in order to execute the code executant and firmware generator scripts. Linux distributions such as Ubuntu come with Python3 already installed by default. On Windows systems Python needs to be installed separately.

- **iverilog:** iverilog is an open-source Verilog simulation and synthesis tool used for designing and testing digital circuits. It supports the IEEE 1364-2005 Verilog standard and provides a command-line interface for compiling and simulating Verilog code. iverilog allows users to simulate and verify the behavior of their digital designs before actual hardware

implementation, making it a valuable tool for digital circuit development and verification. In phoeniX processor project, iverilog version 12 is used for HDL simulation process and final execution of C and assembly codes on the processor.

- **GTKWave:** GTKWave is an open-source waveform viewer for analyzing and visualizing electronic waveforms. It is commonly used in digital design and verification processes, particularly in the field of hardware description languages (HDL) and digital circuit simulation. With GTKWave, you can load waveform files and view the signal waveforms, timing diagrams, and other attributes of digital signals in the phoeniX core. GTKWave also supports advanced features such as hierarchical waveform viewing, cross-probing between source code and waveforms, and the ability to apply filters and color schemes to enhance waveform visualization.

- **RISC-V GCC Compiler Toolchain:** The RISC-V GCC compiler toolchain refers to a collection of software tools that enable the compilation and development of software for RISC-V architecture-based processors. It includes a set of open-source tools, primarily based on the GCC (GNU Compiler Collection), specifically tailored for the RISC-V instruction set architecture (ISA). The RISC-V GCC compiler toolchain plays a crucial role in the development and software ecosystem surrounding the RISC-V architecture, enabling the creation of applications, firmware, and operating systems for RISC-V processors.

- **Venus Simulator (Visual Studio Code):** The Venus simulator is a RISC-V instruction set architecture (ISA) simulator developed by the University of Victoria. It allows users to simulate and execute RISC-V assembly language programs, providing a platform for learning, testing, and debugging RISC-V code. The Venus simulator Visual Studio Code extension is an extension specifically designed for the Visual Studio Code (VS Code) integrated development environment. This extension integrates the Venus simulator directly into the VS Code environment, offering an enhanced development experience for RISC-V programming. With the Venus simulator extension, users can write RISC-V assembly code directly in VS Code, benefit from syntax highlighting and code completion features, and seamlessly run and debug their code using the Venus simulator. The extension

provides an interactive interface within VS Code, allowing users to step through their code, set breakpoints, inspect registers and memory, and observe program execution.

In References section of this document there are useful links with detailed descriptions about the tools used in this project and their installation guides. While the provided shell script automates the installation process on Linux systems, it's important to note that for Windows operating systems, the required software needs to be downloaded and installed separately.

Chapter 2

2 Building Blocks (Modules)

“Great things are done by a series of small things

brought together.”

Vincent van Gogh

In the following chapter, a comprehensive overview of the essential components that form the foundation of the phoeniX core is provided. We will delve into the intricate details regarding the structure and architecture of these modules, presenting them in the order of the dataflow within the pipeline.

Within this project, we have identified 9 key modules that play a crucial role in its operation. Each of these modules will be meticulously described, ensuring a thorough understanding of their individual contributions to the overall pipelined data path. By following the logical progression of the dataflow, you will gain valuable insights into how these modules interact and collaborate to achieve the desired outcomes. Through this comprehensive examination, you will acquire a profound knowledge of the phoeniX core's inner mechanisms.

The top module named phoeniX which acts as a unifying entity that brings together all the aforementioned building blocks, along with additional registers and latches, within the pipeline. By integrating these components into a cohesive unit, the phoeniX module forms the backbone of the processor, orchestrating the flow of data and executing the desired operations.

The memory interface logic is a critical component that facilitates communication between the processor and the memory subsystem. Although it is not directly included within the individual modules, its role is of utmost importance in ensuring the overall functionality of the system. The memory interface logic is implemented separately in the dedicated testbench file. In the forthcoming chapter dedicated to Memory Interface Logic, we will provide a thorough explanation of this logic, delving into its intricate workings. We will explore how it manages the flow of data

between the processor and the memory subsystem, and we will highlight its significance within the broader system architecture.

Before we introduce each of these modules, table 1 included in this section which explains what is the role of each module of the core briefly. It is needed to note that in the upcoming versions of the processor, new features and new modules will be added to the phoeniX core and the mentioned modules in this chapter are the foundation of phoeniX V0.1 CPU.

Modules	Description
Address Generator	Generating address for BRANCH, JUMP and LOAD/STORE instructions
Arithmetic Logic Unit	ALU with support for I_TYPE and R_TYPE instructions
Fetch Unit	Instruction Fetch logic and program counter addressing
Hazard Forward Unit	Hazard detection and data forwarding logic in pipelined processor
Immediate Generator	Generating immediate values according to instructions type
Instruction Decoder	Decoding instructions and extracting *opcode*, *funct* and *imm* fields
Jump Branch Unit	Condition checking for all branch instructions
Load Store Unit	Load and Store operations for aligned addresses and word size management
Register File	Parametrized register file suitable for GP registers and CSRs

Table 1 - phoeniX core modules and descriptions

2.1 Fetch Unit

The Fetch Unit module is responsible for fetching instructions from memory in a processor design.

Input signals:

- `enable`: This input signal is used to enable the memory interface module.
- `PC`: Represents the Program Counter, which holds the address of instruction currently being fetched.
- `address`: This input signal represents the branch or jump target address generated by the Address Generator module.
- `jump_branch_enable`: This signal indicates whether a branch or jump is taken.

Output signals:

- `next_PC`: This output signal is a 32-bit value that represents the address of the next instruction to be fetched and will be latched on the Program Counter register.

Memory Interface Signals:

- `memory_interface_enable`: This output signal is a 1-bit register that enables the memory interface.
- `memory_interface_state`: This output signal is a 1-bit register that indicates the state of the memory interface operation. In the fetch module, it is always set to *READ*, implying that read-only behavior of the code section of the memory.
- `memory_interface_address`: This output signal is a 32-bit register that represents the address being accessed by the process. In this module, it is set to the current value of PC.
- `memory_interface_frame_mask`: This output signal is a 4-bit register that specifies the frame mask for the memory interface. It is set to `4'b1111`, indicating that all four bytes of the memory frame are enabled (Further descriptions on memory frame and frame mask are provided in chapter 3).

The next_PC output is determined based on the jump_branch_enable input signal:

- If `jump_branch_enable` is asserted (`1`), indicating a branch or jump to be taken, next_PC is set to the value of address.
- If `jump_branch_enable` is not asserted (`0`), next_PC is calculated as the current PC value plus 4 (`32'd4`), representing the increment of the Program Counter by 4 bytes (1 word i.e., the size of an instruction).

In summary, the Fetch Unit module generates the necessary signals for the memory interface and determines the next instruction's Program Counter based on the received control signals.

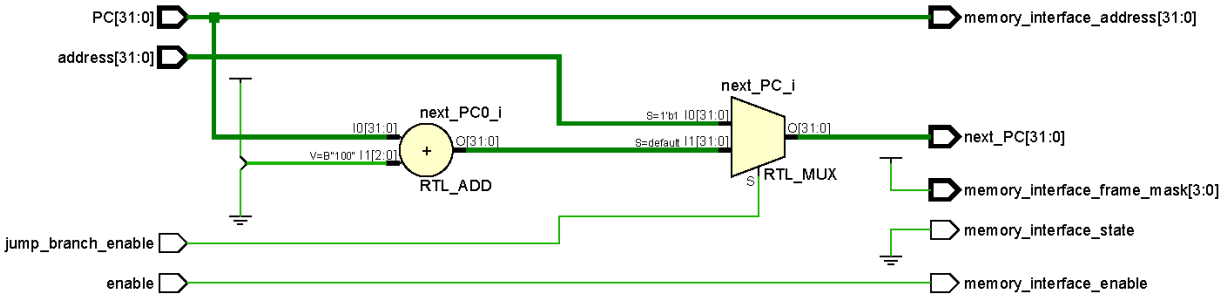

Figure 1 - Fetch Unit schematic output from Xilinx Vivado software

2.2 Instruction Decoder

The Instruction Decoder module is responsible for decoding instructions in a processor design.

Input signals:

- `instruction`: This input signal is a 32-bit value representing the instruction to be decoded.

Output signals:

- `instruction_type`: This output signal is a 3-bit value representing the type of the instruction. It is assigned one of the predefined values: `R_TYPE`, `I_TYPE`, `S_TYPE`, `B_TYPE`, `U_TYPE`, `J_TYPE`, or `1'bz` (for an unknown or invalid instruction type).
- `opcode`: This output signal is a 7-bit value representing the *opcode* of the instruction.
- `funct3`: This output signal is a 3-bit value representing the *funct3* field of the instruction.
- `funct7`: This output signal is a 7-bit value representing the *funct7* field of the instruction.
- `funct12`: This output signal is a 12-bit value representing the *funct12* field of the instruction.
- `read_index_1`: This signal is a 5-bit value representing the first source register index.
- `read_index_2`: This signal is a 5-bit value representing the second source register index.

- `write_index`: This signal is a 5-bit value representing the destination register index.

Internal signals:

- `instruction_type_r`, `instruction_type_i`, `instruction_type_s`, `instruction_type_b`, `instruction_type_u`, `instruction_type_j`: These internal signals evaluate whether the instruction falls into a specific type category
- `read_enable_1`, `read_enable_2`, `write_enable`: These internal signals control the read and write enable signals for the register file based on the instruction type.

Behavior:

The `opcode`, `funct3`, `funct7`, `funct12`, `read_index_1`, `read_index_2`, and `write_index` outputs are assigned values based on specific bits of the instruction input signal.

The `instruction_type` output is determined based on the evaluation of the internal signals mentioned above.

The `read_enable_1`, `read_enable_2`, and `write_enable` signals are evaluated based on the `instruction_type` value using a case statement. Each case assigns the appropriate values to these signals for the corresponding instruction type. Additionally, there is a conditional check to disable the `write_enable` signal when the destination register index (`write_index`) is x0 (all zeros), as writing to register x0 is invalid.

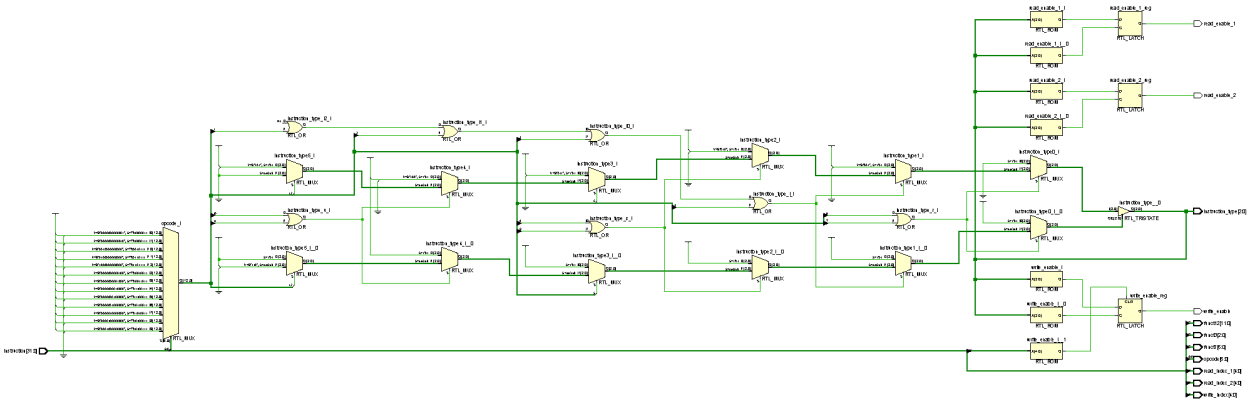

Figure 2 - Instruction Decoder schematic output form Xilinx Vivado software

2.3 Immediate Generator

The Immediate Generator module is responsible for generating the immediate value for instructions based on the instruction type.

Input signals:

- `instruction`: This input signal is a 32-bit value representing the instruction.
- `instruction_type`: This input signal is a 3-bit value representing the type of the instruction. It is used to determine how the immediate value should be generated.

Output signals:

- `immediate`: This output signal is a 32-bit value representing the immediate value generated from the instruction.

Behavior:

Inside the always block, a case statement evaluates the `instruction_type` value.

Depending on the value of `instruction_type`, the structure of the immediate value is assigned to specific fields of the instruction.

For example, if the `instruction_type` is `I_TYPE`, the immediate value is assigned by concatenating 21 duplicates of the most significant bit of the instruction to bits 31 to 11 of the instruction. You can find the immediate generation scheme for each instruction type in RISC-V instruction set architecture specification documents.

