

0.0.0.1. Workflow:

1. Post todo notes on structural content that needs to be there.
2. Dissection: create subsections / paragraphs and split todo notes into smaller ones.
3. Write paragraphs replacing todo notes. put them in a `\draft{}` command.
4. Get feedback by a different person (work on the granularity of draft sections):
 - a) Fix spelling and errors directly
 - b) Add todo/clarify notes if something requires discussion to fix
 - c) Start discussions in comments if you are unsure about something
 - d) Integrate the feedback
5. After consensus was reached on a `\draft` section, change it to `\final`

If you write information that is only valid for the intermediate report and must be removed or revisited for the final document version, encapsulate it in a `\nonfinal{}` command.

If you refer to a file in our code base, enter the full path relative to the environment base and encapsulate it in an `\sfile{}` command, example: `\sfile{riscv-tools/bin/riscv-uart-flash}`

If you refer to a git repository, put it in a `\repo{}` command, example:
`\repo{riscv-llvm}`

When discussing a problem that occurred or may occur (user guide), state the exact problem in a `\problem{}` command and append the solution immediately after that in either a `\solution{}` or `\workaround{}` command.

Paderborn CPU Core for Approximate Computing (PACO)

Implementation Document (intermediate)

September 25, 2016

Contents

0.0.0.1. Workflow:	1
I. User Guide	6
0.1. Introduction	6
0.2. How to read the User Guide	7
0.2.1. Component Overview	8
0.2.2. User Guide Overview	8
0.3. Step-by-step guide	8
0.3.1. Setting up the environment	8
0.3.1.1. Using the Setup Script	11
0.3.1.2. Manual Setup	11
0.3.2. Generating the CPU core	12
0.3.2.1. Note:	12
0.3.3. Synthesizing the system for FPGA instantiation	12
0.3.4. Compiling programs	13
0.3.4.1. Examples	13
0.3.4.2. Manual steps	13
0.4. Run PACO Approximate Applications	17
0.4.1. Simulate using the C Emulator	17
0.4.2. Simulate using QEMU	17
0.4.3. Running Programs on an FPGA Instantiation	18

II. Developer's Guide	21
1. Overview	21
1.1. Current state of implementation	23
1.1.0.1. Environment	23
1.1.0.2. Compiler	24
1.1.0.3. Hardware: ALU	24
1.1.0.4. Hardware: LUT	24
1.1.0.5. Problems	25
1.1.0.6. Timeframe	26
2. Environment	27
2.1. Generating the environment	27
2.1.1. Original code base	27
2.1.2. Modified code	27
2.1.2.1. RISC-V toolchains	30
2.1.2.2. Rocket core	30
2.1.2.3. Rocket SoC	30
2.1.2.4. Rocket SoC bootloader	30
2.1.2.5. RISC-V python library	31
2.1.2.6. RISC-V tests	31
2.1.2.7. QEMU	31
2.1.2.8. Virtual Machine	31
2.2. Tools	32
2.2.1. RISC-V tests	33
2.2.1.1. Testing approximate instructions/hardware	33
2.2.2. Rocket Chip C Emulator	34
2.2.2.1. Changes	35
2.2.3. QEMU	35
2.2.3.1. Usage	35
2.2.3.2. Modifications done	36
2.2.3.3. Further modification	36
2.2.3.4. Adding new decoding masks	37
2.2.3.5. Adding new major opcodes	37
2.2.3.6. Adding decoder / instruction translation	37
2.2.4. UART debug interface (flashing tool)	37
2.2.4.1. UART interface	38
2.2.4.2. UART Out-of-band reset	38
2.2.4.3. Flashing tool	39
2.2.4.4. Modifications	39
2.2.5. Rocket SoC Runtime Library	40
2.2.5.1. Program termination	40
2.2.5.2. Library Components	40
2.2.5.3. Template Applications	40

2.3. Modifications	41
2.3.0.1. Makefile adjustments	41
3. Compiler System	41
3.1. Original Code Base	42
3.2. Modified Code	42
3.2.1. High-level Code Compilation	42
3.2.1.1. General Modifications	43
3.2.1.2. Parsing and Analysis of the Approx Decorator	43
3.2.1.3. Parsing and Analysis of Pragmas	45
3.2.1.4. Parsing of Approximate Arithmetic and Approximated Functions	46
3.2.1.5. Analysis of Approximate Computations	47
3.2.1.6. Code Generation	48
3.2.1.7. Test Code	48
3.2.2. LLVM Translation	48
3.2.2.1. Adding Intrinsics, Translating to Instructions	48
3.2.2.2. Selection DAG, Passes and UUID-Translation	50
3.2.3. Machine Code Generation	51
3.2.3.1. Modifying the Assembler	51
3.2.4. Lookup Table Compilation	52
3.2.4.1. Usage	52
3.2.4.2. File Formats	52
3.2.4.3. Command-line Options	53
3.2.4.4. Segmentation Strategies (Primary)	53
3.2.4.5. Segmentation Strategies (Secondary)	54
3.2.4.6. Approximation Strategies	55
3.2.4.7. Implementation Overview	55
3.2.5. Error Handling and Testing	58
3.2.5.1. Exception classes	58
3.2.5.2. Logging System	59
3.2.5.3. Unit Tests	59
3.2.5.4. System Tests	59
3.3. Limitations and improvement-worthy parts	59
3.3.0.1. Floating-point Support	59
3.3.0.2. Target Function Evaluation	59
3.3.0.3. Default Strategies	59
4. Approximation Hardware	59
4.1. Environment in Rocket chip	59
4.1.1. Chisel	60
4.1.2. Decoder	60
4.2. LUT	61
4.2.1. Overview	61

4.2.2.	Chisel Interface	62
4.2.3.	Hardware Core	64
4.2.3.1.	Component Overview	64
4.2.3.2.	Architecture Parameters	65
4.2.3.3.	LUT controller	65
4.2.3.4.	Input decoder	66
4.2.3.5.	Address translator	66
4.2.3.6.	Lookup Table	66
4.2.3.7.	Interpolator	66
4.2.3.8.	Testing	66
4.2.3.9.	Configuration Bitstream	67
4.2.4.	Design Space	68
4.2.4.1.	Parameters, metrics	69
4.2.4.2.	Exploration	69
4.2.4.3.	Conclusion	69
4.3.	Approximate ALU	69
4.3.1.	Overview	69
4.3.2.	Original code base	70
4.3.3.	Modified Code Base	70
4.4.	Power estimation	73
4.4.1.	Xilinx Power Analyzer	73
4.4.2.	Synopsis Tool	74
Appendices		80
A. External resources		80
A.1.	QEMU	80
A.2.	Clang/LLVM	80
A.3.	GNU/Binutils	80
A.4.	Chisel	80
A.5.	Scala	81
A.6.	RISC-V	81
A.7.	Rocket Chip	81
A.8.	Rocket SoC	81

Part I.

User Guide

0.1. Introduction

This is the User Guide for the Paderborn Approximate Computing Core (PACO), intended to enable a reasonably experienced computer user to commission a PACO approximate computing system and compile/run software on it. If you want to create your own (hardware) approximation techniques, integrate them into the CPU core and modify the compiler toolchain to allow creation of software for it, you will additionally need the *PACO Developer's Guide*.

What is the PACO?

The PACO project was conceived by Christian Plessl and Paul Kaufmann at the University of Paderborn and designed and implemented by a team of Master's students under their guidance.

Goal was to provide an experimentation platform for approximate computing, allowing researchers to explore new approximation techniques in the full stack of a modern computing system.

We picked the open Rocket Chip implementation of the RISC-V instruction set as a basis for our implementation, because it ships with a fairly complete toolchain for modern

- hardware description (<https://chisel.eecs.berkeley.edu/>),
- synthesis for both FPGA and ASIC,
- several levels of emulation and simulation for testing and debugging.

Into this core we have integrated two approximation techniques, an ALU capable of approximate calculations and a Lookup table functional unit capable of approximating part of the domain of a function by sections. To be able to test these approximation techniques, we have also modified the control plane of the underlying chip and extended the RISC V ISA with approximate instructions and created compiler support for them. There are many more approximation techniques either proposed or not thought-of yet. We want to enable other groups interested in Approximate Computing to test and benchmark their techniques in this full-stack environment. So, we have documented each step needed to implement approximation techniques for their benefit, primarily in the Developer's Guide. All our own modifications and new developments have been designed with easy modification in mind.

Precursor Software

We are standing on the shoulders of the Rocket Chip development team, the RISC-V development consortium, the LLVM project and the QEMU project.

0.2. How to read the User Guide

This subsection contains general information allowing you to understand the rest of this User Guide much more quickly: e.g. environment root directory, shell-commands, paths to files, and notes on make.

This is followed by an overview of the major components of a PACO approximate computing system, allowing you to understand the language of the remaining User Guide sections.

Last is an overview of the User Guide.

If you are already familiar with the PACO system and only need the *step-by-step installation guide*, skip to Section 0.3.

Environment root directory is the base repository you check out. All other repositories are located as subdirectories in there. This document will always refer to it as *paco-env*.

Shell-Commands are always framed in a box and should be executed from the *paco-env* directory if not otherwise stated. Every command starts with a "\$" symbol. A command looks like this

```
$ echo "execute this"
```

and should be copied to the terminal.

Paths to files are shortened in this document and they can be looked up in Appendix 4.4.2, in the PDF version with a click. Whenever a file or directory is mentioned it looks like this `Top.GnssConfigNoFPU.v`, and the full path can be found in Appendix 4.4.2. **TODO (MUST):** get Peter to create special files as an appendix

Notes to make include the use of the command line option "-j" which specifies how many jobs can be run in parallel. In this document we will always add "-jN" where *N* is the number of jobs you want to run in parallel. This "N" has to be replaced by you. A good rule of thumb is to set N to the number of cores your cpu has.

\$RISCV variable is used as a base directory for your installed tools, e.g. (RISCV-)gcc, (RISCV-)clang, etc. This variable is automatically set if you have sourced env.sh. The default value for this variable is the directory riscv-tools.

Program Code is framed in a box, similarly to shell-commands. Unlike shell-commands the lines do not start with a "\$" symbol. An example in the C programming language:

```
#include<stdio.h>
int main()
{
    printf("Hello_PACO");
}
```

0.2.1. Component Overview

Table 1 (hardware) and Table 2 (software) can give you an overall view of the PACO components essential to run your first approximate PACO applications on an FPGA. The hardware components can be seen in Table 1. This table describes the components that generate and communicate with the hardware, on an FPGA. The software components are listed in Table 2. This table focuses on the tools a developer needs to compile, debug, and execute PACO C/C++ programs.

0.2.2. User Guide Overview

The next subsection 0.3 contains detailed instructions on how to set up your PACO approximate computing system from scratch, including generation of configuration bitstream for an FPGA and compilation of PACO approximate applications. The last subsection (0.4) of the User Guide shows how approximate applications can be emulated with either QEMU or our cycle-accurate emulator or run on the FPGA.

