phoeniX RISC-V Core V0.1

Arvin Delavari – Faraz Ghoreishy

arvin7807@gmail.com – farazghoreishy@gmail.com

Summer 2023

Iran University of Science and Technology

Electronics Research Center – Digital Design Laboratory

This page was initially left blank.

Preface:

phoeniX is a 5 stage pipelined 32-bit RISC-V processor written in Verilog HDL. This project was started in summer of 2023 in Digital Design 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 phoeniX processor. This version of the processor is able to execute RV32I (integer operations extension) instructions of official RISC-V instruction set manual [1]. Other extensions will be supported by the core in the future updates.

phoeniX core is capable of execution of assembly and C codes using iverilog simulator. Guidance is included in this document for the flow of code execution in this processor. For execution of C codes, the processor 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 to relative sections of this document.

This project is an open source CPU under the GNU V3.0 license [2] and is free to use. You can find source codes and documentations in the following GitHub repository:

<https://github.com/ArvinDelavari/phoeniX-RV32>

We would like to express our sincere gratitude to the following individuals for their invaluable guidance and remarkable insights, which played a pivotal role in the completion of this project:

* Professor Sattar Mirzakuchaki PhD

Iran University of Science and Technology

* Professor Hadi Shahriar Shahhoseini PhD

Iran University of Science and Technology

* Nima Amirafshar MSc

Iran University of Science and Technology

Arvin Delavari – Faraz Ghoreishy

Iran University of Science and Technology – Summer 2023

Contents:

1. Introduction
   1. phoeniX core structure
   2. phoeniX software interface
   3. Required software
2. Building Blocks (Modules)
   1. Fetch Unit
   2. Instruction Decoder
   3. Immediate Generator
   4. Address Generator
   5. Arithmetic Logic Unit
   6. Register File
   7. Hazard Detection and Forwarding Unit
   8. Jump and Branch Unit
   9. Load Store Unit
3. Memory Interface Logic
4. Code Executant Software
   1. Code Execution Flow Requirements
   2. Windows (AssembleX Software)
   3. Linux (RISC-V GCC toolchain)
5. Synthesis Result
   1. Qflow toolchain
   2. TSMC 180nm PDK
   3. phoeniX ASIC result
6. References

Chapter 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 [3]. 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.

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 (V0.1), 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.

phoeniX 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. phoeniX core structure

In the beginning, one of the most specialties of this processor which should be mentioned is not having a single control unit. This processor uses a “Distributed Control Logic” which leads to elimination of control unit from the building blocks. By DCL it means that after decoding the instructions, relative fields such as opcodes, funct3, funct7 and other fields are straightly directed 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 of registers bus or immediate value), 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 target. In fact, modules are somehow known to be self-controllable which in this case we call the general system, Distributed Control Logic.

phoeniX is a 5 stage pipelined processor which has the classic machine cycle stages: Fetch, Decode, Execute, Memory Interface 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 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 skip stalls in the code including dependencies and hazards.

The maximum delay time (critical path) of phoeniX processor analyzed using Yosys [4] open-source synthesis tool and Vesta [5] static time analysis software, in different stages is somewhere around 3800-3900 picoseconds which leads us to 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 [6] microarchitecture will not reach this frequency rate. 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.

Even other open-source RISC-V embedded processors such as PicoRV32 by Claire Wolf [7] won’t reach a higher frequency than the 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 depending to the technology. Another well-known RISC-V processor is named ibex core by ETH Zurich [8] university which also has a frequency range between 100MHz and 300MHz in highest performance.

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. phoeniX software interface

The phoeniX processor uses standard GCC compiler toolchain [9] which is officially verified by RISC-V organization, in order to run C codes and assembly codes on the processor. This 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 the instruction memory. The processor can simulate the executed code using iverilog [10] version 12 tool. There is a shell script included in the repository which helps users to install all software requirements for simulating and coding on phoeniX processor.

The simulation and execution flow is very simplified by phoeniX code executants which will be described briefly in the upcoming sections of this document. There are easy solutions to simulate assembly and C codes 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 help 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 execute 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. Required Software

As mentioned before, there is a shell script integrated with the main directory of phoeniX RV32 core repository, which helps user to 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 in use of phoeniX core execution flow.

* Python3: Python [11] 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 has Python3 installed at the beginning by default. On Windows systems Python needs to be installed individually.
* Iverilog: it 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 we used iverilog version 12 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 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 smoothly 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

Building Blocks (Modules)

*“Great things are done by a series of small things*

*brought together.”*

*Vincent van Gogh*

In the following chapter, we will provide a comprehensive overview of the essential components that form the foundation of the phoeniX core. 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 data processing pipeline. 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, empowering you to comprehend its intricate design and appreciate the optimized performance it offers.

In the end there is a 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 a 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 this modules, there is a table included in this section which explains what is the role of each module of the core in a short and brief way. It is needed to not 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.