If the `instruction_type` does not match any of the predefined types (default case), the immediate value is assigned 32 bits of `1'bz` (unknown value).

In summary, the Immediate Generator module takes an instruction and the instruction type as inputs and generates the immediate value based on the instruction type. The immediate value generated is then assigned to the immediate output signal.

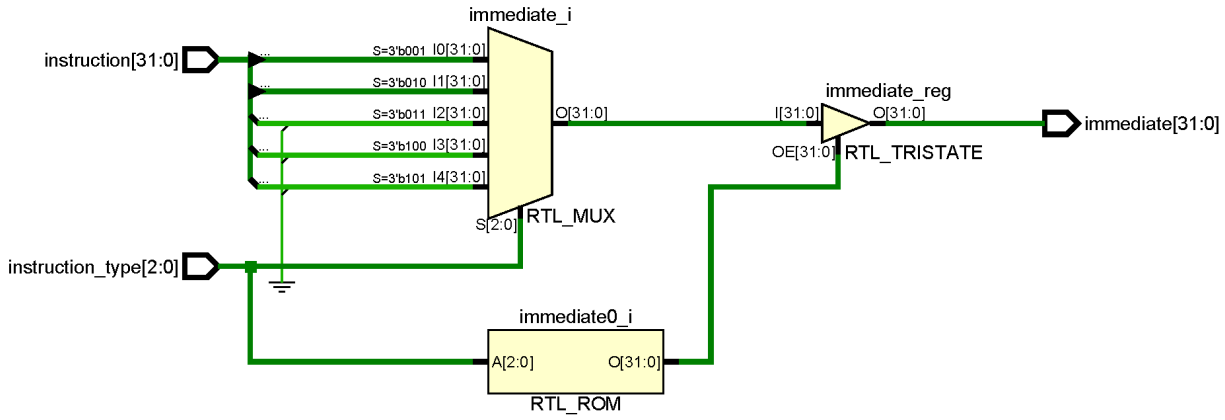

Figure 3 - Immediate Generator schematic output from Xilinx Vivado software

2.4 Address Generator

The Address Generator module is responsible for generating addresses for memory accesses and jump and branch instructions based on the opcode and other inputs.

Input signals:

- `opcode`: This input signal is a 7-bit value representing the opcode of the instruction.
- `rs1`: This input signal is a 32-bit value representing the value of register rs1.
- `PC`: This input signal is a 32-bit value representing the program counter.
- `immediate`: This input signal is a 32-bit value representing the immediate value.

Output signals:

- `address`: This output signal is a 32-bit value representing the generated address.

Internal Signals:

- `value`: This internal signal is a 32-bit value used to hold intermediate values based on the opcode.

Behavior:

Inside the always block, a case statement evaluates the opcode value. Depending on the `opcode`, the `value` signal is assigned to specific values based on the `opcode`.

For example, if the `opcode` matches `STORE`, the value signal is assigned the value of `rs1`. Similarly, for `LOAD`, `JAL`, `JALR`, and `BRANCH`, the value signal is assigned the values of `rs1` and `PC`, respectively.

If the `opcode` does not match any of the predefined opcodes (default case), the value signal is assigned `1'bz` (unknown value). Finally, the `address` output signal is assigned the value of value plus the immediate value.

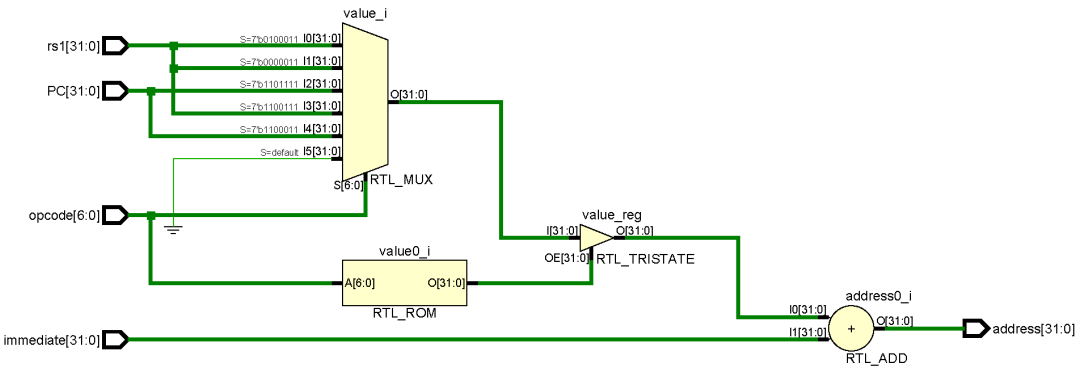

Figure 4 - Address Generator schematic output from Xilinx Vivado software

In summary, the Address Generator module takes the `opcode`, `rs1` value, program counter (`PC`), and `immediate` value as inputs. Based on the `opcode`, it generates the appropriate address by combining the value of `rs1`, `PC`, or other intermediate values with the `immediate` value. The generated address is then assigned to the `address` output signal.

2.5 Arithmetic Logic Unit

The ALU is responsible for executing R-Type, I-Type, U-Type and J-Type instructions. It takes several inputs, including the `opcode`, `funct3`, `funct7`, `PC` (Program Counter), `rs1` (Register Source 1), `rs2` (Register Source 2), and `immediate` (Immediate Value). The output `alu_output`, holds the result of the ALU operation.

The module contains the following components:

- **Multiplexers**: There are two multiplexers with `mux1_select` and `mux2_select` signals that are used to select the appropriate operands for the ALU operation based on the `opcode`. The `mux1_select` determines whether the first operand should come from `rs1` or `PC`, while the `mux2_select` determines whether the second operand should come from `rs2`, `immediate`, or a constant value of 4.
- **ALU Operation Evaluation:** The always block with sensitivity to `opcode`, `funct3`, and `funct7` evaluates the ALU operation based on the input signals. It uses a `casex` statement to match the `opcode`, `funct3`, and `funct7` values and perform the corresponding ALU operation.

The ALU operations include arithmetic operations (addition, subtraction, left shift, right shift), logical operations (bitwise AND, OR, XOR), comparison operations (less than, less than unsigned), and jump instructions. The result of the ALU operation is assigned to the `alu_output` register.

As mentioned before, control signals in ALU are determined inside the module in confirmation of Distributed Logic Control of phoeniX CPU. Overall, this module provides the ALU functionality necessary to execute various instruction in the phoeniX core.

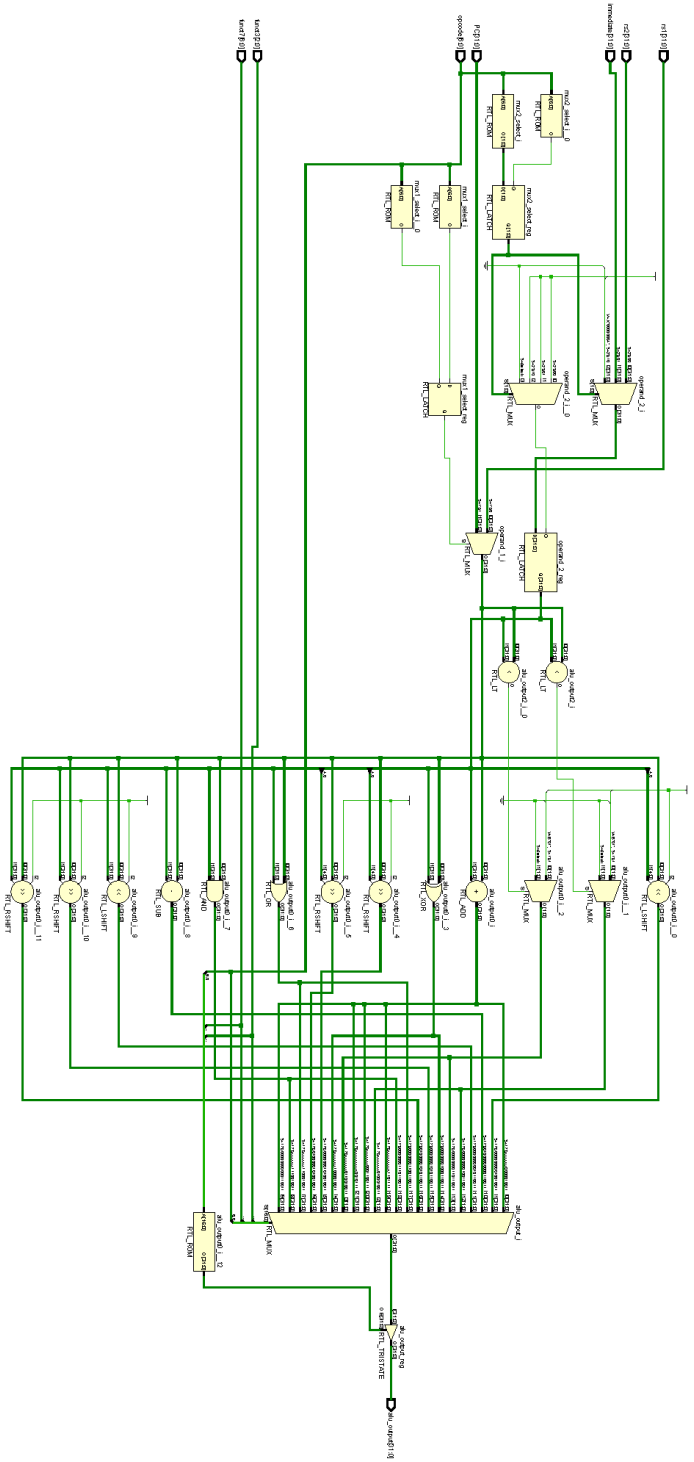

Figure 5 - Arithmetic Logic Unit schematic output from Xilinx Vivado output

2.6 Register File

The Register File module represents a register file in a digital system. It is implemented as a synchronous sequential circuit and provides storage for a set of registers. Here is a description of the module:

Parameters:

- `WIDTH`: Specifies the width (number of bits) of each register in the file. The default value is 32.
- `DEPTH`: Specifies the depth (bits needed to represent number of registers) in the file. The default value is 5.

Input signals:

- `CLK`: Clock signal used for synchronous operations.
- `reset`: Asynchronous reset signal used to initialize the register file.
- `read_enable_1`: A control signal to enable reading from register 1.
- `read_enable_2`: A control signal to enable reading from register 2.
- `write_enable`: A control signal to enable writing to a register.
- `read_index_1`: The index of the register to read from for register 1.
- `read_index_2`: The index of the register to read from for register 2.
- `write_index`: The index of the register to write to.
- `write_data`: The data to be written into the register specified by `write_index`.

Output signals:

- `read_data_1`: The output data read from register 1.
- `read_data_2`: The output data read from register 2.

Internal Signals and Variables:

- `Registers`: An array of registers used to store the register values.

Behavior:

On the positive edge of the `reset` signal, all registers in the `Registers` array are initialized to zero.

On the negative edge of the `CLK` signal, if `write_enable` is asserted (`1'b1`) and the `write_index` is not zero, the data specified by `write_data` is written into the register specified by write_index.

The output `read_data_1` is assigned the value of the register specified by `read_index_1` when `read_enable_1` is asserted (`1'b1`). Otherwise, `read_data_1` is assigned to high-impedance (`1'bz`).

The output `read_data_2` is assigned the value of the register specified by `read_index_2` when `read_enable_2` is asserted (`1'b1`). Otherwise, `read_data_2` is assigned to high-impedance (`1'bz`).

In summary, the Register File module provides a configurable register file with read and write capabilities. It allows reading from two different registers simultaneously and writing to a single register at a time.

2.7 Hazard and Forwarding Unit

The Hazard Detection and Forwarding Unit is responsible for detecting data hazards between instructions and forwarding data from the preceding instructions to the succeeding instructions to resolve these hazards. Here is a description of the module:

Input signals:

- `source_index`: The index of the register being read by the succeeding instruction.
- `destination_index_1`, `destination_index_2`, `destination_index_3`: The indices of the registers being written by the preceding instructions.
- `data_1`, `data_2`, `data_3`: The data values written by the preceding instructions with corresponding indices.

- `enable_1`, `enable_2`, `enable_3`: Control signals indicating whether the preceding instructions are writing to the registers or not.

Output signals:

- `forward_enable`: A control signal indicating whether data forwarding is enabled.
- `forward_data`: The forwarded data value.