0.3. Step-by-step guide

The step-by-step guide allows you to set up a PACO approximate computing system. This will create an environment such that you can instantiate the CPU-core with all components on the FPGA. You can also compile programs to be run on the FPGA, and simulate the core with a C emulator generated from the same Chisel hardware specification used to generate the FPGA configuration bitstream. This guide assumes that you have a Linux system (tested on Ubuntu 16.04 LTS). If you are using Ubuntu 16.04 LTS please make sure you have the following packages installed by running:

```
$ sudo apt-get install autoconf automake autotools-dev curl libmpc-dev \
    libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf \
    libtool patchutils bc uuid-dev liblua5.2-dev
```

0.3.1. Setting up the environment

There are two alternative methods of environment setup:

- invoke the provided setup script

Category	Component	Description	Further documentation
CPU-Core	Rocket-Chip	The Rocket-Chip consists of CPU + Caches + Buses that PACO uses and is written in the Chisel hardware definition language. It is the host to our approximation hardware units, i.e. the Lookup table(LUT) and the Approximate ALU/MUL.	<ul style="list-style-type: none"> • Chisel: Section 4.1.1 • Extern Chisel: Sec. A.4 • Extern Rocket: Sec. A.7
	LUT	The Lookup table (LUT) is a functional unit in the CPU that can e.g. be configured to evaluate arithmetic functions in functional approximation. Functions are linearly approximated within segments of the function domain. Configuration data includes incline and offset per segment, as well as configuration for the segment identification logic. The LUT is currently configured at load time of an application, although nothing should prevent you from configuring during runtime.	<ul style="list-style-type: none"> • Section 4.2
	Approx. ALU	The Approximate ALU/MUL is used to explore the effect of approximation on the output of a program. The unit cuts a programmable amount of least significant bits from approximate ALU operands, resulting in functional approximation.	<ul style="list-style-type: none"> • Section 4.3
non-CPU	Rocket-SoC	The Rocket-SoC is the non-CPU hardware in which the Rocket-Chip is embedded. It contains useful devices, like a UART, boot ROM, and SRAM program memory. All the components are connected via AXI-bus. The UART in particular is used to communicate between PACO on the FPGA and the Linux computer controlling it using the flash-tool.	<ul style="list-style-type: none"> • Extern: Appendix A.8
	flash-tool	The flash-tool allows bidirectional communication with and control of the Rocket-SoC running on the FPGA. It can reset it, write a program into it's SRAM over the UART, start the program, and communicate bidirectionally.	<ul style="list-style-type: none"> • Section 2.2.4

Table 1: PACO hardware component overview: Short descriptions of major hardware components in the PACO approximate computing core, as well as the FPGA interface needed to run applications it. The last column contains references to more detailed descriptions of the components.

Category	Component	Description	
Compiler	Clang/LLVM	Clang/LLVM compiles approximate C/C++ programs into assembly. This is the compiler you want to use for approximate components of applications. For the LUT, it has to be used in combination with the lut-compiler and the lut-startup tools. Clang/LLVM cannot produce assembled files, but this can be done with GCC or Binutils.	<ul style="list-style-type: none"> • Clang: Section 3.2. • LLVM: Section 3.2 • Extern: Appendix
	GCC	GCC is the C/C++ compiler which can compile C, C++, Assembly files into native PACO executables. It should be used as a companion for Clang/LLVM or for parts of your program not using any approximation component.	<ul style="list-style-type: none"> • Extern: Appendix
	Binutils	Binutils has useful tools for assembling, linking and analyzing binary programs. It is the tool GCC eventually calls to assemble/ link. “objdump” is useful to analyze a compiled program or debug a LUT configuration.	<ul style="list-style-type: none"> • Extern: Appendix
Libraries	RocketLib	This library contains useful functions for communication between a program running on the FPGA and your connected Linux PC. One end of the communication is your program using this library, the other end is the flash-tool.	<ul style="list-style-type: none"> • Section 2.2.5
LUT-config.	lut-startup	The lut-startup tool generates assembly code used to load configuration data into the LUT hardware at program startup time.	TODO (MUST): Ad
	lut-compiler	The lut-compiler receives an arithmetic function in C/C++ code as well as approximation parameters and generates configuration data for the LUT.	<ul style="list-style-type: none"> • Section 3.2.4
Emulator	Chisel C-Emulator	The C emulator is generated directly from the Chisel HDL of the CPU and simulates it cycle accurate. This emulator can be used to test the approximate ALU.	TODO (MUST): Ad
	QEMU	QEMU is a fast but functional emulator which is not cycle accurate. It can be used to simulate programs for the approximate ALU and was used as a quickstart for our development.	<ul style="list-style-type: none"> • Section 0.4.2 • Section 2.1.2.7

Table 2: PACO software component overview: Short descriptions of major components needed to compile/generate software for the PACO approximate computing core. The last column contains references to more detailed descriptions of those components.

- setup manually

This guide assumes that you have all the folders at the right place as well as the correct packages, e.g. gcc, liblua, etc. installed. Also make sure you have the file env.sh sourced in your terminal. After installation all tools should be installed in the path riscv-tools and should also reside in your system-path so that you can invoke them on your terminal without giving a path to the program.

TODO (MUST): missing: command line downloading and installing the paco-env repository

0.3.1.1. Using the Setup Script The script lies in the paco-env repository and can be invoked by:

```
$ cd paco-env
$ ./install.sh
```

This builds and installs all tools except the CPU-core into the path riscv-tools.

0.3.1.2. Manual Setup The manual setup consists of five steps:

1. Build the RISC-V Toolchain
2. Build LLVM and Clang
3. Build the LUT Compiler
4. Install the Python Libs
5. Install the Rocket Lib for C programs

1. Build the RISC-V Toolchain To install the RISC-V toolchain run:

```
$ cd paco-env/riscv-tools-src
$ ./build.sh
```

This will install the gcc compiler, binutils, and the riscv-tests into the directory riscv-tools.

2. Build LLVM and Clang Make sure you have the packet libuuid installed and then run:

```
$ cd paco-env/riscv-tools-src/riscv-llvm
$ CC=gcc CXX=g++ ../configure --enable-targets=riscv --prefix=$RISCV
$ make -jN && make install
```

Attention: remember to replace N with the number of threads you want to spawn.

This will install the clang compiler and the llvm tools into the directory riscv-tools.

3. Build the LUT Compiler To install the LUT compiler make sure you have a lua package installed and run:

```
$ cd paco-env/riscv-tools-src/riscv-lut-compiler
$ make -jN && make install
```

This will install the LUT compiler into the directory riscv-tools.

4. Install the Python Libs To install the python libs run:

```
$ cd paco-env/riscv-tools-src/py
$ make install
```

This will install the python libs into the directory riscv-tools.

5. Install the RocketLib for C Programs To install the RocketLib run:

```
$ cd paco-env/rocket-soc/rocket_soc/lib
$ make -jN && make install
```

This will install the RocketLib into the directory riscv-tools.

From this point on your system should be prepared to run all the software tools of this environment.

0.3.2. Generating the CPU core

The Rocket-SoC (System-on-Chip) encapsulates a variety of hardware modules, among them is the CPU (Rocket-Chip). The Rocket-Chip is written in the Chisel HDL and needs to be converted into Verilog to be used by Rocket-SoC. This can be done by running:

TODO (MUST): missing: command line downloading and installing the paco-env repository

```
$ cd paco-env/rocket-chip/fsim
$ make -jN && make install
```

0.3.2.1. Note: Since Chisel is embedded into the Scala language, you need to have the Java Virtual Machine installed for this to work.

Using 'make install', the Verilog description is directly copied into the correct directory Top.GnssConfigNoFPU.v and the Rocket-Soc is now ready for synthesis.

0.3.3. Synthesizing the system for FPGA instantiation

For this step you need to have Xilinx ISE 14.7 installed on your system as well as the Xilinx cable driver. A guide on installing both of them can be found here ¹. This step assumes that ISE is installed in the path `/opt/Xilinx/14.7/ISE_DS`. You start ISE by first sourcing the file `settings64.sh` and invoking:

¹http://www.george-smart.co.uk/wiki/Xilinx_JTAG_Linux

```
$ source /opt/Xilinx/14.7/ISE_DS/settings64.sh
$ ise&
```

This will start ISE in a GUI window. From here you have to select "File - Open Project" and open the Rocket-SoC project file called `rocket_soc.xise`. If a window pops up stating the file "Memo.vhd" cannot be found, you can click the checkbox "Remove unspecified files from project" and proceed. Once you open the project click *generate programming file*, which starts the synthesis process. Once the process is finished, a bitstream file is created called `rocket_soc.bit`.

To flash this bitstream onto your FPGA you have to start *impact* by either clicking on *Configure Target Device* in ISE or run:

```
$ source /opt/Xilinx/14.7/ISE_DS/settings64.sh
$ impact&
```

In *impact* first click on *No* or *Cancel* on every popup window. Then doubleclick on *Boundary Scan*, press *Ctrl-I* to initialize your cable to the FPGA and click *No* and *Cancel* on the two popups. Rightclick on the chip symbol labeled *xc6vlx240t*, select *Assign new configuration File...*, and select the file `rocket_soc.bit`. Finally rightclick the chip symbol again and select *Program*, then *Ok* and after a loading bar a blue box should appear saying *Program succeeded*.

0.3.4. Compiling programs

To compile a program you need to have the environment set up correctly (see Section 0.3.1). If you want to start quickly you can use the example templates or else follow the manual guide which allows you to create an approximate application from scratch. This guide has optional steps which you can ignore if you do not plan on using the LUT. These paragraphs are marked with the *LUT* prefix.

0.3.4.1. Examples can be found in the directory `templates/`: *lut-application* and *alu-gaussian-application*. The *lut-application* demonstrates gamma correction for images and the *lut-gaussian-application* implements a 3x3 Gauss filtering algorithm. We have chosen very different image manipulation algorithms as we think the effects of approximation can be visualized best with image comparisons. For details on their inner workings and evaluation results, see the PACO webpage. **TODO (MUST):** insert precise url

0.3.4.2. Manual steps The manual guide has two mandatory steps:

1. Creating and compiling `main.c`
2. Linking everything

If you plan to use the LUT, two additional steps are needed:

1. Generate LUT configuration

2. Generate lut-startup code.

First we create our working directory which we call *\$WORK* here and should be exported into a shell-variable:

```
$ export WORK=/path/to/working/dir/  
$ mkdir -p $WORK && cd $WORK
```

Create and Compile main.c: First create main.c like a normal C program which should at least contain a *main()* function. If you want to communicate between your Linux PC and the FPGA you need to include the header file *uart.h* from *RocketLib*. For more information on how to use the *RocketLib*, see Section 0.4.3. If you want to use the approximate ALU, you can use the “approx” decorator to mark variables as approximable. If you want to approximate a function using the LUT you can mark it by using the approx decorator on the function. An example looks like this:

```
/* include the RocketLib header */  
#include <rocket/uart.h>  
  
/* this function will be approximated by the LUT */  
int approx(strategy=lut) foo(int a)  
{  
    return a*a;  
}  
  
int main()  
{  
    /* this variable can be approximated and the PACO compiler will aim for a  
       precision reduction of 2 (least significant) bits for the last state  
       of this variable. For details, see Developer's Guide, compiler section */  
    int approx(neglect_amount=2 relax=1 inject=1) approx_var, approx_result;  
    int result, b;  
  
    /* this will result in a LUTE instruction */  
    result = foo(b);  
  
    approx_var = 2;  
    /* this will create an add.approx instruction */  
    approx_result = approx_var + 5;  
    /* we call RocketLib to print characters back over the UART */  
    uart_println("Program_ran");  
    return 0;  
}
```

To compile this into an assembly file *main.S*:

```
$ cd $WORK  
$ clang -Iinclude -I$RISCV/include -I$RISCV/riscv64-unknown-elf/include \  
        -std=gnu99 -static -fno-common -emit-llvm -S main.c -o main.ll
```

This will create a llvm-IR intermediate file called main.ll and if you have used the LUT, an input file for the LUT-compiler. The file will have a UUID (universal

unique identifier) as a name. This looks for example like this: aab80a96-8016-11e6-a47e-00226817fba6.input.