Below is the list of these modules, along with concise descriptions of their critical functions and roles in phoeniX core:

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

Table 1. phoeniX core modules and descriptions

* 1. Fetch Unit

The Fetch Unit module is responsible for fetching instructions from memory in a processor design. Let's go through the description of the module:

Input signals:

* enable: This input signal is used to enable the memory interface module.
* PC: Represents the Program Counter, which holds the address of the next instruction to be fetched.
* address: This input signal represents the branch or jump address generated by the Address Generator module.
* jump\_branch\_enable: This signal indicates whether a branch or jump operation is

enabled.

Output signals:

* next\_PC: This output signal is a 32-bit register that represents the Program Counter (PC) value of the next instruction to be fetched.

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 this module, it is set to READ, implying that the memory interface is performing a read operation.
* memory\_interface\_address: This output signal is a 32-bit register that represents the address being accessed by the memory interface. 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.

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 operation, 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 (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 control signals received.

Figure 1. Fetch Unit schematic output from Xilinx Vivado software

* 1. Instruction Decoder

The Instruction Decoder module is responsible for decoding instructions in a processor design. Let's go through the description of the module:

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 output signal is a 5-bit value representing the register index used for the first read operation.
* read\_index\_2: This output signal is a 5-bit value representing the register index used for the second read operation.
* write\_index: This output signal is a 5-bit value representing the register index used for the write operation.

Internal signals:

* instruction\_type\_i, instruction\_type\_b, instruction\_type\_r, instruction\_type\_i, instruction\_type\_u, instruction\_type\_j: These internal signals evaluate whether the instruction falls into a specific type category (I\_TYPE, B\_TYPE, R\_TYPE, etc.).
* 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 instruction\_type\_i, instruction\_type\_b, instruction\_type\_r, instruction\_type\_s, instruction\_type\_u, and instruction\_type\_j.

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 typically invalid.

Figure 2. Instruction Decoder schematic output form Xilinx Vivado software

In summary, the Instruction Decoder module decodes instructions and generates control signals based on the instruction type, opcode, and other fields of the instruction. It also determines the appropriate register read and write enable signals based on the instruction type.

* 1. Immediate Generator

The Immediate Generator module is responsible for generating the immediate value for instructions based on the instruction type. Let's go through the description of the module:

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 for the instruction.

Behavior:

Inside the always block, there is a case statement that evaluates the instruction\_type value.

Depending on the value of instruction\_type, the immediate value is assigned specific values based on the structure of the instruction.

For example, if the instruction\_type is I\_TYPE, the immediate value is assigned by concatenating 21 copies of the most significant bit of the instruction with bits 30 to 20 of the instruction. You can find the immediate generation algorithm for each type in RISC-V original instruction set architecture documents.

Similarly, for other instruction types (S\_TYPE, B\_TYPE, U\_TYPE, J\_TYPE), the immediate value is generated by concatenating specific bits of the instruction with appropriate padding bits.

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

* 1. Address Generator

The Address Generator module is responsible for generating addresses based on the opcode and other inputs. Let's go through the description of the module:

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.

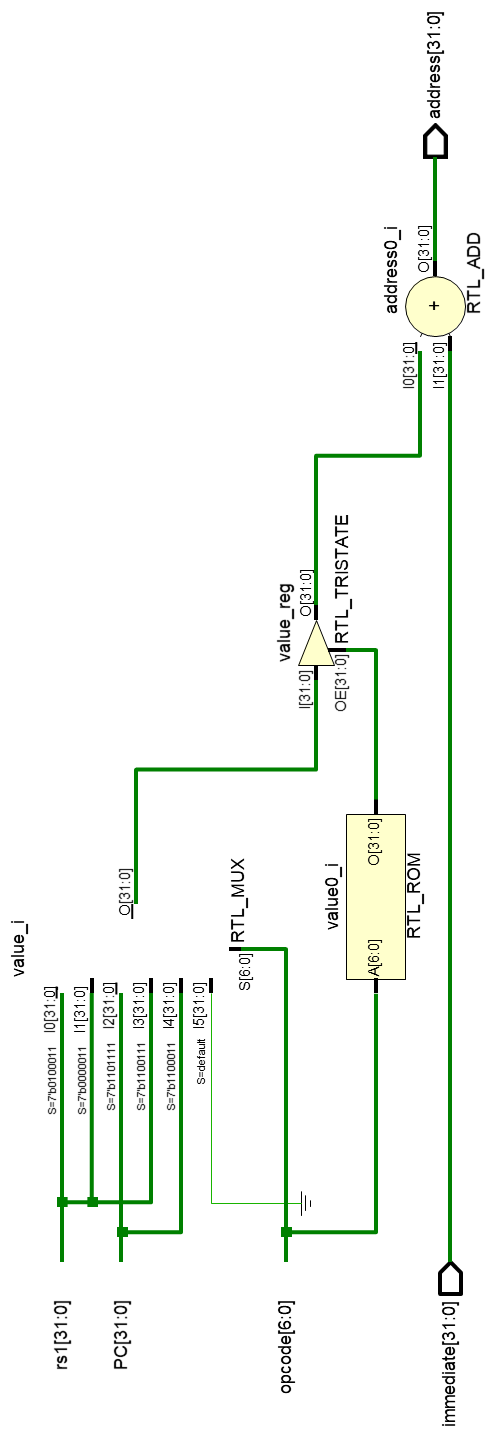
Internal Signals:

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

Behavior:

Inside the always block, there is a case statement that evaluates the opcode value.