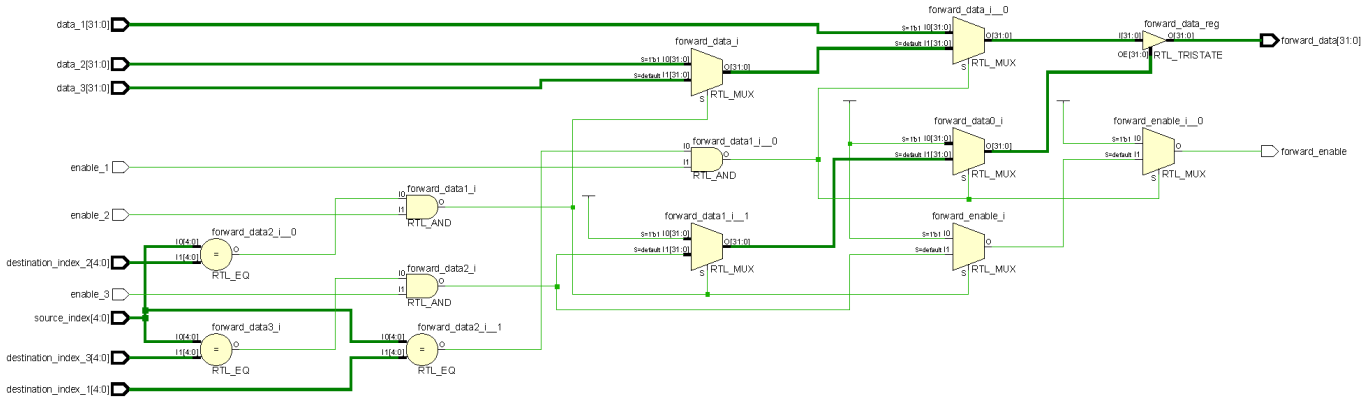

Figure 6 - Hazard Forward Unit schematic output from Xilinx Vivado software

Behavior:

The module uses an `always @(*)` block to perform combinational logic based on the input signals. It checks if the `source_index` matches any of the `destination_index` values from the preceding instructions and if the corresponding enable signal is asserted (`1'b1`).

If a match is found and the corresponding enable signal is set, the data value written by the preceding instruction is assigned to `forward_data`, and `forward_enable` is set to `1'b1` to indicate that data forwarding is enabled. If no match is found or if the corresponding enable signal is not set, `forward_data` is set to high-impedance (`32'bz`), and `forward_enable` is set to `1'b0` to indicate that data forwarding is not enabled.

In summary, the Hazard and Forwarding Unit module detects data hazards by comparing the source index of the current instruction with the destination indices of the preceding instructions. If a hazard is detected, it enables data forwarding by setting the `forward_enable` signal and forwards the appropriate data value to resolve the hazard. Otherwise, it disables data forwarding and sets the `forward_data` signal to high-impedance.

2.8 Jump and Branch Unit

The Jump Branch Unit module is responsible for determining whether a jump or branch instruction should be taken. It generates a control signal, `jump_branch_enable` based on the `opcode`, `funct3`, and `instruction_type` signals which indicates whether a jump or branch should occur.

Input signals:

- `opcode`: The opcode of the instruction.
- `funct3`: The funct3 field of the instruction.
- `instruction_type:` The type of the instruction.

Output signals:

- `jump_branch_enable`: A control signal that enables a jump or branch operation.

Internal Signals and Variables:

- `branch_enable`: A signal that indicates whether a branch instruction should be taken.
- `jump_enable`: A signal that indicates a jump instruction should be taken.

Behavior:

The module uses an `always @(*)` block to perform combinational logic based on the input signals. If the instruction is a B-type instruction (branch), the module checks the value of the `funct3` field using a casex statement. Depending on the value of `funct3`, the module compares the values of `rs1` and `rs2` (source registers) using signed or unsigned comparisons. If the comparison condition is true, `branch_enable` is set to `1'b1` to indicate that a branch should be taken. Otherwise, `branch_enable` is set to `1'b0`.

If the opcode is a `JAL` or `JALR` instruction, `jump_enable` is set to `1'b1` to indicate that a jump should be taken. Otherwise, `jump_enable` is set to `1'b0`.

Finally, `jump_branch_enable` is assigned to the logical OR of `jump_enable` and `branch_enable`, indicating the instruction's final result.

In summary, the Jump Branch Unit module determines whether a jump or branch instruction should be executed based on the `opcode`, `funct3`, and `instruction_type`. It generates the `jump_branch_enable` control signal to indicate whether a jump or branch operation should occur.

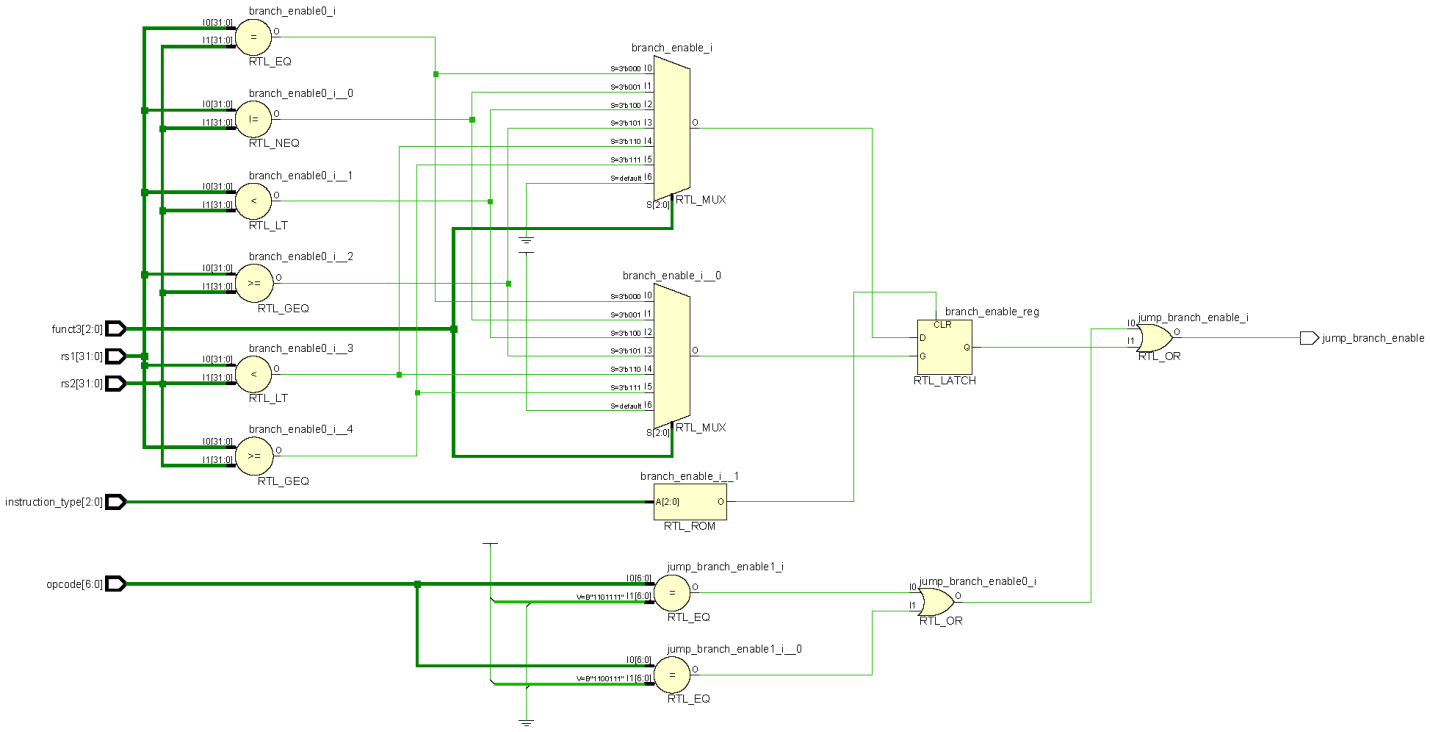

Figure 7 - Jump Branch Unit schematic output from Xilinx Vivado software

2.9 Load Store Unit

The Load Store Unit is responsible for handling all memory accesses for load and store instructions. This module currently only supports word-aligned accesses to memory. Further updates will include support for misaligned accesses by performing two consecutive aligned accesses.

The module has the following input and output ports:

Input signals:

- `opcode`: This input specifies the opcode of the instruction.

- `funct3`: This input specifies the funct3 field of the instruction.
- `address`: This input represents the memory address generated in the Address Generator module.
- `store_data`: This input is connected to Register Source 2 and represents the data to be stored in memory.

Output signal:

- `load_data`: This output represents the data returned from memory for load instructions.

Memory Interface Signals:

- `memory_interface_enable`: This output signal is a 1-bit register that enables the memory interface.
- `memory_interface_state`: This output signal is a 1-bit register that indicates the state of the memory interface operation. (e.g., *READ* or *WRITE*)
- `memory_interface_address`: This output signal is a 32-bit register that represents the address being accessed by the process. In this module, it is set to the value received from the Address Generator.
- `memory_interface_frame_mask`: This output signal is a 4-bit register that specifies the frame mask for the memory interface. Different masks are created for memory interactions based on the instructions word-size access i.e., byte, half-word, word. (Further descriptions on memory frame and frame mask are provided in chapter 3).
- `memory_interface_data`: This output signal represents the data transferred to or from the memory.

The module contains logic to control the memory interface based on the `opcode` and `funct3` inputs. It determines whether the memory interface should be enabled or disabled, sets the memory address, and determines the state and frame mask for the memory interaction.

For load instructions, the module takes the 32-bit data returned from memory and based on the `funct3` field and the frame mask, forms the final value as `load_data`. For store instructions, the module transfers the store data based on the `funct3` field and the frame mask.

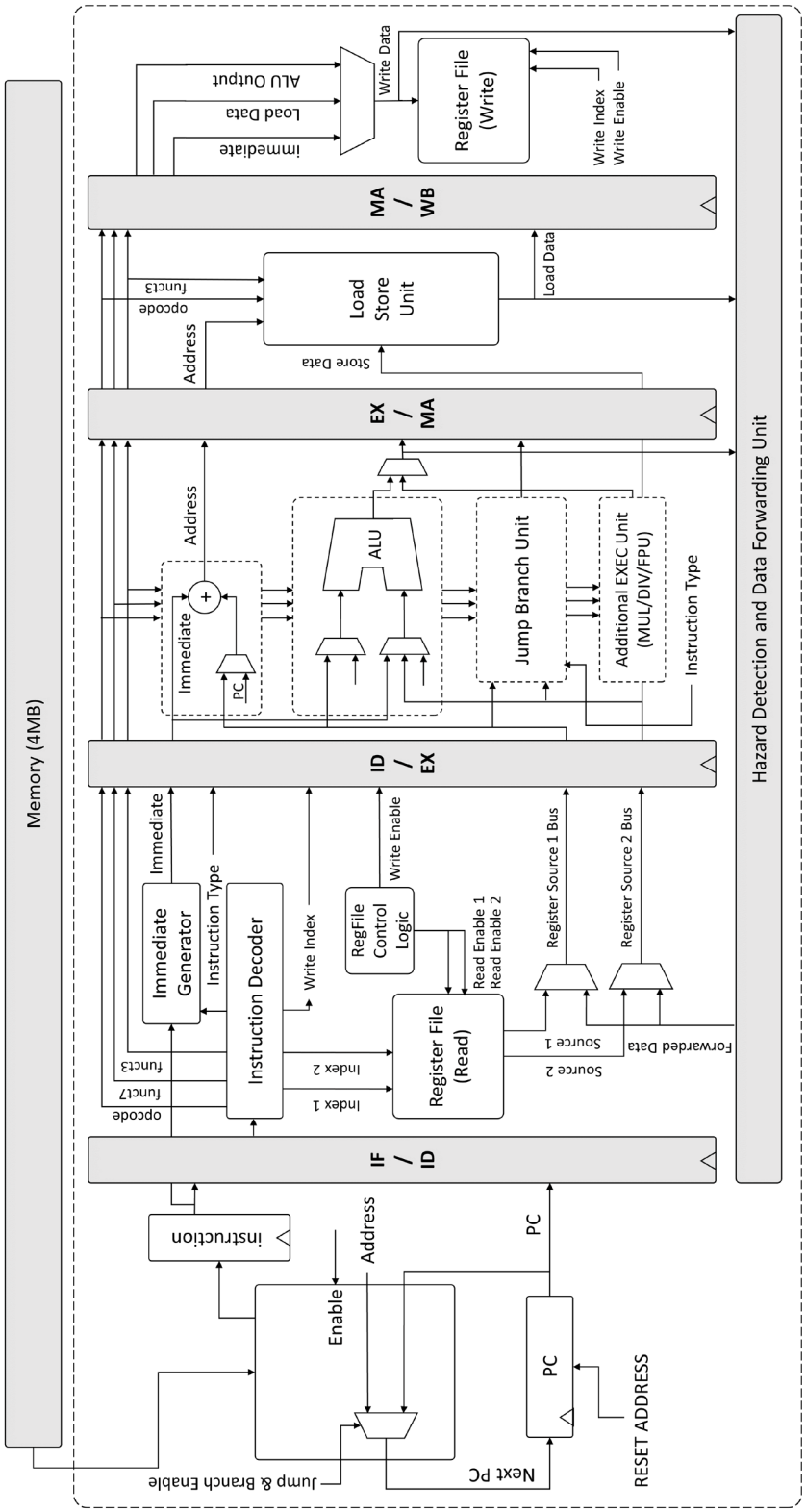

Figure 8 - phoeniX block diagram

Chapter 3

3 Memory Interface Logic

"Sometimes you never know the value of a moment,
until it becomes a memory."