Finally we need to create an assembly file from the *llvm-IR* using `llc`:

TODO (MUST): does this aab80a96-8016-11e6-a47e-00226817fba6.lut0.input if yes then provide consistency in above text, this is

```
$ llc -march riscv -mcpu=Rocket main.ll -o main.s
```

LUT: Generate a LUT configuration These steps generate a configuration for the LUT using the LUT-compiler. First you need to create the file *default.arch* which describes to the LUT-compiler how the LUT hardware was configured during synthesis. This file should look like this:

```
delay_addressTranslator = 0
base_bits = 48
delay_controller = 0
incline_bits = 32
delay_inputDecoder = 0
inputWords = 3
interpolationBits = 12
delay_interpolator = 0
plaInterconnects = 204
segmentBits = 7
selectorBits = 9
```

Second, you need an input file here called *lut0.input*, which should have been created by the clang-compiler in the previous step. If you want to tweak this file yourself or you do not want to rely on clang the file looks like this:

```
name = "lut0"
numPrimarySegments=6
bounds = "(0,16777215)"
segments = "uniform"
approximation = "linear"

%%

foo int -> int

%%

int foo(int a) {
    return a*a;
}
```

If you want further information on how the input and arch files are specified, look at Section 3.2.4.

To create a compilable array containing the data for the LUT:

```
$ cd $WORK
$ riscv-lut-compiler --arch default.arch -C lut0.input
```

This will result give in the file *lut0.c*.

LUT:Generate Startup Code *The startup code is needed to load the configuration data created in the previous step into the LUT hardware. This code can be generated automatically by running:*

```
$ cd $WORK
$ riscv-lut-startup -a default.arch lut0 -o lut-startup.s
```

Linking everything together: This step produces the executable which can be run on the FPGA. This step needs an assembled file *main.S* and, if you are using the LUT, a LUT data file *lut0.c* as well as the LUT startup code file *lut-startup.s*, generated in the previous step is needed. Additionally you need a linker script, called *main.ld* here:

```
OUTPUT_ARCH( "riscv" )
SECTIONS
{
    /* text: test code section */
    . = 0x10000000 ;
    .text :
    {
        *(.text)
    }
    /* data segment */
    .data : { *(.data) }
    .sdata : {
        _gp = . + 0x800;
        *(.srodata.cst16) *(.srodata.cst8) *(.srodata.cst4) *(.srodata.cst2) *(.srodata*)
        *(.sdata .sdata.* .gnu.linkonce.s.*)
    }
    /* bss segment */
    .sbss : {
        *(.sbss .sbss.* .gnu.linkonce.sb.*)
        *(.scommon)
    }
    .bss : { *(.bss) }
    /* thread-local data segment */
    .tdata :
    {
        _tls_data = .;
        *(.tdata)
    }
    .tbss :
    {
        *(.tbss)
    }
    /* End of uninitalized data segement */
    _end = .;
}
```



```
}
```

To generate an executable we will use `gcc` as a linker. That means we can provide our LUT configuration as a C-file. Run:

```
$ riscv64-unknown-elf-gcc lut-startup.s main.S lut0.c -o main.elf -L$RISCV/lib \
-lrocket -nostdlib -nostartfiles -Tmain.ld
```

This creates an ELF-executable file *main.elf* which can be executed on the FPGA. A how-to for that can be found in Section 0.4.3

0.4. Run PACO Approximate Applications

0.4.1. Simulate using the C Emulator

The C Emulator allows you to run programs on a virtual version of the Rocket CPU. The CPU registers are available, as well as any instructions for which hardware description has been provided for. In contrast to the Quick EMUlator (QEMU) described in the next subsection, the C Emulator is generated from the actual Chisel code describing the Rocket CPU core and is cycle-accurate. This means it allows you to test your hardware for logical correctness without synthesizing it and benchmark your applications much more precisely in a virtual environment than with QEMU.

Getting it to run: The C Emulator must first be generated from the Chisel code. For that, move to the emulator/ directory and call:

```
$ make CONFIG=PACOConfigCPP -jN
```

If this works, you can now test the emulator by calling:

```
$ make CONFIG=PACOConfigCPP -jN run-asm-tests
```

Make will link assembly tests assembled to `.hex` into the output/ directory from isa/ and run them, generating `*.out` files in the output/ folder. Prerequisite is of course that the test assembly files have been generated correctly (see Section 0.3.1). The sources for the test assembly files are in the directory tree under isa/. The base riscv-tests directory is very crowded, but contains both a Makefile and README.md. Assembly of the tests is described in the RISC-V tests section (2.2.1).

Additional sources of information:

- If running the emulator fails, also look into Section 2.2.1 and Section 2.2.2.
- Use of the C Emulator is further described in a README.md in the rocket-chip/ directory, but not its role and interdependencies.

0.4.2. Simulate using QEMU

QEMU² allows you to test your approximate programs for logical correctness without having to specify hardware. Drawback: The QEMU emulator cannot be generated

²<http://qemu.org/>

from your hardware specification, giving you no guarantees for parity of the emulator and hardware specification.

QEMU is mentioned here because it also allows you to test newly generated approximate applications, primarily for experienced users of QEMU. Explaining QEMU setup in the User Guide for non-QEMU-experienced users would take too much space, but that can be found in the Developer's Guide, Section 2.2.3.

The following QEMU guide assumes that you have set up your QEMU system and have included the *Berkeley boot loader*, *linux kernel*, and *initrd* with your program. The path to these files will here be referred to as: *\$bbl* for the berkeley boot loader, *\$linux* for the linux kernel and *\$initrd* for the initrd.

To simulate your program first start qemu and boot up linux by executing:

```
$ qemu-system-riscv -kernel $bbl -append $linux -drive file=$initrd,format=raw -nographic.
```

Afterwards you are greeted by a login screen where you login using the username 'root' without password. In your new command prompt you can start the program which you have put into the initrd.

If you want to follow the control flow of QEMU you can enable logging by using the parameter *-d* for the qemu command line. Afterwards you can specify comma-separated additional arguments as listed below:

- *in_asm* shows the target assembly code for each compiled basic block
- *exec* shows the execution trace before each executed basic block. (This should be combined with *nochain*)

Example:

```
$ qemu-system-riscv -d in\_asm, nochain, exec -kernel ...
```

If you want to print this information to file, use the parameter *-D* plus path to the file. If you want to debug your program in qemu using gdb³ you need to start qemu using the additional command line arguments *-S -s*. This sets up a gdb server which halts the guest program at its start point and waits until a gdb connects. To connect with gdb, start up gdb with your program from initrd as a parameter (e.g. *riscv64-unknown-elf-gdb yourprogram.elf*) and then type: *target remote localhost:1234*. From there you can use gdb the same way as a native debugger.

0.4.3. Running Programs on an FPGA Instantiation

There are three prerequisites to running an executable on the FPGA:

- have the environment installed (see Section 0.3.1),
- have a CPU instantiated on the FPGA (see Section 0.3.3),

³ <https://github.com/mythdraenor/riscv-gdb>

- and have a compiled program (see Section 0.3.4).

This section answers two problems:

- How can you load and run an approximate application on the FPGA?
- How can you communicate with the approximate application?

1. How to load and run a program? Loading and running is done using the `riscv-uart-flash` tool over UART communication. For this section we assume your program is called `prog.elf`. It can be loaded onto the FPGA by running:

```
$ riscv-uart-flash -i prog.elf -w
```

This command will also start the execution of the program and anything written back via the UART will appear on the console which starts the tool. The command line option `-w` will tell the `riscv-uart-flash` tool to run until it receives a terminate signal from your program running on the FPGA. The next section will show how to terminate your program. Further information to `riscv-uart-flash` tool can be found in Section 2.2.4.

2. How to communicate with the program? The most common use case is to write data or status information back from the program running on the FPGA. This can be done by using the appropriate function from the RocketLib (see lib). To write back a string you can simply use the following code snippet:

```
#include<rocket/uart.h>
...
void foo()
{
    uart_println("bar");
}
```

If you want to write back a `uint64_t` you can use this code snippet:

```
#include<rocket/uart.h>
#include<rocket/strutil.h>
...
void foo(uint64_t a)
{
    char buf[64];
    int r = 0;
    r = wrstring(buf + r, "bar");
    r = wruint64(buf + r, a);
    uart_println(buf);
}
```

The code snippet uses a array `“buf”` with a 64-byte length to store the string to be printed. The variable `“r”` is used to identify the current index of the array. The function `“wruint64”` takes an array position and an integer, converts the latter into a string and writes into the array it at the given array position. It then returns the new

array position at the end of the written integer. The function “wrstring” works similarly for strings.

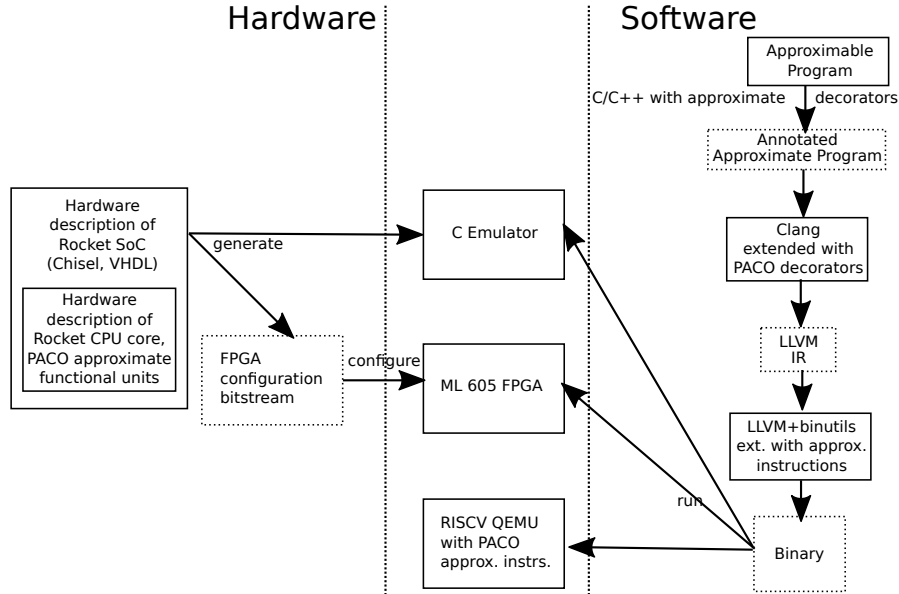


Figure 1: An overview over the workflow in the PACO tools. On the left side are the Hardware descriptions and how they can be transformed into three different platform. The right side shows the workflow to create a program to be run on these platforms

Part II.

Developer's Guide

1. Overview

The Developer's Guide for the PACO project is intended for programmers and hardware developers trying to introduce new instructions and hardware to the PACO core and improving the C++ compiler with approximate extensions. It is as a description of the implementation of the PACO core and the approximation systems as they are described in the design document. For an introduction into the PACO system and tools as well as step-by-step instructions for installation and preparation of the environment refer to the User Guide.

The Developer's Guide also contains a complete description of all the components of our project. This includes a rundown of the creation of the environment directory structure, an introduction to each major part as well as both a coarse-grained and a fine-grained description of modifications done to the original code. The original code being the code base serving as a starting point for our project: The RISC-V toolchain, Clang+LLVM, the Rocket Core and SoC as well as QEMU. Our project is divided into four principal components: The *environment*, the *compiler system*, the *approximate*

ALU and the *Lookup table*. Each component ties into an aspect of modifying the *Rocket SoC* which is the base system for our approximate processor solution. The Rocket SoC (see Appendix) itself is a system surrounding the *Rocket Core*, which is the actual base processor core.

