phoeniX RISC-V processor platform is designed in Verilog HDL based on the 32-bit Base Instrcution Set of RISC-V Instruction Set Architecture and can execute `RV32IEM` instructions (user have options to choose between `I` and `E`, and also to activate/deactivate `M` extensions), with special features supported for **approximate computing** techniques. In fact, **phoeniX** is a novel modular and extensive RISC-V platform for approximate computing.

The constant demand for energy-efficient and high-performance embedded systems motivates the development of new processor architectures, leading to modern concepts in computer engineering and digital systems, which **approximate computing** is a well-known example of. This project includes a novel modular and extensive approximate computing embedded processor platform named **phoeniX**, using the standard RISC-V ISA extensions, which aims to maximize energy efficiency while maintaining acceptable application-level accuracy.

The proposed platform enables integration of approximate arithmetic units at the core level, with different structures, accuracies, timings and etc. without any need for editing rest of the core, especially in control logic. This platform is allowing configurable trade-offs between speed, accuracy and energy consumption based on specific application requirements. Additionally, the platform includes a modular architecture that enables easy integration of various specialized units, such as hardware accelerators and coprocessors, to enhance performance for specific tasks.

To evaluate the effectiveness of the platform, extensive experiments were conducted on a set of benchmark applications, showcasing significant energy and speed improvements compared to conventional RISC-V platforms. Overall, the results demonstrate the potentials of the proposed modular and extensive approximate computing processor platform for energy-efficient and high-performance embedded systems.

You can find a full list of RISC-V assembly instructions in the ISA Specifications Documents.

The core can be implemented as a softcore CPU on Xilinx 7 Ultrascale/Ultrascale+ series FPGA boards using logic synthesis. This allows 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.

The core has undergone a complete synthesis flow to become an Integrated Circuit using **Cadence Genus** tool. The implementation was specifically carried out utilizing the `NanGate 45nm` Process Design Kit (PDK).

This repository contains an open source CPU including RTL codes and assistant software, under the GNU V3.0 license and is free to use. The platform's technical specifications are published under supervision of IUST Electronics Research Center.

You can cite the document as:

- A. Delavari, F. Ghoreishy, H. S. Shahhoseini and S. Mirzakuchaki. (2023), “phoeniX: A RISC-V Platform for Approximate Computing V0.1 Technical Specifications,” [Online]. Available: http://www.iust.ac.ir/content/76158/phoeniX-POINTS--A-RISC-V-Platform-for-Approximate-Computing
- Designed By : Arvin Delavari and Faraz Ghoreishy

- Contact us : arvin7807@gmail.com - farazghoreishy@gmail.com
- Iran University of Science and Technology, Summer 2023 - Present

Table of Contents

Features

- **Optimized 3 stage pipline**

The 3-stage pipeline in a processor improves instruction throughput by dividing execution into sequential stages with minimal internal fragmentation. By incorporating data forwarding and bypassing options (such as forwarding data from execution, memory or writeback stage) the pipeline minimizes stalls caused by data hazards. As a result, the pipeline achieves higher performance, reduced stalls, and improved instruction-level parallelism, enabling concurrent processing of independent instructions.

- **Modularity and Extensiveness**

Modularity in processor design promotes flexibility, reusability, scalability, simpler testing, and increased system reliability by breaking down the processor into smaller, independent modules that form the building blocks. Each one of these building blocks can be designed, optimized, and tested separately. This approach offers several benefits.

First, modularity increases flexibility and reusability, as individual modules can be easily interchanged or upgraded without requiring significant changes to the main core. This enables efficient customization and adaptation to different application requirements (e.g. adding a multiplier/divider module to the design would cause significant changes to a centralized control unit, but in this methodolgy, designing self-controlled execution units would lead to a much simple integration of the module to the main core).

Secondly, modularity aids in design verification and testing, as individual modules can be tested in isolated testbenches, simplifying the debugging process and reducing the overall development time.

Additionally, modular designs can lead to improved overall system reliability, as faults and failures in one module are less likely to affect the functionality of the entire processor.

- **A Novel Platform for Approximate Computing**

The phoeniX RISC-V core introduces novel features that will help the emerging field of approximate computing techniques. With its modular design and extensive architecture, phoeniX presents a configurable platform for exploring and implementing approximate computing methodologies for developers and designers.