Dr. Seuss

In the following chapter, we will delve into the logic behind the phoeniX core native memory interface system. However, it's important to note that due to the limitations of HDL simulation and synthesis for microprocessors, the logic is implemented solely in the testbench. It is not designed as a distinct unit or module within the core's building blocks.

In the current version of the processor (V0.1), the Fetch Unit and memory interface is designed in a way to load 32-bit hexadecimal instructions based on the positive edge of the core's clock signal. However, this version lacks a standard interface between the memory and the core.

To enhance the memory interface in upcoming updates, it is planned to incorporate standard interfaces such as AXI4 Lite [8] into the project. The inclusion of AXI4 Lite will provide a reliable and widely recognized interface for communication between the memory and the core. This interface will enable seamless data transfer, improved compatibility, and better integration with other components of the system.

The provided testbench in main directory is designed for the top module, phoeniX. The `phoeniX_Testbench` module contains logic for the memory interfaces (instruction memory interface and data memory interface) used by the phoeniX core. The testbench acts a wrapper around the main core and incorporates memory logic into it.

phoeniX currently supports 32-bit word memories with synchronized access time. The core always addresses memory by a word aligned address and access a four-byte frame from memory which is then operated on based on a `frame_mask` for half-word and byte operations. Designed with the

influence of Harvard architecture, the phoeniX native memory interface ensures the elimination of structural hazard occurrences while accessing memory. It incorporates two distinctive address and data buses, specifically dedicated to instructions and data. As can be seen from the top module's port instantiations, both these memory interfaces for instruction have a data, address and control bus. Data bus related to data memory interface is bi-directional and therefore defined as `inout` net type while the data bus for instruction memory interface is uni-directional and is considered as an `input` from the processor's point of view.

Frame mask is used for defining the target bytes, half-words or words which should be loaded or stored in memory. The frame_mask signal can take the following values based on the word-size needed for a particular memory access:

- `4'b1111`: for all 32-bit accesses with word-aligned addresses.
- `4'b1100, 4'b0011`: for all 16-bit accesses with half-word-aligned addresses.
- `4'b1000, 4'b0100, 4'b0010, 4'b0001`: for all 8-bit accesses with byte-aligned accesses.

Address	Memory	Word Aligned	Half-Word Aligned		Byte Aligned			
------00		1	1	0	1	0	0	0
------01		1	1	0	0	1	0	0
------10		1	0	1	0	0	1	0
------11		1	0	1	0	0	0	1

Figure 9 - Frame Mask Values on different aligned memory accesses

Instruction Memory Interface Signals:

- `instruction_memory_interface_enable`: A wire signal indicating whether the instruction memory interface is enabled.
- `instruction_memory_interface_state`: A wire signal representing the state (*READ/WRITE*) of the instruction memory interface.
- `instruction_memory_interface_address`: A wire signal specifying the address for accessing instruction memory.
- `instruction_memory_interface_frame_mask`: A wire signal indicating the frame mask for the instruction memory interface.
- `instruction_memory_interface_data`: A wire signal for reading instruction data from the instruction memory.

Instruction Memory Interface Behavior:

The behavior of the instruction memory interface is described using an always block triggered by the negative edge of the `CLK` signal. It is important to note that phoeniX native memory interface does not support misaligned accesses. If the instruction memory interface is not enabled the data bus is set to high-impedance (`32'bz`).

If the instruction memory interface is enabled, and the state is *READ*, the `instruction_memory_interface_data` is assigned the value read from the `Memory` array based on the `instruction_memory_interface_address`.

Data Memory Interface Signals:

- `data_memory_interface_enable`: A wire signal indicating whether the data memory interface is enabled.
- `data_memory_interface_state`: A wire signal representing the state (*READ/WRITE*) of the data memory interface.
- `data_memory_interface_address`: A wire signal specifying the address for accessing data memory.

- `data_memory_interface_frame_mask`: A wire signal indicating the frame mask for the data memory interface.
- `data_memory_interface_data`: A wire signal for reading data from the data memory.
- `data_memory_interface_data_reg`: A register storing the data to be written to the data memory.

The memory interface logic structure and main behavior are similar for both the instruction memory and data memory. The data memory supports both store and load operations, allowing for flexible access to specific bytes, half words, and words based on the frame mask and target address. On the other hand, the instruction memory always loads a 32-bit data, encompassing the instructions stored within. Whether for instruction or data, the core always reads a 32-bit value from memory and operates the necessary functions on this value based on frame mask.

Data Memory Interface Behavior:

The behavior of the data memory interface is described using an always block triggered by the negative edge of the `CLK` signal. If the data memory interface is not enabled (`data_memory_interface_enable` is not set), the `data_memory_interface_data_reg` i.e., the data bus is set to high-impedance (`32'bz`).

If the data memory interface is enabled and the state value is *WRITE*, the data is written to the defined `Memory` array address based on the `data_memory_interface_address` and `data_memory_interface_frame_mask`. If the state value is equal to *READ*, the `data_memory_interface_data_reg` is assigned the value read from the `Memory` array based on the `data_memory_interface_address`.

The `data_memory_interface_data_reg` is set to high-impedance (`32'bz`) in a separate always block triggered by the positive edge of the `CLK` signal to avoid glitches.

Overall, this testbench provides the necessary logic to interface with the phoeniX design module's instruction and data memory interfaces, initializes memories, and monitors register values for debugging purposes. It also includes simulation control and finish condition detection.

In the end, it is crucial to emphasize that the memory interface logic is fully implemented within the testbench. Designing a large memory within the foundation modules of the processor is not a recommended approach due to synthesis limitations in both FPGA and ASIC designs. Moreover, such an approach can significantly impact power and area efficiency within the design. Instead, in the testbench, a memory space of 4MB is defined. This memory space serves also as storage for the firmware file, which contains the compiled and assembled instructions of a code. By adopting this methodology, the memory is effectively managed within the testbench while accommodating the necessary code instructions.

Chapter 4

4 Code Executant Software

"The purpose of software engineering is to control complexity,

not to create it."

Pamela Zave

In the preceding chapters, we delved into an exploration and elucidation of the hardware design and features of the phoeniX RISC-V core. Our upcoming chapter aims to provide a comprehensive explanation and description of the interface between the phoeniX hardware and software layers, as well as the procedural flow for executing C and assembly codes on the processor.

Prior to getting into the flow explanation, it is important to note that the execution and simulation process on the phoeniX core requires several essential software applications. These applications, along with their respective purposes, were comprehensively detailed in the introductory chapter of this document. As previously mentioned, within the `Setup` directory of the phoeniX core repository, a shell script has been provided to streamline the installation process of the required software mentioned earlier. This script serves to automate the installation, ensuring a more convenient and efficient setup experience for users.

4.1 Code Execution Flow

The first important tool which is used in this process is RISC-V GCC Compiler Toolchain. You can use the scripts provided in the original RISC-V repositories and riscv-tools [9]. The default settings in the original repos build-scripts will build a compiler, assembler and linker that can target any RISC-V ISA. You can also use the provided shell script in `Setup` directory. All shell scripts and Makefiles provided in this repository target Ubuntu 20.04 unless otherwise specified. Simply run the `setup.sh` from your terminal, it will automatically install the required perquisites, iverilog version 12 and gtkwave.

```
user@Ubuntu:~$ git clone https://github.com/phoeniX-Digital-Design/phoeniX.git
user@Ubuntu:~$ cd Setup
user@Ubuntu:~$ chmod +x setup.sh
user@Ubuntu:~$ ./setup.sh
```

Using your favorite editor open `.bashrc` file from the home directory of your Ubuntu. Replace `{user}` with your own username and add the following lines to the end of file. This will your `PATH` environment variable and is required to run RISC-V GNU Compiler automatically without exporting `PATH` variable each time.

Note: The script provided setup.sh and the following lines are set configure the toolchain based on 8.3.0 version of the compiler and toolchain. If you wish to install a different version, please beware and change the required lines in `setup.sh` and the following lines:

```
export PATH=/home/{user}/riscv_toolchain/riscv64-unknown-elf-gcc-8.3.0-2019.08.0-x86_64-linux-ubuntu14/bin:$PATH
export PATH=/home/{user}/riscv_toolchain/riscv64-unknown-elf-gcc-8.3.0-2019.08.0-x86_64-linux-ubuntu14/riscv64-unknown-elf/bin:$PATH
```

Upon completion of the `setup.sh` process, the RISC-V compiler toolchain, iverilog V12.0, and gtkwave will be installed on the system. This comprehensive installation equips users with the

necessary tools to execute C and assembly codes on the phoeniX processor and analyze the resulting outputs. These simulation tools enable users to effectively evaluate the workflow and outcomes of their code execution.

4.2 Windows (AssembleX Software)

We have meticulously developed a sophisticated, lightweight, and user-friendly software solution with the help of Python. Our innovative software, "AssembleX," has been crafted to cater to the specific needs of Windows systems, enabling seamless execution of assembly code on the phoeniX processor. This tool significantly enhances the efficiency and effectiveness of the code execution process, offering a streamlined experience for users seeking to enter the realm of assembly programming in a very simple way.

AssembleX stands as a testament to our commitment to delivering excellence in software interface of the embedded phoeniX processor. It has been thoughtfully designed and carefully integrated into the core repository of the phoeniX core. This inclusion ensures easy access and availability to users, providing them with a comprehensive environment for executing assembly programs.

With its intuitive interface and robust functionality, AssembleX simplifies the process of assembly code execution. It offers a seamless and efficient workflow, allowing users to compile and run assembly programs effortlessly using Venus simulator in Microsoft Visual Studio Code and then turning the output in the suitable firmware format in order to be executed on phoeniX core.

AssembleX provides developers, enthusiasts, and researchers alike with a simple and user-friendly toolset to explore the capabilities of the phoeniX processor. We remain dedicated to continuous improvement, actively nurturing and expanding the functionality and features of AssembleX.

Venus is a highly regarded RISC-V instruction set architecture (ISA) simulator. It is designed to provide a platform for users to simulate and experiment with RISC-V assembly code and programs. Venus offers a valuable learning and development environment for students, researchers, and enthusiasts interested in understanding and exploring the RISC-V architecture.

The simulator is written in Java, making it platform-independent and easily accessible on various operating systems. It provides a graphical user interface (GUI) that enables users to interact with

the simulator, load and execute RISC-V assembly programs, and observe the program's execution step by step, by a Visual Studio extension.

One of the key features of Venus is its support for various RISC-V instruction set extensions, including the standard RV32I (desired extension for phoeniX V0.1) and RV64I base instruction sets, as well as optional extensions like M (Integer Multiplication and Division), A (Atomic), F (Single-Precision Floating-Point), and D (Double-Precision Floating-Point). This flexibility allows users to experiment with different instruction set configurations and explore the functionalities provided by each extension.

The workflow of the processor is integrated with the help and benefits of Venus Simulator extension on well-known code editor, Visual Studio Code. In the beginning you'll have write and simulate your assembly codes by Venus Simulator. Debug interface of Visual Studio will help you monitor registers, floating point registers, memory and etc. of the assembly code you wrote.

After you simulated the code, in the VS-Code debug panel, select `VENUS OPTIONS > VIEWS > Assembly`. A new tab will pop up in the Visual Studio Code editor including full instructions (with pseudo code replacements and etc.), program counter and hex format of the instructions which is crucial for the firmware of the phoeniX core.

Save the new tab as a text file in the right directory (User Codes or Assembly Sample Codes). This file will be the input for the AssembleX script to create the firmware file. Before we explain the commands for execution process, it is good to know about the structure of AssembleX.

Here's a breakdown of the code in the following section.

Importing Required Modules:

- The `os` module is imported to interact with the operating system.
- The `sys` module is imported to access command-line arguments.
- The `glob` module is imported to retrieve file paths using pattern matching.

Setting Variables:

- `testbench_file` is set to the name of a testbench file.
- `option` is set to the first command-line argument passed to the script.
- `project_name` is set to the second command-line argument passed to the script.
- `output_name` is set by concatenating `project_name`, "`_firmware`", and "`.hex`" to indicate the format of the firmware file.