Our project builds on these parts by adding two approximate computing units (Lookup table and approximate ALU) directly into the Rocket Core description, extending its the instruction set.

To exploit the respective instruction set extensions, we extend the binutils of the GNU Compiler Collection to translate the corresponding assembly instructions into machine code understood by our modified processor.

Additionally we modify the Low-level Virtual Machine (LLVM) compiler and the Clang C/C++/Objective C front-end to translate an extended version of C/C++ into assembly code targeted for our approximate extensions.

For a detailed description of the instruction set extension, assembly code additions and modifications to C/C++, please refer to the design document

CITATION NEEDED

The environment is the directory structure containing all the other components as well as tools required to interoperate them.

TODO (MUST): explain structure of this document (AFTER it was finished, duh)

TODO (MUST): remove the intermediate status of implementation and structure the contents

This document is structured as follows:

The rest of this overview section provides an overview of the state of the implementation as of 2016-05-29. We are currently implementing in three of the four larger divisions of implementation work:

- Environment
- Compiler
- Hardware (ALU)

For each of those divisions, we provide minimal descriptions of the smaller work packages identified during the design phase, as well as an estimation of its readiness. Independently of these divisions we provide a report of difficulties surmounted and our situation regarding the schedule.

The next three sections detail the changes we have made to the existing code base in each of the previously mentioned larger divisions of implementation and provide context to illustrate what has been done and what still needs to be done. Where deviations from the Design Document were advisable, we have described those deviations as well as the reasons. For full accountability, each description of a tool also contains information on how to use that tool to replicate our work.

The last section, “External Resources” lists the most useful links we found during our work on the PACO project.

1.1. Current state of implementation

TODO (MUST): migrate this section into future work

TODO (MUST): Update this section for the end of the official implementation phase.

TODO (MUST): figure: toolflows visualized

The implementation effort is split into four components: *Environment*, *Compiler*, *Hardware (ALU)* and *Hardware (LUT)*.

The environment implementation handles setting up the directory structure and source control, preparing tool flows and ensuring the correct behavior of all components.

The compiler part contains the modification of existing toolflows to allow the compilation of our language extensions to generate executable binaries.

This process itself is split into low-level and high-level compilation.

Hardware implementation handles the specification of CPU core and peripherals relating to the ALU and LUT approximation techniques.

In the following each part

is subdivided into tasks, listed together with a short description and status of completion.

CLARIFY: is this going to stay in the final version?

1.1.0.1. Environment Completion of environment implementation is required *as soon as possible* as all other parts depend on it.

component	status	comments
Preparation	Completed	Forking of all external resources and setting up directory structure
System administration	Completed	Acquisition of local and remote hardware, setting up toolflows
Source to FPGA flow	Completed	Ability to compile source code and execute it on the RISC-V core instantiated on an FPGA
RISC-V test cases	In progress	Compilation of RISC-V test cases and execution within the C simulator and FPGA instantiation
Implementation document (intermediate)	In progress	Mid-term implementation documentation as status report / reference manual
Implementation document (final)	Pending	Revision of the implementation document for final delivery
Demonstration application	Pending	Single application demonstrating the features of our system
Finalization / loose ends	Pending	Assembling the entire code base, documents and knowledge base into a single deliverable, composition of the project web site

1.1.0.2. Compiler The compiler is regarded as a *should*-level priority to be implemented during the entire implementation phase by a small sub team. Both the low-level and high-level compiler parts are not strictly necessary to utilize the approximation techniques implemented in the CPU core however they offer a simplification of using them.

component	status	comments
Approx decorator (parsing)	Completed	Parsing of approx decorator syntax and evaluating the key-value data
Approximate types (semantics)	In progress	Exploitation of type approximation in expression trees.
Approximate instructions (code generation)	Pending	Output of approximate instructions / intrinsics where approximate expressions were translated
Pragma integration	In progress	Integration of pragma directives for manipulating the translation process.
Approximate functions (semantics)	Pending	Analysis of approximation parameters on functions and invocations and assignment of LUT identifiers.
LUT extraction (code generation)	Pending	Extraction of source code and key-values for the LUT compiler to use.

1.1.0.3. Hardware: ALU The ALU is the first approximation technique to be implemented. This is done before work on the LUT is begun and thus is connected with overhead for getting used to the base system to be extended.

component	status	comments
Instruction decoding	Completed	Addition of control signals and assignment based on opcodes
Approximate addition	Testing	Execution of approximate addition
Approximate multiplication	In progress	Execution of approximate multiplication
Approximate floating-point operations	In progress	Execution of approximate operations in the floating-point unit
Power consumption estimation	In progress	Measuring and/or estimation of power consumption compared to an unaltered system

1.1.0.4. Hardware: LUT The LUT is implemented as the final component to the system and consists of both the hardware core itself and a compilation tool used to create LUT configurations out of source code.

component	status	comments
LUT core	Pending	The hardware core executing LUT computations
LUT integration	Pending	Insertion of the hardware core into the CPU template: Instructions and memory-mapped configuration.
LUT compilation: frontend	Pending	Processing of LUT compilation input and invocation of compilation backends
LUT compilation: backend	Pending	Numeric generation of LUT segments
LUT compilation: code generation	Pending	generating of assembly code from LUT segments

1.1.0.5. Problems During the implementation phase, a number of unforeseen problems occurred that required more time than favorable:

- Program execution on FPGA implementations: Out of the box there was no way to quickly load and execute programs on an FPGA implementation, instead the entire bitstream had to be re-synthesized each time a new program had to be executed.
This problem was solved by writing a custom bootloader and flashing tool which itself presented as a challenge due to erroneous implementation of the UART peripheral in the code base.
- Test cases / C simulator state: The RISC-V test cases fail with the C simulator due to conflicts in the version history. This problem is currently being addressed.

Due to a lack of experience in the respective parts, we expect to encounter more difficulties in the remainder of the implementation phase:

- LUT description and implementation: The execution of the Lookup table is designed to work within the processor pipeline. As the LUT hardware performs moderately complex computations, the data path delay may be too long to do so.
- Chisel abstraction: As chisel is a high-level hardware description language, its abstraction level may be too high for describing our hardware cores effectively.
- Clang compiler complexity: As it is a significant software project, modifying the clang compiler is ambitious at best. The semantic analysis, translation and code generation components are still to be implemented.

TODO (MUST): does the developer guide need to be provided with the details of timeframe and work assignemnt ? or better to have another section called as workflow management

1.1.0.6. Timeframe The implementation phase is set in the time interval from 2016-03-28 to 2016-07-25. During this time the implementation work is addressed in four tracks: Environment, compiler, ALU and LUT. The Environment was handled in the beginning as it amounts the foundation of all the other tracks. Compiler modifications are handled as a long-running track throughout the entire implementation phase and the ALU / LUT implementation are performed sequentially, initially focusing all efforts on the ALU.

Preparation of the environment was unexpectedly time-consuming as communication between our primary target, an FPGA-implemented Rocket SoC, and a host computer was particularly difficult. Currently all issues pertaining the communication with the FPGA have been solved.

Further tasks of the environment track solely consist in wrapping up the directory structure together with our documentation and resources as a clean deliverable in the end.

The implementation of the ALU has reached a point where the basic operations (addition, subtraction) are implemented and functioning, however estimated power gain is minimal. Therefore the ALU effort was reduced to implementing addition, subtraction and multiplication, postponing the approximation of floating-point operations.

The ALU implementation is set to be finished by 2016-06-13, directing the primary attention to LUT implementation thereafter.

TODO (MUST): the task descriptions can go to previous tables describing work packages for lut, repetition of lut tasks

The LUT implementation track is starting at 2016-06-13 and is split into the following tasks:

- Implementation and testing of the LUT hardware core.
- Simulation of the LUT hardware core in QEMU.
- Integration of the hardware core in the RISC-V pipeline, memory-mapping the configuration data.
- Fine-grained design and implementation of the Lookup table compilation tool

The compiler track has finished the implementation of assembly code generation for a subset of instructions with the remainder of instructions being trivial to implement. High-level translation, divided into the LLVM backend and the Clang frontend is still ongoing and successful implementation by the end of the implementation phase at 2016-07-25 is uncertain.

CLARIFY: should the workflow management really be there in developer guide this belongs more to project management stuff

Out of eight members, we initially assign two people to the Lookup table compilation tool, one person to the QEMU implementation and three members to the LUT hardware core (implementation, testing). The remaining two continue working on the compiler track.

As the QEMU simulation and the LUT compiler are expected to be completed within two weeks of work, the assigned people afterwards will be distributed among the LUT and compiler implementation tracks.

In total, six weeks of time are left for implementing the remainder of our modifications to Clang/LLVM, the LUT compiler, the LUT hardware core and its integration into the rocket core.

2. Environment

The environment is a tree of git repositories and automatically generated directories combined from different projects in which the developers are working in and the entirety of files reside.

A directory tree of the most important files and directories can be seen in figure 2 with a short description on their purpose. It starts from the directory *paco-env* and marks folders being the root of a git repository with a logo of a folder plus the git logo, e.g. *riscv-tools-src*. Directories with a folder symbol without git logo are tracked by the first parent being a git repository, e.g. the directory *py* is tracked by the git repository *riscv-tools-src*. Directories containing a folder symbol and a gear-wheel are build or installation directories and thus are not tracked by any git repository, e.g. *riscv-tools*.

2.1. Generating the environment

This section describes how the environment can be recreated from scratch. If you want information on setting it up for usage please look at the step by step guide in section 0.3.

2.1.1. Original code base

This section covers the steps taken to generate our code repositories out of the original code base.

The initial base for our project consists of a number of different projects: QEMU, Clang, LLVM, Rocket Chip, Rocket SoC. The Clang/LLVM projects were taken from a fork done for the RISC-V architecture itself.

The initial structure of our code base can be re-created by checking out the respective repositories found in table 3. Care must be taken to check out the repositories into the correct directory structure as depicted in figure 2.

2.1.2. Modified code

TODO (MUST): structure this section some parts are described in tools sections

CLARIFY: does some of the below context comes in user guide

This section describes what steps are required to prepare the environment after it was checked out from our code base.