Depending on the value of opcode, the value signal is assigned 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.

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.

Figure 4. Address Generator schematic output from Xilinx Vivado software

Arithmetic Logic Unit

The ALU is responsible for executing R-Type, I-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 Source). The output of the ALU is the alu\_output, which holds the result of the ALU operation.

The module contains the following components:

* Multiplexers: There are two multiplexers which mux1\_select and mux2\_select signals 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 (JAL and JALR). The result of the ALU operation is assigned to the alu\_output register.

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

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

* 1. 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 (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.

* 1. Hazard and Forwarding Unit

The Hazard Detection and Forwarding Unit module represents a hazard forwarding unit in a digital system. It 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.
* enable\_1, enable\_2, enable\_3: Control signals indicating whether the preceding instructions are writing to the registers.

Output signals:

* forward\_enable: A control signal indicating whether data forwarding is enabled.
* forward\_data: The forwarded data value.

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).

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

If a match is found, 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 asserted, 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.

* 1. Jump and Branch Unit

The Jump Branch Unit module is responsible for determining whether a jump or branch instruction should be executed based on the opcode, funct3, and instruction\_type. It generates a control signal, jump\_branch\_enable, which indicates whether a jump or branch should occur. Here is a description of the module:

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 register that indicates whether a branch operation should be enabled.
* jump\_enable: A register that indicates whether a jump operation should be enabled.

Behavior:

The module uses an always @(\*) block to perform combinational logic based on the input signals.

If the instruction type 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 the logical OR of jump\_enable and branch\_enable, indicating whether a jump or branch operation should be enabled.

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

* 1. Load Store Unit

The Load Store Unit represents a load/store unit in a computer system, which is responsible for handling memory accesses for load and store instructions.

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\_enable: This output signal indicates whether the memory interface should be enabled or disabled.
* memory\_interface\_state: This output signal represents the state of the memory interface.
* memory\_interface\_address: This output signal specifies the address for memory access.
* memory\_interface\_frame\_mask: This output signal indicates the frame mask for memory access.
* 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 memory access.

For load instructions, the module latches the data returned from memory based on the funct3 field and the frame mask. For store instructions, the module latches the store data based on the funct3 field and the frame mask.

Overall, the module provides the necessary functionality to perform memory accesses for load and store instructions in a computer system.

Additional explanations are included in Memory Interface Logic (chapter 3) of this document which will explain the logic behind memory interface signals both in Load Store Unit and Fetch Unit.

Chapter 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 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 clock signal of the core. 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 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 is 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 design module.

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 point of view.

Let's break down the memory interface logic in a complete way:

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. (frame mask is used for defining the target bytes, half words or words which should be loaded or stored in 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 negedge of the CLK signal.

The core always addresses memory by a word aligned address and access a 4-byte frame from memory which is then operated on based on a `frame\_mask` for half-word and byte operations.

It is important to note that phoeniX Load Store Unit does not support misaligned accesses. If the instruction memory interface is not enabled (instruction\_memory\_interface\_enable is false), the instruction\_memory\_interface\_data 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. (frame mask is used for defining the target bytes, half words or words which should be loaded or stored in 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. On the other hand, the instruction memory always loads a 32-bit data, encompassing the firmware instructions stored within.

Data Memory Interface Behavior:

The behavior of the data memory interface is described using an always block triggered by the negedge of the CLK signal.

If the data memory interface is not enabled (data\_memory\_interface\_enable is false), the data\_memory\_interface\_data\_reg 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 posedge of the CLK signal to avoid glitches.

Overall, this testbench module 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 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

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 delving 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 prepared 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.

* 1. Code Execution Flow

The mentioned shell is in the Setup directory in the phoeniX core repository. 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 [12]. 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. 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/ArvinDelavari/phoeniX-RV32.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 user name 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.

* 1. Windows (AssembleX Software)
  2. 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 delving into the intricacies of the Makefile, it is essential to understand the requisite 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 .globl, .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 connected 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 readable hex file representing the instruction memory for the phoeniX core which is designed in Verilog.

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.

Chapter 6

References

[1] RISC-V ISA

[2] GNU LICENSE

[3] IUST ERC WEBSITE

[4] Yosys

[5] Vesta

[6] ARM Cortex M

[7] PicoRV32

[8] ibex

[9] GCC compiler toolchain

[10] iverilog

[11] Python

[12] RISC-V tools