Handling Options:

- If option is "`sample`", the directory variable is set to "`Sample_Assembly_Codes`".
- If option is "`code`", the directory variable is set to "`User_Codes`".
- If option is neither "`sample`" nor "`code`", a `Value Error` is raised.

File Path Operations:

- The script searches for a specific file in the "`Software`" directory, based on the directory and `project_name` variables. The file is identified using the '`*.txt`' pattern.
- The `input_file` is set to the path of the found file.
- The `output_file` is set to the path of the output file, based on the directory, `project_name`, and `output_name` variables.

Reading and Modifying Input File:

- The script reads the contents of the `input_file` line by line.
- The first and third columns are removed from each line, leaving only the assembly code.
- The "`0x`" prefix is removed from each line.
- The resulting modified lines are stored in the `final_hex_code` list.

Writing Modified Contents to Output File:

- The modified content stored in `final_hex_code` is written to the `output_file`.
- Changing Firmware in the Testbench File:
- The script reads the contents of the `testbench_file` line by line.

- If a line starts with "`define FIRMWARE`," it is modified to include the `output_file` path.
- The modified content is written back to the `testbench_file`.

Executing Verilog Simulations:

- The script uses the `os.system()` function to execute Verilog simulation commands via the command prompt.
- First, it compiles the Verilog files using iverilog.
- Then, it runs the compiled simulation using `vvp`.
- Finally, it opens the simulation waveform using `GTKWave`.

Cleaning Up the Testbench File (Reset Testbench):

- The script opens the `testbench_file` again.
- If a line starts with "`` `define FIRMWARE `` " it is modified to remove the firmware file address.
- The modified content is written back to the `testbench_file`.

Overall, this code is helping to automate the process of processing assembly code files, extracting relevant information, modifying files, and executing Verilog simulation on the phoeniX processor.

→ Running Sample Codes:

The directory `Software` contains sample codes for some conventional programs and algorithms in Assembly, which can be found in `Sample_Assembly_Codes` directory.

PhoeniX convention for naming projects is as follows; The main source file of the project is named as `{project.s}` for sample assembly codes. This file along other required source files are kept in one directory which has the same name as the project itself i.e., `project`.

Sample projects provided at this time are `bubble_sort`, `fibonacci`, `find_max_array`, `sum1ton`.

Before running the script, note that the assembly output of the Venus Simulator for the code must be also saved in the project directory as it was explained previously.

To run any of these sample projects simply run `python AssembleX.py sample` followed by the name of the project passed as a variable named project to the Python script.

The input command format for the terminal follows the structure illustrated below:

```
python AssembleX.py sample {project_name}
```

For example:

```
python AssembleX.py sample fibonacci
```

After execution of this script, firmware file will be generated and this final file can be directly fed to our Verilog testbench. AssembleX automatically runs the testbench and calls upon gtkwave to display the selected signals in the waveform viewer application, gtkwave.

→ Running Your Own Codes:

In order to run your own code on phoeniX, create a directory named to your project such as `/my_project` in `/Software/User_Codes/`. Put all your `.s` files in `my_project` and run the following command from the main directory:

```
python AssembleX.py code my_project
```

Provided that you name your project sub-directory correctly the AssembleX software will create `my_project_firmware.hex` and fed it directly to the testbench of phoeniX processor. After that, iverilog and GTKWave are used to compile the design and view the selected waveforms.

4.3 Linux (RISC-V GCC toolchain)

We have developed an automated software process and simulation for the phoeniX processor, designed to be user-friendly and easily accessible on Linux systems. To streamline the simulation process, we have included a Makefile in the main repository directory. This Makefile simplifies the previously explained lengthy process of simulation with the RISC-V GCC toolchain.

In contrast to the Windows solution, which was limited to running assembly codes on the phoeniX core due to the complexities of setting up a RISC-V GCC Toolchain on this operating system, our Makefile offers broader functionality. It enables the execution of both C codes and assembly codes using the original RISC-V compiler and assembler. This enhanced capability provides greater flexibility for developers working with the phoeniX processor.

Prior to getting into the details of the Makefile, it is essential to understand the perquisite dependencies that the GCC compiler toolchain necessitates for executing the user's code on the processor. In addition, the repository incorporates a directory called `Firmware` that encompasses essential scripts crucial for executing C or assembly code on the processor using the RISC-V GCC compiler toolchain. It is imperative to refrain from removing any scripts within the "Firmware" directory, as the Makefile relies directly on these scripts. However, it is possible to make alterations to these scripts to suit the user's application requirements, if necessary.

Preserving the integrity and functionality of the Makefile ensures seamless utilization of the scripts for executing C or assembly code on the processor, while allowing flexibility for customization as per the user's specific needs. The following section provides a detailed description of each script within the `Firmware` directory:

`syscall.c`:

This C code is implementing a minimalistic set of system calls and related functions. Let's break down the code and describe each section:

Header Includes:

- `sys/stat.h`: Provides the necessary data structures and function declarations related to file status and information.
- `unistd.h`: Contains function declarations and constants related to system calls, file access, and standard I/O.
- `errno.h`: Defines the global integer variable `errno` and provides error code definitions.

Macro Definition:

- `UNIMPL_FUNC(_f):` This macro defines a string representation of a function name, preceded by `.global`, `.type`, and the function label. It is used to declare and define each unimplemented system call as a global function.

Assembly Code:

- The following assembly code section defines each of the unimplemented system calls as global functions. Each function is labeled using the macro `UNIMPL_FUNC(_f)`.

`unimplemented_syscall()` Function:

- This function is called when an unimplemented system call is invoked. It outputs a message indicating that an unimplemented system call was called, writes the message to a specific memory address (0x10000000), and triggers a breakpoint (`ebreak`). The function is marked with the `__builtin_unreachable()` attribute, indicating that it should never be reached.

`_read()` Function:

- This function is an implementation of the `_read` system call. It returns 0, indicating an end-of-file (`EOF`) condition.

`_write()` Function:

- This function is an implementation of the `_write` system call. It writes the contents of the provided buffer (ptr) to a specific memory address (0x10000000) and returns the length of the written data.

`_close()` Function:

- This function represents an implementation of the `_close` system call. It returns 0, indicating a successful close operation. It is called before the `_exit()` function.

`_fstat()` Function:

- This function is an implementation of the `_fstat` system call. It sets the `errno` variable to `ENOENT` (indicating "No such file or directory") and returns -1, indicating an error.

`_sbrk()` Function:

- This function represents an implementation of the `_sbrk` system call, which is used for dynamic memory allocation. It updates the `heap_end` variable with the requested increment and returns a pointer to the allocated memory.

`_exit()` Function:

- This function is called to terminate the program. It triggers a breakpoint (`ebreak`) and is marked with the `__builtin_unreachable()` attribute, indicating that it should never be reached.

Overall, this code provides minimal implementations of various system calls required for basic functionality. It serves as a placeholder or stub implementation that can be used for testing or in situations where a full-fledged operating system or underlying system libraries are not available.

`start.s`:

The logic and functionality of the code can be summarized as follows:

Register Initialization:

- All general-purpose registers (`x1` to `x31`) are set to zero. This ensures that they start with a known and consistent value.

Stack Setup:

- The stack pointer (`sp`) is set to a specific memory address, allocating a stack size of 4 MB.
- Zeros are pushed onto the stack to reserve space for `argc` and `argv`.

Jump to C Library Initialization:

- The code jumps to the `_ftext` symbol, which represents the start of a C library initialization routine. This allows the program to continue its execution from that point onwards.

The code sets up the initial state of the program by zero-initializing registers, establishing the stack, and preparing for subsequent C library initialization. The specifics of the C library initialization and the overall program functionality are not provided in this code snippet. In the execution process, the assembled form of this code will connect to the main body of the program in the beginning of the firmware.

`start.ld`:

This linker script provides instructions for the linker, which is responsible for combining object files and generating the final executable or binary. Let's break down the linker script:

`. = 0x00000000`:

- This directive sets the current address (.) to `0x00000000`, indicating the starting address of the memory region being defined.

`.text : { *(.text) }`:

- This directive defines a memory region called .text.
- The {} braces enclose the content that will be placed in the .text memory region.
- `*(.text)` specifies that all sections with the name .text should be included.
- The * symbol matches any number of occurrences, meaning that all sections named .text from the object files will be placed in this memory region.

`_ftext = 0x00010000`:

- This line assigns the value `0x00010000` to the symbol `_ftext`.
- The symbol `_ftext` typically represents the start address of the program code region.
- By assigning a specific value to `_ftext`, it can be used as a reference point in the code.

Overall, this linker script defines a memory region called .text and includes all sections with the name .text from the object files. It also assigns a specific value (`0x00010000`) to the symbol `_ftext`, which can be used to reference the start address of the program code. The linker will use this script to arrange the sections and generate the final executable or binary file.

`riscv.ld`:

The linker script for RISC-V 32-bit architecture is responsible for organizing the different sections of an executable in memory. It provides instructions that dictate how the sections should be arranged and accessed by the program at runtime.

The script includes sections such as `.text` for executable code, `.init` for initialization code executed before the program starts, and `.fini` for cleanup code executed when the program exits. It also includes sections like `.rodata` and `.rodata1` for read-only data, `.data` and `.data1` for initialized data, and `.bss` for uninitialized data.

Additionally, the linker script defines sections related to dynamic linking, such as `.plt` and `.iplt` for the Procedure Linkage Table (PLT) and the Indirect Procedure Linkage Table (IPLT). These tables are used for resolving function calls to external libraries or shared objects at runtime.

Other sections in the linker script handle exception handling, thread-local storage, constructor and destructor functions for C++ programs, debugging information, data relocation, and more. These sections ensure that the program is properly linked and can execute correctly by organizing the memory layout and providing necessary runtime support.

In summary, the linker script for RISC-V 32-bit architecture outlines the logical organization of sections within an executable, enabling proper linking and execution of the program.

`hex_convertor`:

This Python script takes an input file, which is typically the output of the RISC-V GCC compiler toolchain, and processes it to generate a Verilog standard hex file representing the instruction memory for the phoeniX core.

Here's a breakdown of the script:

- It imports the `fileinput` and `itertools` modules.
- It initializes variables `ptr` and data (list to store the instruction data).
- The `write_data()` function is defined. It checks if there is any data in the data list, and if so, it writes the data to the output. The data is formatted as hexadecimal words, with each word representing 4 bytes of instruction data.
- The script iterates over each line in the input file using `fileinput.input()`.
- If a line starts with `@`, it indicates a change in the memory address. The script converts the address to an integer and checks if it is greater than the current `ptr + 4` (indicating a gap in memory). If so, it calls the `write_data()` function, updates the `ptr` to the new address, resets the data list, and adds padding zeros to align the memory address to a multiple of 4.
- If a line does not start with `@`, it assumes it contains instruction data in hexadecimal format. The script splits the line into tokens, converts each token to an integer, and appends them to the data list.
- After processing all the lines, the script calls the `write_data()` function to flush any remaining data.
- The resulting hex file, representing the instruction memory, is printed to the console.

Overall, this script reads the output file of the GCC toolchain processed object files and then processes the memory addresses and instruction data, and outputs a readable hex file that represents the instruction memory of the phoeniX processor.

Now that we have gained an understanding of the purpose and function of each script in the `Firmware` directory, we are prepared to delve into the details of the Makefile specifically crafted to facilitate code execution and simulation on the phoeniX processor.

This Makefile orchestrates a sequence of instructions required to execute C or assembly code on the phoeniX processor. The process begins by compiling the C code, along with the `syscall.c` and `start.s` scripts, using the RISC-V GCC compiler toolchain. These individual components are then linked together using the linker scripts provided in the `Firmware` directory, as previously described. The resulting output is a firmware file that undergoes conversion into a format compatible with the phoeniX processor's instruction memory using the `hex_convertor.py` Python script. Ultimately, this Makefile streamlines the compilation, linking, and preparation of the firmware to seamlessly enable code execution on the phoeniX processor. Next step after preparing the firmware to be executed is giving the created file to the testbench of phoeniX processor which takes responsibility for memory interfaces in this processor simulation flow. The firmware file will be given to the testbench and it will be executed on the core using iVerilog commands in the Makefile. In the end, GTKWave will open the waveform of the processor executing the given codes. You can see a complete view of the data flow in the processor and the internal signals while it is executing the C and assembly code.

→ Running Sample Codes:

The directory `Software` contains sample codes for some conventional programs and algorithms in both Assembly and C which can be found in `Sample_Assembly_Codes` and `Sample_C_Codes` sub-directories respectively.

The phoeniX processor convention for naming projects is as follows; The main source file of the project is named as {`project.c`} or {`project.s`}. This file along other required source files are kept in one directory which has the same name as the project itself i.e., `project`.