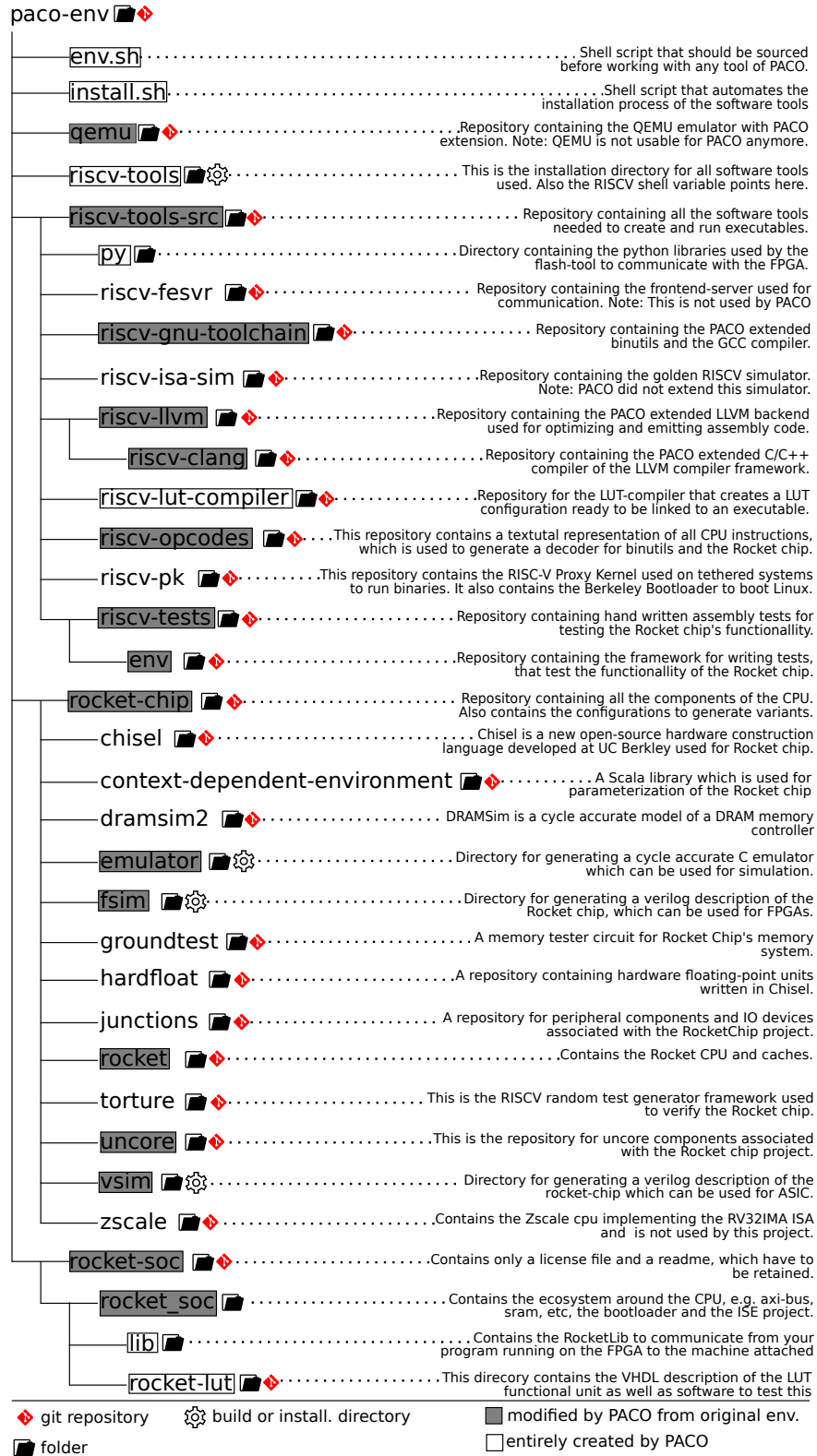


Figure 2: A directory tree of the most important files and directories with a short description on their purpose. Git repositories are marked with a git symbol, folders with a folder symbol, and build or installation folders have a gear symbol. Every directory surrounded by a solid white box is entirely written for PACO, every directory surrounded by a grey box contains modified code from the original repositories, and every other directory is untouched.

directory	repository name	original remote	commit hash
rocket-soc	rocket-soc	https://github.com/sergeykhbr/riscv_vhdl.git	547e74f
riscv-tools-src	riscv-tools	https://github.com/riscv/riscv-tools.git	419f1b5
riscv-tools-src/riscv-fesvr	riscv-tools-fesvr	https://github.com/riscv/riscv-fesvr.git	0f34d7a
riscv-tools-src/riscv-pk	riscv-tools-pk	https://github.com/riscv/riscv-pk.git	85ae17a
riscv-tools-src/riscv-tests	riscv-tools-tests	https://github.com/riscv/riscv-tests.git	c9022d2
riscv-tools-src/riscv-tests/env	riscv-tools-tests-env	https://github.com/riscv/riscv-test-env.git	566e47e
riscv-tools-src/riscv-opcodes	riscv-tools-opcodes	https://github.com/riscv/riscv-opcodes.git	b29f84f
riscv-tools-src/riscv-isa-sim	riscv-tools-isa-sim	https://github.com/riscv/riscv-isa-sim.git	3bfc00e
riscv-tools-src/riscv-llvm	riscv-tools-llvm	https://github.com/riscv/riscv-llvm.git	f11191e
riscv-tools-src/riscv-llvm/tools/clang	riscv-tools-llvm-clang	https://github.com/riscv/riscv-clang.git	5ca597d
riscv-tools-src/riscv-gnu-toolchain	riscv-tools-gnu-toolchain	https://github.com/riscv/riscv-gnu-toolchain.git	728afcd
qemu	qemu	https://github.com/riscv/riscv-qemu	8dac7fc
rocket-chip	rocket	https://github.com/ucb-bar/rocket-chip.git	90a73c6
rocket-chip/groundtest	rocket-groundtest	https://github.com/ucb-bar/groundtest.git	f411a73
rocket-chip/uncore	rocket-uncore	https://github.com/ucb-bar/uncore.git	7ff3c3e
rocket-chip/torture	rocket-torture	https://github.com/ucb-bar/riscv-torture.git	b54b3d0
rocket-chip/dramsim2	rocket-dramsim2	https://github.com/dramninjasUMD/DRAMSim2.git	0b3ee67
rocket-chip/zscale	rocket-zscale	https://github.com/ucb-bar/zscale	de370f6
rocket-chip/context-dependent-environments	rocket-context-dependent-environments	https://github.com/ucb-bar/context-dependent-environments.git	8671317
rocket-chip/chisel	rocket-chisel	https://github.com/ucb-bar/chisel.git	1631c45
rocket-chip/hardfloat	rocket-hardfloat	https://github.com/ucb-bar/berkeley-hardfloat.git	978b226
rocket-chip/junctions	rocket-junctions	https://github.com/ucb-bar/junctions	5138397
rocket-chip/rocket	rocket-rocket	https://github.com/ucb-bar/rocket.git	6c0e1ca

Table 3: Table containing the original repositories without any modification and the place where they should be checked out.

For a first installation please refer to the user guide's step by step section 0.3. This section will give shortcuts for rebuilding the tools after you made changes. In the following, unless otherwise specified, all instructions require setting up the PACO environment by sourcing the `env.sh` script. Furthermore all paths described are relative to the PACO environment directory.

If you need to enter this path or a subpath specifically, it is denoted as `paco-env/`.

2.1.2.1. RISC-V toolchains The principal toolchains for the RISC-V are located in the repository `riscv-tools` and are built by invoking the script `build.sh`. This causes all tool to be compiled. If you only did a change to a single component you can use the following shortcut. Lets say the tool we changed is `binutils`. Then we can do the following:

```
$ cd paco-env/riscv-tools-src/riscv-gnu-toolchain/build/build-binutils-newlib/  
$ make && make install
```

This will incrementally build only the `binutils` and install them into the regular place.

2.1.2.2. Rocket core The rocket core is a single Verilog code file translated from Chisel specification in the repository `rocket-rocket`. To build it, use the makefile Makefile:

```
cd rocket-chip/fsim  
make CONFIG=PACOConfigFPU
```

Afterwards, the Rocket core file `Top.PACOConfigFPU.v` must be copied into the Rocket SoC, replacing the file `Top.GnssConfigNoFPU.v`.

If you created your own config you have to change the `CONFIG` variable to that config name. For example, if you called your config class `PACOConfigNoFPU` you have to invoke it by running:

```
$ cd rocket-chip/fsim  
$ make CONFIG=PACOConfigNoFPU
```

The resulting file will be called `Top.PACOConfigNoFPU.v` accordingly.

2.1.2.3. Rocket SoC Whenever the Rocket SoC peripherals or the Rocket core itself were changed, the Rocket SoC must be re-built. See section 0.3.3 of the user guide for details.

2.1.2.4. Rocket SoC bootloader The bootloader starts up the Rocket SoC and waits for the `uart-flash-tool` for commands. If you need to change the bootloader you can do so in folder `src/`. It contains two important files: `main.c`, and `trap.c`. The former contains the function `_init()` which is the first function called in the bootloader and the latter contains the function `handle_trap()` which is called to handle traps.

To recompile the bootloader run the following:

```
$ cd paco-env/rocket-soc/rocket_soc/fw/boot/makefiles  
$ make
```

Afterwards the bootloader image can be found here `bootimage.hex`. To use it you have to copy it into the folder `fw.images/`, replacing the old `bootimage.hex`. The final step is to resynthesize the Rocket SoC (see section 0.3.3).

2.1.2.5. RISC-V python library The RISC-V python library will be installed under `py`, which will be found by the python interpreters through the `PYTHONPATH` variable. The `PYTHONPATH` variable is set by `env.sh`.

2.1.2.6. RISC-V tests If you changed or added tests to the `riscv-tools-tests` repository you can use the same shortcut as for the RISC-V toolchain from Section 2.1.2.1, to incrementally build them by running:

```
$ cd paco-env/riscv-tools-src/riscv-tests/build
$ make && make install
```

2.1.2.7. QEMU If desired, the RISC-V fork of QEMU can be built as follows. Note that explicitly stating the path to a python 2 interpreter is optional on most systems and added here as a precaution in case the default python interpreter points to python 3.

```
mkdir qemu-build
cd qemu-build
../qemu/configure --target-list=riscv-softmmu --enable-debug
--python=/usr/bin/python2
make
```

2.1.2.8. Virtual Machine

TODO (MUST): download external resources and refer to them here

To build virtual machine images to be run via QEMU, a number of components are required. Building these is explained in this paragraph.

First we need to build the Linux kernel. If this is not desired, a pre-built version can be found in `vmlinux`. To build it yourself, create and enter a new directory and execute:

```
curl -L https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.1.17.tar.xz |
tar -xJ
cd linux-4.1.17
git init
git remote add -t master origin https://github.com/riscv/riscv-linux.git
git fetch
git checkout -f -t origin/master
```

This will prepare the linux kernel for compilation. Now configure and build it with:

```
make ARCH=riscv defconfig make -jN ARCH=riscv vmlinux
```

The resulting file `vmlinux` can now be used instead of the pre-built file in `vmlinux`.

The next component required is the initial ram disk (`initrd`). Again, a pre-built version is available in `rootfs.ext2`. To build it manually check out the necessary tools into a directory of your choosing:

```
git clone https://github.com/a0u/buildroot.git
cd buildroot
```

Now generate a configuration and build it:

```
make riscv64_defconfig
make -jN
```

After completion the resulting file `output/images/rootfs.ext2` is the initrd you can use instead of the pre-built version in `rootfs.ext2`.

Problem: Creating the configuration fails

Solution: Manually create the configuration by invoking:

```
cd buildroot
make menuconfig
```

This will open a menu in which the following settings should be applied:

- Under **Target options/Target Architecture**, select **RISCV64**.
- Under **Toolchain**, set:
 - **Toolchain type** to **external**,
 - **Toolchain** to **custom**,
 - **Toolchain path** to **paco-env/riscv-tools**,
 - **Toolchain prefix** to **\$(ARCH)-unknown-linux-gnu**,
 - **External toolchain gcc version** to **5.x**,
 - **External toolchain C library** to **glibc/eglibc**
 - **External toolchain kernel headers series** to **3.14.x**

Finally, to run the system, a Berkeley bootloader is required. To forego this step, a pre-built version can be found in `bbl`. To generate it manually, create and enter any directory and run:

```
CLARIFY: the prefix was set to RISC/riscv-64-unknown-elf, but there is
a riscv-64-unknown-elf directory under pac0-env/riscv-tools. figure out
which one is correct.
```

```
paco-env/riscv-tools-src/riscv-pk/configure
--prefix=$RISC/riscv-tools/riscv-64-unknown-elf --host=riscv64-unknown-elf
make
```

The resulting file `bbl` should serve as a viable replacement for `bbl`

2.2. Tools

TODO (MUST): divide the section into original code base and modified code base similar to environment section for sake of consistency

This section presents the set of tools we used for developing and testing modifications and applications for the RISC-V Rocket core.