This platform enables researchers and developers to delve into the field realm of approximate computing, where trade-offs between accuracy and computational efficiency can be carefully balanced. By offering a range of specialized instructions, optimized datapaths, and adaptable precision controls, phoeniX empowers users to use the help of approximation in diverse application domains, openning the way for advancements in energy-efficient computing, machine learning, image processing, and etc.

Directory Map

The tree below provides a map to all directories and sub-directories of the repository. Detailed descriptions on contents of these directories are provided in the following sections and each specific `README.md`.

```
repository/
    ├── Setup/
    │   └── setup.sh
    ├── Documents/
    │   ├── Images/
    │   ├── phoeniX_Documentation_V0.2/
    │   └── RISCV_Original_Documents/
    ├── Features/
    │   ├── AXI4-Lite/
    │   ├── Branch_Prediction/
    │   ├── Clock_Genrator/
    │   └── ...
    ├── Synthesis/
    │   ├── TSMC_018um/
    │   │   ├── layout/
    │   │   ├── synthesis/
    │   │   ├── log/
    │   │   └── ...
    │   └── NanGate_45nm
    ├── Modules/
    │   ├── Address_Generator.v
    │   ├── Arithmetic_Logic_Unit.v
    │   └── ...
    ├── Firmware/
    │   ├── hex_converter -> hex_converter.py
    │   ├── start_procedure -> start.s
    │   ├── start_linker -> start.ld
    │   ├── riscv_linker -> riscv.ld
    │   └── syscalls -> syscalls.c
    ├── Software/
    │   ├── Sample_Assembly_Codes/
    │   │   └── Program_directory/
    │   │       ├── Program.S
    │   │       ├── Program.txt
    │   │       └── Program_firmware.hex
    │   ├── Sample_C_Codes/
    │   │   └── Program_directory/
    │   │       ├── Program.c
    │   │       ├── Program.o
    │   │       └── Program_firmware.hex
    │   │
```

```
│   └── User_Codes/
│       └── Program_directory/
│           ├── Program.c
│           ├── Program.o
│           └── Program_firmware.hex
├── phoeniX.v
├── phoeniX_Testbench.v
├── phoeniX.vvp
├── phoeniX.vcd
├── phoeniX.gtkw
├── AssembleX_V1.0.py
└── Makefile
```

phoeniX Core Structure

The repository contains a collection of Verilog modules that build up the phoeniX RISC-V processor. These building block modules are included in `\Modules` directory. Each modules was designed with concepts of modularity and distributed-control in mind. This deliberate approach allows for effortless replacement and configuration of individual building blocks, resulting in a simplified process. This will help designers with integrating new modules (especially arithmetic and execution units) within the processor.

The proposed platform enables integration of approximate arithmetic units at the core level, with different structures, accuracies, timings and etc. without any need for editing rest of the core, especially in control logic. This platform is allowing configurable trade-offs between speed, accuracy and energy consumption based on specific application requirements.

This repository includes detailed documentation, user manual, and developer guidelines for future works and updates. These resources make it extremely easy for users to execute C and Assembly code using the standard RISC-V GCC toolchain on the processor, and helps developers to understand its structure and architecture, in order to update and validate new designs using the base processor, or adding and testing approximate arithmetic circuits on the core, without any need of changes in other parts of the processor such as control logics and etc. With this knowledge, developers can enhance the processor, add new features, and develop different architectural techniques effectively.

Module	Description
`Address_Generator`	Generating address for BRANCH, JUMP and LOAD/STORE instructions
`Arithmetic_Logic_Unit`	ALU with support for `I_TYPE` and `R_TYPE` RISC-V instructions
`Control_Status_Unit`	CSR instructions and custom CSRs for approximate computing acceleration of the phoeniX
`Divider_unit`	Divider unit with a modular design (Default module: Error configrable non-restoring divider)
`Fetch_Unit`	Instruction Fetch logic and program counter addressing
`Hazard_Forward_Unit`	Hazard detection and data forwarding logic in pipelined processor

Module	Description
`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 wordsize management
`Multiplier_Unit`	Multiplier unit with a modular design (Default module: Fast, low-power approximate multiplier)
`Register_File`	Parametrized register file suitable for GP registers and CSRs (2 read & 1 write ports)

The `phoeniX.v` contains the main phoeniX RISC-V core and is included in the top directory of this repo:

Module	Description
`phoeniX`	phoeniX 32 bit RISC-V core (RV32IEM) top Verilog module
`phoeniX_Testbench`	phoeniX testbench module including main core, memory and interface logic

phoeniX Memory Interface

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 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.