Sample projects provided at this time are `bubble_sort`, `fibonacci`, `find_max_array`, `sum1ton`.

To run any of these sample projects simply run `make sample` followed by the name of the project passed as a variable named project to the Makefile.

The input command format for the terminal follows the structure illustrated below:

```
make sample project={project}
```

For example:

```
make sample project=fibonacci
```

Provided that the RISC-V toolchain is set up correctly, the Makefile will compile the source codes separately, then using the linker script riscv.ld provided in Firmware directory and it links all the object files necessary together and creates `firmware.elf`. It then creates `start.elf` which is built from `start.s` and `start.ld` and concatenate these together and finally forms the `{project}_firmware.hex`. This final file can be directly fed to our Verilog testbench. Makefile automatically runs the testbench and calls upon gtkwave to display the selected signals in the waveform viewer application, GTKWave.

→ Running Your Own Codes:

In order to run your own code on phoeniX, create a directory named to your project such as `/my_project` in `/Software/User_Codes/`. Put all your `.c` and `.s` files in `my_project` and

```
make code project=my_project
```

run the following make command from the main directory:

Provided that you name your project sub-directory correctly and the RISC-V Toolchain is configured without any troubles on your machine, the Makefile will compile all your source files separately, then using the linker script `riscv.ld` provided in `Firmware` it links all the object files necessary together and creates firmware.elf. It then creates start.elf which is built from `start.s` and `start.ld` and concatenate these together and finally forms the `my_project_firmware.hex`. After that, iverilog and GTKWave are used to compile the design and view the selected waveforms.

→ Further Configurations:

The default testbench provided as phoeniX_Testbench.v is currently set to support up to 4MBytes of memory and the stack pointer register `sp` is configured accordingly. If you wish to change this, you need configure both the testbench and the initial value the `sp` is set to in `/Firmware/start.s`. If you wish to use other specific libraries and header files not provided in `Firmware` please beware you may need to change linker scripts `riscv.ld` and `start.ld`.

Chapter 5

5 Synthesis Result

"But life at its best is a creative synthesis of

opposites in fruitful harmony."

Martin Luther King Jr.

This chapter provides a comprehensive overview of the synthesis process for the phoeniX core, leading to the creation of the processor layout. The RTL design has undergone a complete flow in order to create a physical layout design for this 32-bit RISC-V core, using open-source tools which will be fully described in this chapter.

Before getting into the assessment of the physical design's results, it is crucial to acknowledge that this particular design is capable of synthesis for FPGA targets as well. The code has been carefully crafted to enable the utilization of the processor as a soft-core on Xilinx FPGA devices. While the core is entirely synthesizable, it is important to note that implementation is limited to Xilinx Ultrascale and Ultrascale+ series of AMD Xilinx FPGA devices, owing to the processor's size requirements.

In fact, the phoeniX core can be implemented as a softcore CPU on Xilinx 7 Ultrascale/Ultrascale+ series FPGA boards using logic synthesis. This allows for flexible integration of the core's functionality within the FPGA fabric. The Xilinx 7 series FPGA boards provide a versatile platform for hosting the softcore CPU implementation, offering configurable features and adaptability.

Using soft-core processors on FPGA devices offers numerous advantages in project development. Firstly, it provides the flexibility to incorporate processor functionalities into the FPGA design, allowing for efficient integration of software and hardware components. Soft cores enable the execution of complex algorithms and control functions, which are traditionally handled by general-

purpose processors, directly within the FPGA fabric. This eliminates the need for external microcontrollers or additional components, simplifying the overall system architecture.

Furthermore, soft-core processors offer significant customization opportunities. Designers can tailor the processor's architecture, instruction set, and peripherals to suit the specific project requirements. This level of customization enables optimized resource utilization, improved performance, and reduced power consumption. Additionally, soft cores can be easily updated or modified, allowing for iterative design improvements without the need for hardware changes.

Using soft-core processors on FPGA devices empowers designers to create highly integrated and versatile systems, combining the benefits of hardware acceleration and software programmability. It offers a scalable and cost-effective solution, particularly in applications that demand real-time processing, rapid prototyping, or system-on-chip (SoC) integration.

In the following sections, we will delve into the intricate details of the ASIC design and physical design processes employed for the phoeniX processor. Notably, we will focus on the utilization of the open-source software toolchain, Qflow [10], which plays a pivotal role in facilitating these processes.

5.1 Qflow toolchain

This section aims to provide an extensive overview and analysis of Qflow, a popular open-source VLSI (Very Large Scale Integration) design tool. Qflow is a complete design flow that encompasses various stages of the VLSI design process, including synthesis, placement, routing, and verification. The report delves into the key features, functionalities, and advantages of Qflow, while also discussing its limitations and potential areas for improvement. Furthermore, it highlights real-world applications and provides insights into the future prospects of this tool.

The field of VLSI design plays a crucial role in the development of integrated circuits (ICs) and electronic systems. With the ever-increasing complexity of ICs, the demand for efficient and reliable VLSI design tools has grown significantly. Qflow, an open-source tool, has emerged as a popular option due to its comprehensive design flow and ease of use. This section explores the various aspects of Qflow and its impact on the VLSI design community.

Qflow is a complete tool chain for synthesizing digital circuits starting from Verilog source and ending in physical layout for a specific target fabrication process. In the world of commercial electronics, digital synthesis with a target application of a chip design is usually bundled into large EDA software systems like Cadence or Synopsys. As commercial electronics designers need to maintain cutting-edge performance, these commercial toolchains get more and more expensive, and have largely priced themselves out of all but the established integrated circuit manufacturers. This leaves an unfortunate gap where startup companies and small businesses cannot afford to do any sort of integrated circuit design. The algorithms for digital synthesis are all tied up in closed-source software, and development is controlled by a few EDA software vendors.

After a thorough investigation of alternatives by Qflow development team, two good choices for front-end synthesis have been selected: The first is a combination of the open-source Verilog parser Odin-II and optimizer/mapper “abc” to anchor the digital synthesis flow. They have the advantage of being part of an existing open-source FPGA synthesis flow called “vtr” ("Verilog-to-routing"). The second choice is the extraordinarily capable Yosys. This is a synthesis tool that can handle every type of Verilog-2005 syntax and can synthesize large projects off of “Open Cores”. Due to its versatility and capability, Yosys has been chosen as the default frontend. Odin-II and ABC can be specified as alternative frontends.

Digital standard cell libraries are a major component of the flow. While synthesis tools can make use of proprietary digital standard cells provided by various vendors (usually the fabrication facility, such as X-Fab, or IBM, or TSMC, but sometimes by 3rd-party vendors), it should be noted that proprietary standard cell libraries cannot be distributed and therefore cannot be used for examples or posted on public websites. Fortunately, there are a few sets of open-source standard cells available for popular processes. Some of these are based on the "scalable CMOS rules" from MOSIS, a set of design rules that are more conservative than the vendor rules and are allowed by the vendors to be distributed openly. Physical layout generated using the scalable CMOS rules can be distributed as open-source and can be fabricated in the processes for which the rules were designed. For the first distribution of Qflow, the open-source standard cell [11] library from Oklahoma State University (OSU) is being used. In the near future, there are plans to expand the distribution to include support for the open-source libraries from VLSI Technology (vlsitechnology.org), as well as the rest of the OSU standard cell libraries.

After mapping onto a standard cell library, a design needs to go through the placement and routing stages. The placement stage determines a rough estimate of the required routing and tries to put all the cells into a block, ordering them to minimize the total amount of wiring connecting all the pins together. A professional-grade placement tool known as “graywolf”, which was developed (under the name "TimberWolf") at Yale University, is used for placement. The last open-source version of this tool does not perform detailed routing but serves as an excellent placement tool.

The final step in creating a digital standard-cell layout is the detailed routing, which describes exactly how the physical wiring should be generated to connect all the pins in the design. An open-source detail router has been missing from the toolchain, but recognizing the need for an open-source digital synthesis tool flow for chip design, a moderately capable detail router called “Qrouter” has been developed, making it the final link in the open-source synthesis chain.

It should not be assumed that the Qflow toolchain can be used to create the next generation of multi-gigahertz microprocessors. However, the Qflow toolchain is perfectly capable of handling digital subsystems needed by many chips, including host-to-device communications (SPI and I2C, for example), signal processing (digital filters, sigma-delta modulators), arithmetic logic units, and more. Early versions of the Qflow digital flow have been used to create digital circuits used in high-performance commercial integrated circuits.

One of the verified processors designed using Qflow is the Raven microprocessor [12]. It is a RISC-V processor designed using Qflow version 1.3 on the efabless.com "Open Galaxy" (CentOS) platform where it was set up for the X-Fab XH018 fabrication process.

This small 32-bit processor was designed for embedded mixed-signal applications and has a 100MHz internal clock. The RISC-V processor core has over 20,000 gates, synthesized by Yosys, placed and routed by graywolf and Qrouter, and validated by vesta static timing analysis. It was a first-time silicon success triumph for open-source EDA tools. In the end the microcontroller is packaged in a 48-pin QFN (leadless) package.

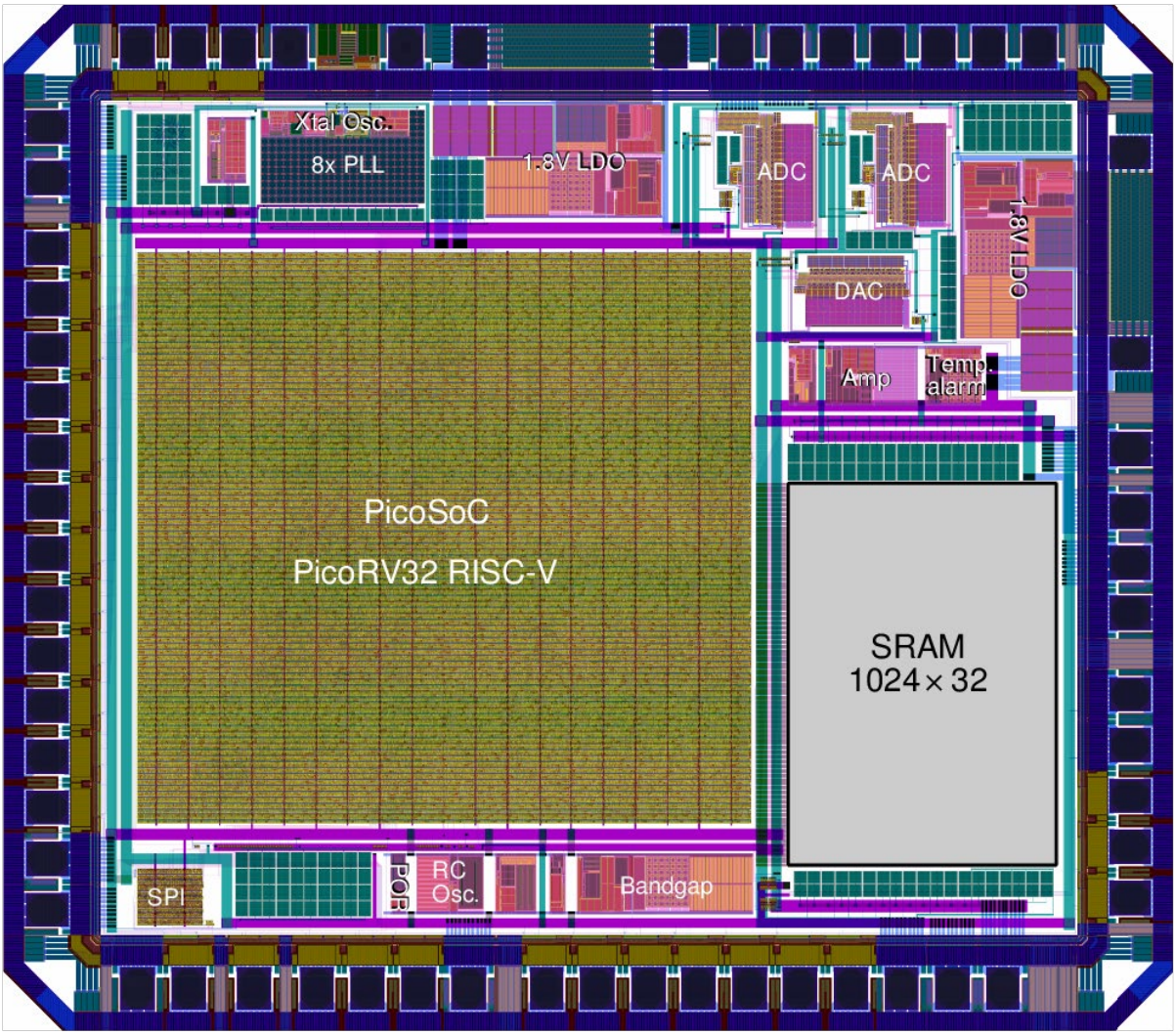

Figure 10 - The Raven RISC-V microprocessor from efabless [12]

The digital core was placed and routed by Qflow and verified with vesta; the top-level assembly was done with Magic VLSI [13] and verified with Netgen [14].