TODO (MUST): figure: graph showcasing the tool flow for the hardware generation and verification process including risc-v tests executed on the simulator and fpga generated by a common chisel source

2.2.1. RISC-V tests

The RISC-V tests are a set of assembly programs you can use to test the correctness of a RISC-V implementation. You can also use them as benchmarks to test the performance of a RISC-V implementation.

They are situated as a sub-repository to the riscv-tools, in directory riscv-tools-tests.

For instructions on compiling the test suite refer to section 2.1.2, in particular paragraph 2.1.2.6

Changes:

- PACO added test programs for the approximate ALU instructions introduced for the PACO core. They are located alongside the precise tests in their source directories and named `<operation>_approx.S`.
- From the initial git repository the RISC-V tests did not run in the emulator. There were two problems from the test side:
 - The tests were not compiled from the emulator Makefile as the rocket-chip README.md promised. Thus, the tests were run with outdated versions of the machine code and failed with the message
Assertion failed: DCache exception occurred - cache response not killed.
That message could only be read if the Makefile was modified so it would not delete the file the error messages were redirected to after any failed test. This problem could be remedied by compiling the RISC-V tests via the Makefile in `riscv-tools-src/riscv-tests`.
 - The version of the tests that was installed with the RISC-V tools was not made for the version of the rocket chip that was installed. This meant incompatible machine code and thus failure of almost all tests. This has been mostly resolved by reverting riscv-tools-tests to commit `fc000796c11f84a5e1997c67fcfac751aa64a916` and reverting commit `fe978bbd63d8044eda9ad029dedfddfe1137a7e`. The fcvt and fdiv instruction tests still do not succeed, for unknown reasons. They are not essential to our testing though.

Usage with the C emulator is described in the C emulator section (2.2.2), the tests are not very useful on the FPGA currently, because they individually need to be modified to return exit state via UART.

2.2.1.1. Testing approximate instructions/hardware Test assembly programs have been created to test the approximate ALU instruction. We will use the test of the approximate add instruction as an example: It can be found as `add_approx.S`. The

header file `test_macros_appr_scalar.h` contains macros helping to create tests comparing a calculated result to a pre-calculated range of approximately correct values. The `add_approx.S` program also contains detailed instruction on how to create test programs for other approximate instructions.

To have this test program compiled with the rest of the test suite, add a line to `Makefrag` (obvious how to do it.)

2.2.2. Rocket Chip C Emulator

The role and possible uses of the C Emulator are described in the User Guide (section 0.4.1). Some additional details may be of use for hardware development:

- The C Emulator is more useful for hardware description testing than for software testing. If it compiles tests and benchmarks correctly, that is a good sign that the logic you have described works as intended. It does not mean that the timings all work out. Almost all detail at the electrical level is abstracted away from. Up to a magnitude of 100 thousand instructions can be emulated relatively quickly. To test larger programs, QEMU will probably be a better fit.
- The C Emulator also allows you to quickly prototype new instructions and the hardware supporting them and test them within a few minutes, which would otherwise take you a lot longer since you have to synthesize for the FPGA. QEMU is unsuitable for this since its emulation is not derived from the hardware description in Scala.
- Finally, the C Emulator allows you to benchmark your hardware precisely, something that will not be possible with QEMU in the near future.

Running the C Emulator to test your hardware (or assembler) requires a Scala description of the chip that can be used to generate the C Emulator, the assembly test programs that should be simulated on the chip, and an assembler that can generate machine code from the test programs. For the PACO core, all of these are included in the git.

From the initial state of the PACO git, these are the changes you have to make so it actually runs:

- in `rocket-chip/`:

```
git revert 4389daf2eef084b518c847f539d3f2db2156ec89
```

This commit originally introduced patches by `sergeykhbr` to get the Rocket chip to run on the type of FPGA used by our group.

- harmonize the versions of the rocket tested and the `riscv-tools-tests`.
It appears that during conversion of the git with submodules to our structure with multiple individual git repositories a desynchronization occurred, of the tests with the version of the chip.

2.2.2.1. Changes The changes made to the emulator are minimal: The emulator core to be made has been changed in the Makefile, to the one specified in the Scala description of our core: `PACOConfigCPP`. More substantial are the changes made to the assembly test programs (see section 2.2.1) so they can test approximate instructions. Usage of the C Emulator: Usage as well as location of README files is described in the User Guide (see section 0.4.1).

2.2.3. QEMU

QEMU is an open source virtual machine emulator that was extended by the original developers of the RISC-V CPU to support their architecture.

It is supplied in repository `qemu` and built as part of the environment (see section 2.1.2, paragraph 2.1.2.7).

Further information and resources for QEMU can be found in the appendix.

2.2.3.1. Usage QEMU is used by supplying a virtual machine file which is simply a hard drive image being booted by QEMU itself.

To use QEMU with the RISC-V / PACO core, an image is already provided

CITATION NEEDED

This can also be generated manually by preparing a linux kernel, adding any test programs into an initial ram disk (`initrd`) and loading everything with a berkeley bootloader. Instructions on preparing each of these components can be found in section 2.1.2, paragraph 2.1.2.8.

For the following description the QEMU executable must be located in the `PATH` environment variable and we work from a directory containing all components: The linux kernel (e.g. from `vmlinux`), the `initrd` (e.g. from `rootfs.ext2`) and the bootloader (e.g. from `bbl`).

Having prepared the required components as discussed above, a program to be tested has to be added to the `initrd`. Let the program in question be called `test-program`. This is then inserted into the `initrd` by mounting it and adding the respective executable to the file system:

IMPROVE: reformulate (hard to understand)

```
mkdir -p rootfs && mount -o loop rootfs.ext2 rootfs
cp test-program rootfs/root/ umount rootfs
```

Having added a program to the `initrd`, it can now be tested through the virtual machine. To execute it, run:

```
qemu-system-riscv -kernel bbl -append vmlinux -drive
file=rootfs.ext2,format=raw -nographic
```

This will boot the virtual machine. Eventually the entry of a user name is required. Enter `root` (no password required). When logged in, the test program can simply be run by:

```
/root/test-program
```

TODO (COULD): loading compiled programs from a running machine

2.2.3.2. Modifications done So far we added simulation for the addition and subtraction instructions (`add.approx`, `sub.approx`). For this we added a single function `gen_approx` in the appropriate file (`translate.c`) which selects the instruction to be executed based on additional bits supplied to it.

Problem: No additions for `mul.approx`?

This function first generates temporary registers to duplicate the operands, determines how many bits to neglect and copies them from newly added shadow registers. The actual addition and subtraction instructions are generated as ordinary instructions for the target, operating on the temporary registers.

The shadow registers are two registers `operand_shadow[2]` defined in the `RISCVCPUState` structure that remember the inputs to arithmetic instructions in order to draw neglects bit from them.

Therefore the operands of the regular addition and subtraction instructions are also written into these shadow registers in the appropriate functions `codegen_arith`, `gen_arith_w`, `gen_arith_imm`, `gen_arith_imm_w`, `gen_branch`, `gen_load`, `gen_store`, `gen_jal`, `gen_atomic`, `gen_csr_htif` and `gen_system`.

To help in decoding, two macros were also added to the `instmap.h`: `MASK_OP_APPROX` retrieving the operation to approximate (`funct4`) and `GET_APPROX_BITS` retrieving approx bits from an opcode.

2.2.3.3. Further modification QEMU uses the concept of a binary translator, called Tiny Code Generator (TCG), which is used to translate assembler instruction of the guest architecture to assembler instructions of the host architecture on the basis of a basic block

CITATION NEEDED

The translated blocks are then saved and put into a hash table which uses the guest program counter (`pc`) as a key. If a guest `pc` is encountered twice the basic block does not need to be translated a second time. As all good compilers TCG uses an intermediate representation called micro-ops. So each extension to a guest only needs to generate micro-ops. A list of available micro-ops can be found in the file `README` and each micro-op is generated by using `tcg_gen_<insn_name>` function calls.

The CPU state of the guest is represented by a structure called `RISCVCPUState` and has all important registers defined in the member `active_tc`. The members of `active_tc` are then mapped to TCG in the method `riscv_tcg_init` in `translate.c` and can afterwards be used by micro-ops.

The translation happens in the file `translate.c` and starts in the function `gen_intermediate_code_internal` which runs in a loop and loads a new instruction from guest memory and tries to decode it in the function `decode_opc` until it reaches a branch instruction which ends the basic block. The `decode_opc` function decodes the binary instructions similar to a disassembler and generates corresponding micro-ops. Adding a new instruction occurs in three steps: Adding a new decoding mask, adding new major opcodes to the list of opcodes and adding decoder / instruction translation.

2.2.3.4. Adding new decoding masks Decoding masks are defined in `instmap.h`. It contains a set of enums and macros to extract information from opcodes by means of bit operations.

2.2.3.5. Adding new major opcodes Opcodes are defined in the same file (`instmap.h`) as part of the first enum. A new opcode consists simply of a new entry in this enum.

2.2.3.6. Adding decoder / instruction translation All the code pertaining decoding and translation of instructions is located in the file `translate.c`. Translating a new instruction is done by first defining a new function for that instruction, prefixed with `gen_` by convention. As arguments it accepts a disassembly context, the program counter and the arguments as expected by the corresponding instruction. This function is called out of the `decode_opc` of the same file in which a new case is simply added to the switch structure with the instruction's code as label. Within the newly defined generator function, target instructions are generated by `tcg_gen_` methods which in turn interact with registers defined for the target. These registers are either defined in the processor state (`RISCVCPUState`) or they are temporary registers used with `tcg_temp_new` and `tcg_temp_free`. Whether they are temporary or not, register contents are copied using `gen_get_gpr` and simply handed to target instructions via `tcg_gen_` functions. To add a new register to the processor, it must first be defined in the `RISCVCPUState` structure. To make it usable, it must be allocated in the `riscv_tcg_init` method using the call `tcg_global_mem_new` and defined again as a global static in the file `translate.c`.

2.2.4. UART debug interface (flashing tool)

The UART flashing system consists of a custom bootloader written for the Rocket SoC that is embedded as a ROM into the Rocket SoC FPGA bitstream. This bootloader offers a binary command shell on the FPGA's UART port exposing an interface into the SoC's main memory.

Through this interface any RISC-V program can be loaded and executed without having to build a new bitstream.

By the time the project group started no method for accessing the memory of the SoC existed that could be made operational, leaving a built-in ROM for a bootloader the only entry point for code to be tested.

For changes in the bootloader code to become effective, the entire FPGA bitstream had to be re-synthesized causing delays on the order of 60 minutes for each iteration in testing and debugging.

Our custom bootloader in conjunction with a Python script running on a host computer connected via UART enables us to write into the SoC's main memory and setting the program counter to its starting address, greatly reducing the time required to test a program.

2.2.4.1. UART interface The UART interface of the custom bootloader expects a single byte identifying the command to be executed followed by additional arguments depending on the command. Currently the following commands are understood:

command name	code	arguments	description
NOP	00h	none	Does nothing. Used for synchronizing by sending an arbitrary number of zero bytes.
SYNC	10h	none	Deprecated. Sends a simple response on the UART to indicate synchronization has set in.
BLOCK ADDR	21h	u32 addr	Sets the address of the next block transfer and reports it back as an eight digit zero-padded hexadecimal number.
BLOCK WRITE	22h	256 bytes	Writes a single block of data into RAM starting at the previously configured block address. After completion, reports the CRC-32 of the received data as an eight digit hexadecimal number.
BLOCK CRC	23h	none	Computes the CRC-32 of the 256-byte block starting at the previously configured block address and reports it as an eight digit hexadecimal number.
EXEC	42h	none	Terminates the UART command shell, prints a confirmation text on the UART and jumps execution into the beginning of the RAM afterwards.
LED	51h	1 byte	Uses the argument as a bit mask controlling the GPIO LEDs on the FPGA board.
DIP	52h	none	Reports the status of the FPGA board's DIP switches as a 32 bit mask as an eight digit hexadecimal number.