> **WARNING**
> Unaligned Memory Accesses: phoeniX Load Store Unit does not support misaligned accesses. At the moment we are working to add support accesses that are not aligned on word boundaries by implementing the procedure with multiple separate aligned accesses requiring additional clock cycles.

Building RISC-V Toolchain

In order to be able to compile your source files and run or simulate with RISC-V, you need to install `RISC-V GNU Compiler Toolchain`. You can follow the installation process from the riscv-gnu-toolchain repository or use the scripts provided in the original RISC-V repositories and riscv-tools. 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 prequistes, iVerilog version 12 and gtkwave.

```
user@Ubuntu:~$ git clone https://github.com/phoeniX-Digital-Design/phoeniX.git
user@Ubuntu:~$ cd phoeniX
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 change 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 for a `x86_64` machine. 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
```

phoeniX Execution Flow

Linux

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.

phoeniX 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.

```
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` 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.

Running Your Own Code

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 run the following `make` command from the main directory:

```
make code project=my_project
```

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`.

Windows

Running Sample Codes

We have meticulously developed a lightweight and user-friendly software solution with the help of Python. Our execution assistant 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 enhances the efficiency of the code execution process, offering a streamlined experience for users seeking to enter the realm of assembly programming on pheoniX processor in a very simple and user-friendly way.

Before running the script, note that the assembly output of the Venus Simulator for the code must be also saved in the project directory. To run any of these sample projects simply run python `AssembleX_V1.0.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_V1.0.py sample {project_name}
```

For example:

```
python AssembleX_V1.0.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 Code

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 ``user_code.s` files in my_project and run the following command from the main directory:

```
python AssembleX_V1.0.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.

Synthesis Result

The code has been crafted to enable the utilization of the processor as a synthesizable and implementable soft-core on Xilinx Ultrascale and Ultrascale+ FPGA devices. The RTL synthesis of the phoeniX processor was done using Cadence Genus tool, using the `NanGate 45nm` technology, also known as the `FreePDK45, Open Cell Library` process technology. The Static Time Analysis (STA) results indicate that the maximum delay observed in the core modules, and consequently in the pipeline stages, is about less than 1900 picoseconds using the **45nm** technology. Setting the clock cycle time at **2 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 **500MHz**, enabling efficient execution of instructions and supporting the desired operational specifications in embedded processors.

This table provides a comparison of similar embedded processors to phoeniX, used in the industry, in terms of frequency, architecture, and manufacturing technology. This analysis helps to assess their performance and technical aspects, aiding decision-making for selecting the most suitable processor for various industrial applications. It is important to note that phoeniX is an embedded processor platform which is extensive, and

execution units are replaceable; This means that these reported results of phoeniX core is extracted from the platform using its default (demo) execution engine.

Processor	Max Frequency (MHz)	Technology Node (nm)	Brand	Architecture
phoeniX	500	45	IUST ERC	RV32IEM
Cortex-M0	48	90	ARM	ARM Cortex-M0
Cortex-M0+	48	90	ARM	ARM Cortex-M0+
Cortex-M1	128	180	ARM	ARM Cortex-M1
Cortex-M3	120	90	ARM	ARM Cortex-M3
Cortex-M4	180	90	ARM	ARM Cortex-M4
Cortex-M7	400	40	ARM	ARM Cortex-M7
Cortex-M23	48	55	ARM	ARMv8-M
Cortex-M33	100	40	ARM	ARMv8-M
Cortex-A5	500	40	ARM	ARMv7-A
Cortex-A7	1000	28	ARM	ARMv7-A
Cortex-A9	1500	28	ARM	ARMv7-A
FE310	150	180	Si-Five	RV32IMAC
ESP32	240	40	Espressif	Xtensa LX6
PIC32MX795F512L	80	90	Microchip	MIPS32 M4K