Qflow itself is a framework for managing a digital synthesis flow. It contains a number of tools unique to itself, but also relies on a number of tools that must be obtained elsewhere.

Required components of Qflow:

Note that Qflow can make use of both front-end synthesis tools, Yosys and Odin_II [15], but only one of them is mandatory, and Yosys is the preferred front-end. abc is used by both Yosys and Odin_II, but different versions need to be compiled for each tool; see details below.

Qflow:

- The Qflow package contains all the scripts and most of the tools necessary for the open-source digital synthesis flow. It also comes with some of the files from the OSU (Oklahoma State University) 0.35um standard cell library, to provide a default technology.

Verilog source:

- Most of the target applications on Open Cores make extensive use of all the features of Verilog-2005; they are all easily handled by Yosys. The alternative Odin-II frontend will usually fail to synthesize the Open Cores sources without some editing of the source code.

Digital standard cell technology

- For purposes of experimenting with the flow, Qflow provides a set of files for the OSU 0.35um open-source standard cell set. This is an excellent source for LEF and GDS standard cells. The cells are all compatible with the MOSIS SCMOS rules for the various processes available through MOSIS (mostly TSMC and AMI, 0.18um to 0.5um). The standard cell set is available from `vlsiarch.ecen.okstate.edu` and is available for download, free of charge.
- You can see the Reference page for instructions on how to use Qflow with a different technology.

The important components to have (all of which are in the OSU standard cell sets) are:

- A LEF-format file with all of the macros defined
- A LEF-format file describing routing resources
- A GDS file with all of the standard cell layouts
- Timing parameters, preferably in Liberty format
- A compatible technology file for Magic

Verilog parser, high-level synthesis, and logic optimization and verification (Yosys):

- This extraordinarily capable Verilog parser and logic synthesizer was written by Clifford Wolf. It is an industrial-grade tool that will synthesize everything from decoders up to microprocessors with ease and accuracy. The code can be obtained from the Yosys website.

- Note that Yosys makes use of the logic optimizer “abc”, but will download and compile the appropriate version as part of its make script.

Verilog parser and logic verification (Odin-II)

- This Verilog parser, written by Peter Jamieson, was introduced at FCCM 2010 and continues to undergo development as part of the FPGA synthesis tool chain VTR. Obtain Odin-II from Google Code. You may want to follow the link to the VTR distribution (also see below), which may contain the most recent code base for Odin-II.

Logic optimization (abc):

- This tool takes the BLIF format description and creates a netlist representation using a set of standard cells described in the "genlib" format, performing logic optimization along the way. Obtain this tool from EECS at U.C. Berkeley [16].

The complete FPGA synthesis flow vtr contains both Odin-II and abc. In general, this is the best place to get the most recent sources for both Odin-II and abc.

Note that Yosys also uses abc for combinatorial logic optimization, but Yosys will automatically download and compile the correct version of abc with the compile-time options needed by Yosys.

Placement (graywolf):

- Obtain this tool from GitHub graywolf [17].
- Graywolf is the placement tool formerly known as TimberWolf. It is currently maintained by Ruben Undheim in a much-streamlined form.

Detail Router (Qrouter 1.4):

- Obtain Qrouter 1.4 from OpenCircuitDesign: Qrouter home page [18]
- This is a work in progress, but is a full-featured multi-layer, sea-of-gates maze router. For a very long time, a good open-source maze router was not available anywhere, and it had been on my list of important things to do for ages. Finally, in June of 2011, I finally sat down and wrote it. Although as a starting point it was based on router code by Steve Beccue, it is totally overhauled and can be considered written from scratch for all practical

purposes. It is a command-line-only tool, taking standard format (LEF and DEF) files as input and generating an annotated DEF format file as output.

Static Timing Analysis (vesta):

- Vesta is a static timing analysis tool that is part of the Qflow package.

Layout viewer (Magic 8.1):

- Magic VLSI Tool is an open-source software used for designing and analyzing digital integrated circuits. It offers a user-friendly interface, supports various layout formats, and provides features for layout editing, design rule checking, simulation, and extraction. It is widely used in the VLSI community and is popular for its flexibility and extensive capabilities.

By harnessing the capabilities of these powerful tools, Qflow empowers users to seamlessly navigate the entire process of digital integrated circuit design, encompassing diverse fabrication technologies, from an HDL source to comprehensive physical design implementation.

5.2 TSMC 180nm PDK

The OSU Open Cell Standard Library is an open-source library specifically designed for VLSI (Very Large Scale Integration) processes. It provides a comprehensive set of standard cell components that are essential building blocks for digital circuit designs.

The library is developed and maintained by researchers and engineers at Oklahoma State University (OSU). It is intended to be used by designers, researchers, and students involved in VLSI design projects, enabling them to access a reliable and openly available set of standard cells.

The OSU Open Cell Standard Library is based on the "scalable CMOS rules" provided by MOSIS (Metal Oxide Semiconductor Implementation Service). These rules are design guidelines that are more conservative than vendor-specific rules, allowing for greater compatibility across different VLSI fabrication processes. The library adheres to these rules, ensuring that the physical layout

generated using the library is compliant and can be fabricated in the processes for which the rules were designed.

The library includes a wide range of standard cell components, such as basic logic gates (AND, OR, XOR, etc.), flip-flops, multiplexers, decoders, and arithmetic units. These cells are optimized for performance, power consumption, and area efficiency, striking a balance between these key metrics for digital circuit design.

The OSU Open Cell Standard Library is distributed in a format that is compatible with popular VLSI design tools, allowing designers to seamlessly integrate the library into their design flows. It is typically used in conjunction with synthesis tools to map high-level hardware descriptions (HDL) onto the library's standard cells, forming the foundation of the physical design.

As an open-source library, the OSU Open Cell Standard Library promotes collaboration, knowledge sharing, and innovation in the VLSI community. Designers can study and modify the library's cells to suit their specific design requirements, contributing to its ongoing improvement and expansion.

Overall, the OSU Open Cell Standard Library provides a valuable resource for VLSI designers, offering a collection of optimized standard cells adhering to scalable CMOS rules. Its availability as an open-source library promotes accessibility, flexibility, and collaboration in the field of VLSI design.

The OSU018 technology, also known as the TSMC 180nm process technology, is a widely adopted semiconductor fabrication process developed by TSMC (Taiwan Semiconductor Manufacturing Company). It belongs to the 180nm technology node, which refers to the minimum feature size or gate length of transistors produced using this process.

The OSU018 technology offers several key characteristics and capabilities that make it suitable for a range of digital integrated circuit designs. Here are some notable features:

Gate Length:

- The gate length in the OSU018 technology is approximately 180nm, which defines the size of the transistors and their switching speed.

Supply Voltage:

- The technology supports a typical supply voltage of 1.8 volts (hence the name 180nm). This voltage level is common for digital integrated circuits and facilitates compatibility with a wide range of system requirements.

Transistor Types:

- The OSU018 technology employs CMOS (Complementary Metal-Oxide-Semiconductor) transistor technology. CMOS transistors consist of both n-type and p-type MOSFETs (Metal-Oxide-Semiconductor Field-Effect Transistors) that provide low power consumption and high noise immunity.

Standard Cell Libraries:

- The OSU018 technology is associated with a specific set of standard cell libraries designed to be compatible with the process. These libraries contain essential building blocks, such as logic gates and flip-flops, enabling designers to implement digital circuits efficiently.

Design Rules:

- The OSU018 technology follows a set of design rules that define the constraints and guidelines for designing layouts to ensure manufacturability. These rules specify dimensions, spacing, and other factors critical for successful fabrication.

Performance and Power Trade-off:

- The OSU018 technology strikes a balance between performance and power consumption. It offers a reasonable level of performance for various digital circuit applications while keeping power consumption within acceptable limits.

The OSU018 technology has been widely utilized in various domains, including consumer electronics, telecommunications, automotive, and industrial applications. It provides a reliable and mature process for manufacturing digital integrated circuits with moderate complexity.

TSMC has always insisted on building a strong, in-house R&D capability. As a global semiconductor technology leader, TSMC provides the most advanced and comprehensive portfolio of dedicated foundry process technologies.

5.3 phoeniX Verification Results

The phoeniX RISC-V processor has successfully completed an extensive circuit integration workflow, employing the highly regarded Qflow toolchain as outlined and meticulously elucidated in section 5.2 of the documentation.

The core is meticulously crafted in Verilog, ensuring full compatibility with the standards embraced by the Qflow toolchain for robust hardware synthesis. The synthesis phase is proficiently executed through the utilization of Yosys.

The process was meticulously carried out using the renowned osu018 technology, which is based on TSMC's advanced 180nm process technology, developed in collaboration with Oklahoma State University.

Subsequent to the synthesis, placement, and routing stages, a thorough Static Timing Analysis (STA) was conducted on the design utilizing the vesta tool.

Static Timing Analysis (STA) is an essential step in the VLSI design process. It is a methodical approach to evaluate the timing behavior of a digital circuit under various conditions. STA analyzes the propagation delays of signals and ensures that the circuit meets the required timing constraints.

During STA, the tool analyzes the delays introduced by various components of the circuit, such as gates, interconnects, and flip-flops, to determine the worst-case timing scenario. It considers

factors like clock frequency, data arrival times, and setup/hold times to assess if the circuit will function correctly within the desired operating conditions.

STA helps identify potential timing violations, such as setup and hold violations, excessive delay paths, or violations of maximum operating frequency. It provides critical information to designers, enabling them to optimize the circuit's performance, meet timing requirements, and enhance overall functionality.

In summary, STA plays a crucial role in VLSI design by ensuring that the circuit functions correctly and reliably by accurately assessing and validating its timing characteristics.

Vesta STA tool employs a comprehensive analysis approach, meticulously examining a multitude of paths within the designed module. The tool meticulously assesses the timing characteristics of each path, enabling it to provide valuable insights into the maximum propagation delays observed throughout the design. By conducting an extensive path analysis, vesta facilitates a thorough understanding of the circuit's timing behavior, aiding designers in identifying critical paths and potential timing bottlenecks. Ultimately, the output generated by vesta offers valuable information regarding the maximum propagation delays, empowering designers to optimize the circuit's performance and meet stringent timing requirements.

The table presented below exhibits the outcomes of the static time analysis carried out for individual modules within the phoeniX core. In this particular design, conducting separate analysis and testing for each module holds utmost significance. This approach allows for a thorough examination of the delay characteristics specific to each module, as these delays directly impact the overall delay time of the pipeline stages. The cumulative effect of these delays ultimately defines the clock cycle time of the processor.

Module	Max Delay (ps)
Address Generator	3844.84
Arithmetic Logic Unit	3099.01
Control Status Registers	747.689
Hazard Forward Unit	1131.73
Immediate Generator	1016.44
Instruction Decoder	716.437
Jump Branch Unit	243.115
Register File	695.34
Normalized Memory Access Time	10000 - 40000
Fetch Unit	308.907
Load Store Unit	569.903

Table 2 - STA results of phoeniX core modules

It is worth noting that while Vesta may not encompass every possible path for analysis within the modules, it typically covers a significant number of paths that are valid and relevant. The reported numbers in the analysis results have been normalized, taking into account multiple iterations of the synthesis process conducted for the designs. This comprehensive approach ensures that the reported data accurately represents the performance characteristics of the modules and provides valuable insights for further optimization and refinement.

The analysis results indicate that the maximum delay observed in the core modules, and consequently in the pipeline stages, is approximately 4 nanoseconds. Based on this finding, it is necessary to establish a clock cycle time of 4 nanoseconds to accommodate the critical path and ensure proper operation. This implies a performance capability of approximately 250 MHz for the main phoeniX core.

Setting the clock cycle time at 4 nanoseconds allows for sufficient margin to account for the maximum delay across the modules, ensuring that data propagates through the pipeline within the specified time frame. By adhering to this timing requirement, the processor can achieve a performance level of approximately 250 MHz, enabling efficient execution of instructions and supporting the desired operational specifications.

It is crucial to note that selecting an appropriate clock cycle time is a key consideration in optimizing the overall performance of the core. By carefully analyzing the maximum delay and

configuring the clock cycle accordingly, designers can strike a balance between performance, timing constraints, and the specific requirements of the phoeniX core architecture and process design kit technology.