Upon startup, the bootloader reports its readiness by printing a greeting in the form `PAC0 Rocket SoC bootloader version 12` onto the UART. Depending on the setting of DIP switch 2, the bootloader then either loads another ROM into the FPGA RAM and executes that program (DIP switch set high) or enters the aforementioned UART command shell (DIP switch set low).

To signal the readiness of the uart shell, immediately before entering it the line `Boot (uart ready)` is printed on the UART.

2.2.4.2. UART Out-of-band reset To allow resetting the CPU without having to manipulate the FPGA physically or even cut power, the UART hardware core was modified to accept out-of-band communication.

To achieve this, the UART was configured to use a parity bit, in-band communication occurring with even parity and an odd parity being interpreted as out-of-band data.

This data is interpreted directly by the hardware core and signals a reset signal upon receiving of the byte sequence `DEh ADh BEh EFh`. This reset signal is or-combined with the push-button reset signal of the SoC.

2.2.4.3. Flashing tool To facilitate the usage of the UART interface, a Python script was written that is installed in riscv-uart-flash.

It was written concurrently with the development of the bootloader, thus it supports a number of deprecated versions. By the time this document was written, bootloader version 12 was the most up-to-date one.

The primary function of the tool is to reset the connected RISC-V CPU, use the bootloader's UART shell to transfer a single program, start its execution and enter an interactive shell for the user to interact with the program via UART.

The general usage of the tool is invoked by executing it with:

```
riscv-uart-flash -i -w <program>
```

Where **<program>** represents a file name pointing to the program to be loaded.

This can be either an ELF file or a binary dump of its sections⁴.

The **-i** command-line option instructs the tool to enter an interactive shell after flashing is done, otherwise it would just exit quietly.

-w instructs the tool to terminate after an exit code was received from the running program via UART. For further information on this behavior, refer to section 2.2.5.

By default, UART communication is initiated on device **/dev/ttyUSB0** and baud rate 115200. To change these settings, command-line options **-p** and **-b** can be used, respectively.

Further information on usage of this tool can be found by invoking it with **--help**.

2.2.4.4. Modifications The original UART core located in `nasti_uart.vhd` was not

viable for two-way communication for a number of issues.

Problem: By using busy waiting while reading from the UART and thus querying the status register, the data cache also fetches the data register, popping new bytes from the internal FIFO.

Solution: The FIFO popping semantic was removed from the reading of the data register. Instead, the current value in the receiver FIFO is now removed once the corresponding sequence number is written into the status register.

Furthermore, the current data value and the corresponding sequence number were added to unused bits in the status register, reducing the number of words to be read for UART communication.

Problem: Reading bytes via UART presented extremely high loss rates.

Solution: By offsetting the sampling of bits within a UART frame into the center of the symbol interval instead of the beginning, clock drift and jitter were remedied and the loss rate was reduced to a negligible amount.

The usage of block-based data transfer and checksums also aides in the elimination of transmission errors.

⁴ The binary dump can be obtained by invoking
`riscv64-unknown-elf-objcopy -O binary <input file> <output file>.`

2.2.5. Rocket SoC Runtime Library

A C library has been written that facilitates writing programs that can be run on an instantiation of the Rocket SoC FPGA. This library encapsulates several commonly used features that are mentioned below. It is delivered as a part of the rocket-soc repository in the directory `lib`. The directory consists of the include files library (under `include/`), the source files (under `src/`) as well as a makefile used for compiling the library and installing it into `riscv-tools` directory. This is done by invoking:

```
make
make install
```

In addition to the runtime library, a set of example programs can be found in the `templates/` subdirectory, which serves as templates for a custom application.

2.2.5.1. Program termination In most cases, programs on the FPGA are supervised by a program running on a host machine, communicating via UART. In order to signify that the FPGA program has terminated, a special sequence of bytes is sent by the FPGA, consisting of three bytes `00h XXh 0ah`, where `XXh` is a one-byte exit code. This exit code should be 0 to signify successful program execution and a non-zero number to signify its failure.

Under no circumstances may the exit code equal 10 as this may be misinterpreted as a new line character.

2.2.5.2. Library Components *Currently the library in the rocket-soc repository contains the following parts:*

- *UART communication (include `rocket/uart.h`): Methods for reading and writing text or binary data via the Rocket SoC's UART port. Additionally some methods to output integers in hexadecimal representation and to signal program termination are also provided.*
- *String composition methods (include `rocket/strutil.h`): In order to compose strings (e.g. for UART transmission), several methods are provided that write characters into a pre-allocated buffer similar to the behavior of the `sprintf` method.*
- *Clock cycle counting (include `rocket/util.h`): A simple method that returns the number of clock cycles since the startup of the FPGA, as a 64 bit integer.*

2.2.5.3. Template Applications All template applications come with a makefile supporting at least the targets `all`, `clean` and `run`. The `run` causes all necessary compilations to be executed and then downloads the program onto the FPGA via the UART; running it in an interactive shell.

The UART port `/dev/ttyUSB0` is used by default. This can be modified by specifying the UART environment variable, e.g. as such:

```
UART=/dev/ttyUSB1 make run
```


Available templates are:

- **basic-application:** This is a basic application showcasing UART communication via a line-by-line echo service as well as transmitting exit codes via `uart_exit` upon receiving the termination command ('q').
- **timing-application:** This template demonstrates timing analysis based on clock-cycle counts by using the built-in clock cycle register.
- **lut-application:** This template demonstrates the use of the Lookup-Table based approximation hardware. In addition to the usual source files, a file `lut0.input` can be found in this directory. It represents the configuration of a single lookup-table to be integrated into the program. From within the `main.c`, this lookup table can be queried using the `LUTE` macro. To add further lookup tables, simply add more `.input` files and add them to the `LUTS` variable in the Makefile.

2.3. Modifications

This section gives an overview of the changes that were made to the original code in the environment as such. This does not include modifications to specific components as discussed in later sections of this document.

TODO (MUST): list parts that were added / altered and refer to sections detailing the purpose of those changes

2.3.0.1. Makefile adjustments *To eliminate the potential for errors in the process of compiling and integrating the Rocket Chip into the Rocket SoC, the relevant makefile in Makefile was adjusted to use the PACO-specific chip configuration and to support the `install` target, which copies the generated verilog file into the right location for use by the Rocket SoC: `Top.GnssConfigNoFPU.v`*

3. Compiler System

The compiler system is a software solution that generates machine code files executable on the PACO core.

This task subdivides into three separate components: First, high-level code compilation translates C/C++ code into assembly instructions targeted for the PACO core as well as Lookup table descriptions.

Second, the lookup table compilation component accepts such descriptions and generates data sections in the form of assembly code describing the lookup table configurations.

Finally, machine code generation accepts assembly code and translates it into a single binary file ready to be executed on a PACO core implementation.

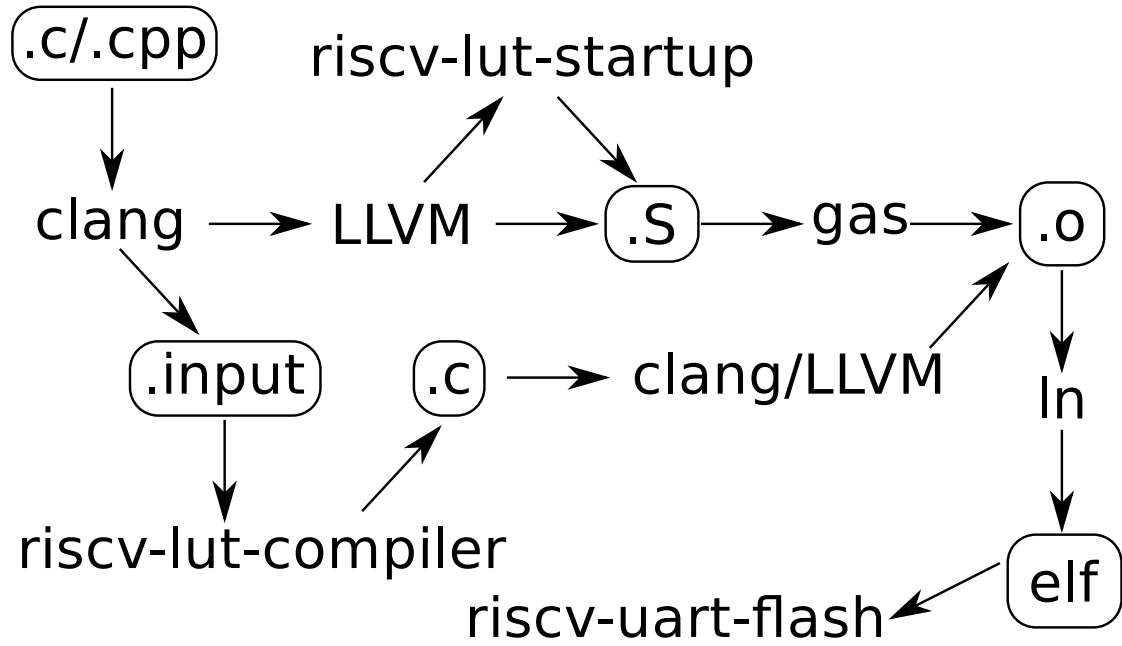


Figure 3: Toolflow for generating executable utilizing the PACO language extensions and hardware cores.

3.1. Original Code Base

The compiler system consists of several programs. The rocket core uses Clang as compiler front-end and LLVM as compiler back-end. GNU-binutils is generating the target code. To introduce approximation all programs are modified. To generate the configuration for the LUT hardware, the compiler system was extended with the LUT-Compiler (LTC). Since the LTC is a new tool, there is no basic implementation. Clang and LLVM were used at version 3.3 which was the current state for the rocket core at project begin. The interactions of all these tools are described in our design document A. A detailed description of the building process can be found here 0.3.1.2. The following sections describe each tool and the modification in detail.

3.2. Modified Code

3.2.1. High-level Code Compilation

High-level code compilation consists of translating the source code of an high-level programming language into assembly code. The following subsections give a detailed view on the modification within Clang (repository: riscv-tools-llvm-clang) and LLVM (repository: riscv-tools-llvm).

3.2.1.1. General Modifications To properly put the PACO compiler extensions in place within Clang, a number of general modifications were made. These adjustments helped to smooth the process of integrating the necessary extensions.

PACO Language Option The adding of a language option to Clang helps to encapsulate all features of the PACO extensions. This language option serves two purposes: First it offers the disabling of PACO extensions during Clang compile time. Two, it serves as an indication within the Clang source code for all modifications made for the PACO extensions. This indication enables programmers to identify parts of code that are responsible solely for handling PACO extensions. The language option was added simply by modifying the language options tabledata file (LangOptions.def).

PACO Keyword Definitions For better maintenance purposes all PACO specific keywords and constants are stored within a new created file (PACO.h). This provides an much easier process for adding and modifying keywords.

Preprocessor Macro To make the original compiler capable of understanding the PACO extensions, the preprocessor macro `_paco` was added according to the design document A. This macro is defined as an empty symbol in LangOptions.def and can only be used if the PACO language option is set.