After static time analysis step, DRC step was taken for phoeniX core successfully. DRC, in the context of VLSI process, stands for Design Rule Checking. It is a crucial step in the semiconductor manufacturing process that involves verifying the compliance of a chip design layout with the specified design rules.

During DRC, the layout of the integrated circuit (IC) is compared against a set of predefined design rules. These rules define the minimum and maximum dimensions, spacing, alignment, and other geometric constraints that the layout must adhere to. Design rules ensure that the IC can be manufactured accurately and reliably, preventing issues such as short circuits, open circuits, and other manufacturing defects.

DRC checks for violations of the design rules by analyzing the layout data. It examines the geometric relationships between various components, layers, and interconnects to identify any deviations from the specified rules. DRC tools highlight violations and generate error reports to guide designers in resolving the issues.

By performing DRC, designers can ensure that their chip layout meets the required manufacturing constraints, improving the chances of successful fabrication and reducing the risk of costly rework or silicon failures. DRC plays a vital role in guaranteeing the manufacturability and reliability of integrated circuits in the VLSI design process.

The las step before final verification and exporting the GDS output file which contains complete layout design and information, is LVS. LVS, stands for Layout versus Schematic. It is a very important step in the semiconductor manufacturing process that involves verifying the consistency and accuracy between the chip's layout and its corresponding schematic design.

LVS ensures that the layout, which represents the physical implementation of the integrated circuit (IC), accurately matches the intended functionality described by the schematic design. It compares the geometric representation of the layout (including transistors, interconnects, and other components) with the logical connectivity and circuit elements defined in the schematic.

During LVS, specialized software tools analyze both the layout and the schematic representations of the IC. The tools examine the connectivity of the components, the consistency of node names, and the correspondence between the schematic elements and their physical counterparts in the layout. Any discrepancies or differences between the layout and the schematic are identified as LVS errors. These errors typically arise due to issues such as missing or extra connections, incorrect transistor sizes or orientations, and discrepancies in the interconnect routing. The LVS software generates error reports that help designers identify and rectify these inconsistencies, ensuring that the layout accurately reflects the intended circuit functionality.

By performing LVS, designers can validate the correctness and integrity of their chip design, minimizing the risk of functional errors or mismatches between the schematic and the physical implementation. LVS plays a critical role in ensuring the successful fabrication and functionality of integrated circuits in the VLSI design process.

In the end, the final GDS file will be generated which contains the complete layout of the processor. All of the outputs and log files of this processor verification process has been generated by different tools are included in the repository, Synthesis directory. The output layout files can be opened in both Magic VLSI and Klayout tools.

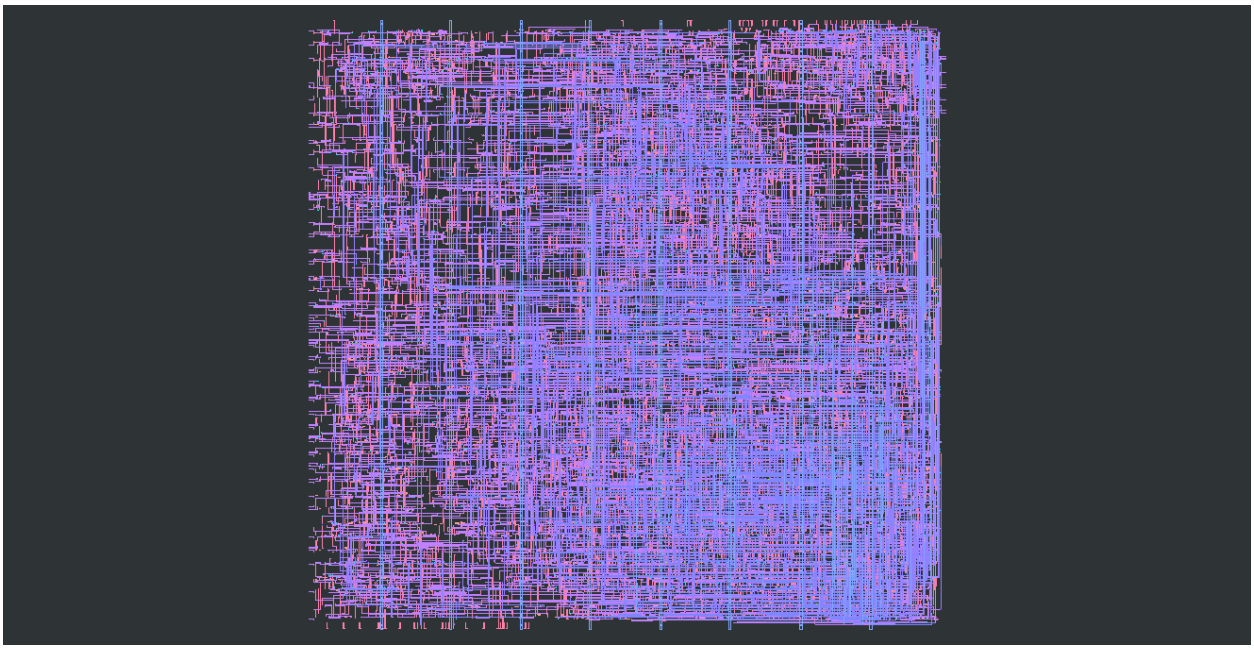

Figure 11 - phoeniX core .MAG layout in Klayout software

It is important to note that .MAG files are designed to be opened in Magic VLSI application but they can also be previewed in Klayout too (only preview is available in this format and cannot be edited in Klayout). GDS files can be opened, viewed and edited in both software applications.

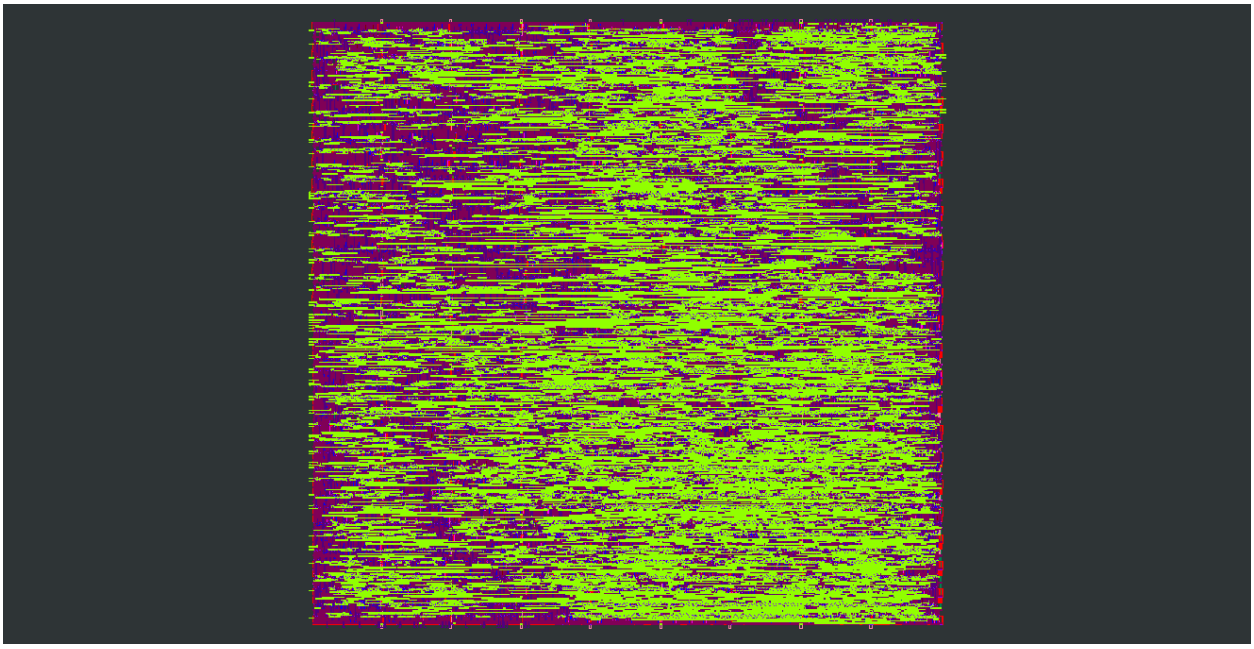

Figure 12 - phoeniX core .GDS layout in Klayout software

To investigate more details of the design you should install one of the mentioned software and open the design files in the repository.

In conclusion, this chapter has provided a comprehensive exploration of the phoeniX RISC-V processor, implemented in Verilog and utilizing the Qflow toolchain. By delving into the intricacies of synthesis and physical design, this chapter has shed light on the remarkable journey of transforming a high-level functional description into a physical reality.

The synthesis process which was done by Yosys has played a pivotal role in optimizing the phoeniX processor's design for performance, power, and area. Through the application of advanced algorithms and techniques, the Verilog code was transformed into a gate-level representation, ensuring that the resulting design meets the acceptable specifications.

The physical design phase, facilitated by the Qflow toolchain, has been instrumental in translating the synthesized design into a manufacturable layout. From floor planning to placement, routing, and verification, each step has been seamlessly integrated within the Qflow toolchain, streamlining the design flow and ensuring a robust physical implementation. The toolchain's ability to handle various design constraints and its comprehensive suite of tools have further empowered designers to achieve the desired performance targets while effectively managing power and area considerations.

The utilization of Verilog as the hardware description language has been pivotal in capturing the intricate behavior and structure of the phoeniX RISC-V processor. Verilog's flexibility, scalability, and compatibility with modern EDA tools have facilitated a smooth and efficient development process. Moreover, the widespread adoption of Verilog in the industry has ensured the availability of ample resources and support, further enhancing the feasibility of the design implementation.

The amalgamation of Verilog and the Qflow toolchain has exemplified the power of modern EDA methodologies, enabling the realization of a high-performance and efficient processor design. The insights gained from this chapter not only contribute to the body of knowledge in processor implementation but also pave the way for future advancements in the realm of digital design.

In conclusion, through meticulous synthesis optimization and a streamlined physical design flow, the phoeniX processor has successfully transitioned from a functional concept to a physical reality.

References

[1] “The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213”, Editors Andrew Waterman and Krste Asanovic, RISC-V Foundation, December 2019.

[2] Yosys Open-Source Projects: https://www.yosyshq.com/open-source

[3] Vesta Static Timing Analysis: http://www.maaldaar.com/index.php/vlsi-cad-design-flow/vesta

[4] C. Wolf. Yosyshq/picorv32: Picorv32 - a size-optimized risc-v cpu. Last Accessed: September 19, 2023. [Online]. Available: https://github.com/YosysHQ/picorv32

[5] lowRISC lowRISC/ibex: Ibex is a small 32 bit RISC-V CPU core, previously known as zero-riscy. Last Accessed: September 19, 2023. [Online]. Available: https://github.com/lowRISC/ibex

[6] riscv-collab/riscv-gnu-toolchain: GNU toolchain for RISC-V, including GCC. Last Accessed: September 19, 2023. [Online]. Available: https://github.com/riscv-collab/riscv-gnu-toolchain

[7] Stephen Williams steveicarus/iverilog: Icarus Verilog. Last Accessed: September 19, 2023. [Online]. Available: https://github.com/steveicarus/iverilog

[8] AMBA 4 AXI4, AXI4-Lite, and AXI4-Stream Protocol Assertions User Guide: https://developer.arm.com/documentation/dui0534/b/?lang=en

[9] riscv-software-src/riscv-tools: RISC-V Tools (ISA Simulator and Tests). Last Accessed: September 19, 2023. [Online]. Available: https://github.com/riscv-software-src/riscv-tools

[10] Qflow 1.3: An Open-Source Digital Synthesis Flow : http://opencircuitdesign.com/qflow/

[11] Oklahoma State University System on Chip (SoC) Design Flows: https://vlsiarch.ecen.okstate.edu/flow/

[12] efabless/raven-picorv32: Silicon-validated SoC implementation of the PicoSoc/PicoRV32. Last Accessed: September 19, 2023. [Online]. Available: https://github.com/efabless/raven-picorv32

[13] Magic VLSI Layout tool: http://opencircuitdesign.com/magic/

[14] Netgen version 1.5 netlist comparison (LVS) and format manipulation: http://opencircuitdesign.com/netgen/index.html

[15] Odin II, logic synthesis and elaboration tool: https://docs.verilogtorouting.org/en/latest/odin/

[16] berkeley-abc/abc: ABC: System for Sequential Logic Synthesis and Formal Verification. Last Accessed: September 19, 2023. [Online]. Available: https://github.com/berkeley-abc/abc

[17] rubund/graywolf: graywolf is used for placement in VLSI design. Last Accessed: September 19, 2023. [Online]. Available: https://github.com/rubund/graywolf

[18] Qrouter version 1.3 (stable) and 1.4 (development) multi-level, over-the-cell maze router: http://opencircuitdesign.com/qrouter/