Error Messages For outputting error messages, new errors with corresponding messages were defined. The errors of the parsing process are listed in DiagnosticParseKinds.def. Equivalent are the errors of the semantic analysis stored in DiagnosticSemaKinds.def. In the case of an error, a `Diag` object is called using the source location and an error flag as operands. Errors ,appearing within the code generation process, are not described in an own definition file. These errors are printed by using LLVM classes which contain error functions. The source location and the error message are handed as operands.

3.2.1.2. Parsing and Analysis of the Approx Decorator The approx decorator is the principal way of adding any approximation into C/C++ code. It is represented by a new declaration node within the abstract syntax tree which built by Clang.

Context Parsing the approx decorator begins with the designation of a new keyword, `approx`, in the keywords tabledata file TokenKinds.def. This keyword will be interpreted at declaration: To implement approximate types and functions, it is part of declaration specifiers, thus accepting the approx decorator anywhere types are expected: Type definitions, variable declarations, function definitions. This has been implemented.

To parse an approx decorator as part of a declaration specifier, a new case was added to the declaration specifier function in ParseDecl.cpp. In this case of the occurrence of

an **approx** keyword, the resulting decorator is registered within the declaration specifiers structure. After parsing, the declaration specifier is converted to a **Decl** object. To keep the **ApproxDecoratorDecl** as a member of the declaration, the **ApproxDecoratorDecl** is transferred to a new object, done in **SemaDecl.cpp**.

Approx decorators Approx decorators are represented in the abstract syntax tree as a new declaration node type. This node is defined in **Decl.h** as **ApproxDecoratorDecl** class. This class itself holds an array of **KeyValue** instances that each represent a single key-value pair as defined in the approx decorator. As such they have an identifier and one of two values: A **StringRef** in case of a string key-value or a **APValue** for numeric key-values. The latter is a real or complex integer or floating-point value. The parsing of the approx decorator itself occurs in **ParseDecl.cpp** in the method **ParseApproxDecorator**. This method parses all key-values enclosed within parenthesis following an **approx** keyword, before passing them to the semantic analysis where an approx decorator instance is created. While parsing the key-values, either a sequence of string literals or an expression are accepted as values. The semantic analysis of the key-values then extracts a single string or a number from those entities. To extract a number from an expression, it must be statically evaluable. If more than one **approx** is used in the same declaration, they will be combined and interpreted as one **approx** field.

Approx Decorator Key-Values During the process of creating a new approx decorator, new key-value objects are created. These objects are filed with the information from the source code. Before writing the information to these key-value objects, some testing is done. It is important that the given value matches the expected value of the used identifiers. For this purpose it is proven if the value is either a number or a string literal. These checks are done by parsing the key-values. For each legal identifier an own test is executed. In case of the **neglect_amount** key-value, it will be tested, if it contains a valid amount (2, 4, 7, 10, 15, 20, 27). It is also checked if the key-value of this identifier already exists. In case of existence, it will be overridden and a warning will be produced. For easier computations in later parts, the value of the **neglect_amount** will be translated into a **mask** value. Here the original identifier keeps untouched. In the case of using the **neglect_amount** and **mask** within one approx decorator will create an error. Key-value checking for LUT related keywords have not been implemented.

Preparations for the Lookup Table Compiler (LTC) For using the LTC, the approximated function needs to be written into a specific file. For invoking this process within the source code the keyword **strategy** is needed. The parsing process of this keyword and emitting of the new file is implemented in **ParseDecl.cpp**. Before the approximated function is written to this file, the LUT definitions (key-values) from the

approx decorator are parsed and stored in the beginning of the file. These definitions are separated from the function with "%" as separation markers. The name of the file has the format `uuid.input`, where `uuid` is an universal unique identifier which is used to identify the function. The tokens of this function are not consumed during these process, so Clang can still perform its semantic analysis of this function, because the LTC has no semantic analysis. This is the reason why the code of approximated functions are not emitted in code generation. Currently the printing of includes to the emitted output file is not implemented and it is necessary to insert them manually.

3.2.1.3. Parsing and Analysis of Pragmas

Context Pragmas are annotations in the source code configuring the translation process of other constructs such as approximate types.

Definition In Clang, pragmas are translated by the parser into a special token identifying which pragma it is, e.g. the code `#pragma paco combine` is represented by a single token of kind `pragma_paco_combine`. As such they are defined in `TokenKinds.def` via the `ANNOTATION` macro.

Lexing The actual parsing of pragmas is performed by pragma handler classes inherited from `PragmaHandler`, declared in `ParsePragma.h` and implemented in `ParsePragma.cpp`. For our pragmas we added `PragmaPACOCombineHandler` and `PragmaPACOIntermediateLiteralHandler`.

These handlers get invoked during parsing in their `HandlePragma` method which consumes a number of tokens and emits new ones depending on the pragma. To register the pragma handlers so that they get invoked in the parsing process, the `Parser` class gets instances of them as fields which are initialized in its constructor (and cleaned up in the destructor) in `Parser.cpp`. For our extension, the handlers only get initialized if the `PACO` language option is set. Disabling our pragmas altogether if it is not.

Parsing In the parser, methods `HandlePragmaCombine` and `HandlePragmaIntermediateLiteral` are added that each accept a token of the respective pragma, decode it and pass their values on to the Sema library. To put the pieces together, these methods get called in the appropriate steps of the recursive descent parser upon occurrence of the corresponding pragma tokens:

- Within `ParseExternalDeclaration` in `Parser.cpp`.
- Within `ParseStatementOrDeclarationAfterAttributes` in `ParseStmt.cpp`.

Semantic Analysis The values associable to each of the settings controlled by pragmas are defined in enumerations in `Sema.h` as part of the `Sema` class. It also has a field for each of the settings containing the current value, to be used during parsing.

To act on the occurrence of these pragmas itself, the `Sema` class is furthermore extended with methods `ActOnPragmaPACOCCombine` and `ActOnPragmaPACOIntermediateLiteral`

3.2.1.4. Parsing of Approximate Arithmetic and Approximated Functions

Context In Clang, arithmetic expressions are separated into a left hand side and a right hand side. The left hand side contains the result and the right hand side contains an operator and an operand. If there is another operator following the operand which is typically the case in arithmetic expressions, the right hand side is also split into a new left hand side and a new right hand side. The new left hand side contains the result of the computation of the new right hand side which contains two operands and an operator. If there is another operator again, the splitting is executed recursively until there is no operator following.

Approximate Arithmetic An arithmetic computation is approximate depending on the approximate definitions of the result variable (`RelaxMask`) and the approximate definitions of the operands (`InjectMask`) which are combined to the neglect mask which is stored to the operator expression.

Parsing Arithmetic Approximations Arithmetic computations are part of assignments. Therefore the parsing of approximate arithmetic expressions also start in `ParseAssignmentExpression` in file `ParseExpr.cpp`. As long as the order of priority of the operators are equal, the function `ParseRHSOfBinaryExpression` which implements the splitting described above, will be executed. If this order is lower then conditional order of priority (which is the case by e.g. multiplication, computation with braces or else), the parser descends and calls `ParseAssignmentExpression` again to compute this values before continuing. This results into a recursive loop which makes it necessary to know when the parsing has started with the first call of the function `ParseAssignmentExpression`. This is done with the parser variable `relaxIsSaved`. At this starting point the `RelaxMask` from the result variable is also saved, because the `RelaxMask` from the result variable is needed later for the analysis of approximate computations. This `RelaxMask` is stored to every operand expression of this assignment. If the result variable is not defined as approx, the `RelaxMask` is set to a precise value. When `ParseAssignmentExpression` is called again by a descend call, the saved `RelaxMask` is also stored to the descend expressions. The mask is only released when the parsing of the assignment expression is done and the `relaxIsSaved` variable is set to `false`. To ensure the written masks are not deleted when the operand expressions are replaced by other expression objects, which happens at some locations in the code (e.g. by using braces in arithmetic computations), the function `copyPACOVValues` is used to copy the mask values from the old expression. While parsing, all operands and operators of an assignment are combined to a tree structure. Figure 4 shows two examples for this tree. You can see that all operands are always leafs, all operators are never leaves and the root is always the assignment

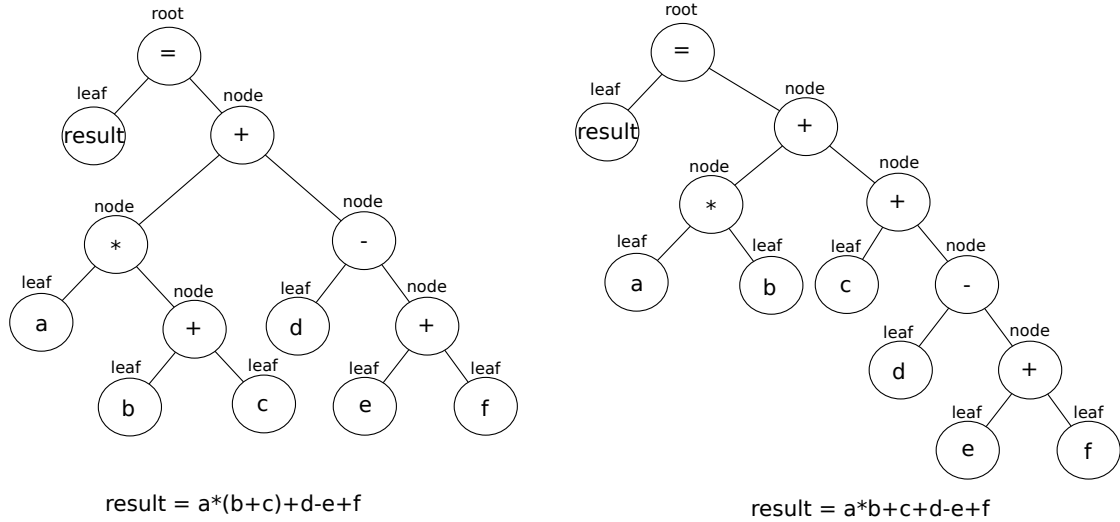


Figure 4: Example of PACO tree for approximate computation analysis

operator. This is needed to identify the source of the expression in the analysis of approximate computations. A leaf which represents an operator contains the mask value in the approx decorator, where an operator node, which contains an intermediate result, has no approx decorator. An operator node contains the mask value as a result in the `NeglectMask`, a variable which was added to the `Expr` class, implemented in `Expr.h`. The figure 4 also shows what happens if we leave out the braces. Without braces, the order of the parsing is different, which results into a different computation order.

Parsing of Call Expressions Similar to copy the PACO values in expressions, the approx decorator gets lost when a new declaration node is produced in the semantic analysis. To prevent this the approx decorator is copied from the old to the new declaration node.

3.2.1.5. Analysis of Approximate Computations

TODO (MUST): figure: abstract syntax tree with clang class names, annotated with changes made by us (ask basti for further details)

After parsing the approx decorators, the values of the approximate definitions are stored in the `ApproxDecoratorDecl`. For executing approximate computations in hardware, the approximate values must be stored in the `BinaryOperator` which contains the binary operation for arithmetic computations. Therefore the needed values for the computation of the approximation level are stored in the class `Expr` `Expr.h`. These values are filled with data in the function `SetMasks` which is implemented in the file `SemaExpr.cpp`. In this function, the masks are set depending on the pragma values as described in the design document A. First the `InjectMask` of both sides are computed depending on their individual neglect value. If the operands

are leafs in the PACO tree (constructed while parsing), the neglect values have to be read from the approx decorator of the operand. If the operator is no leaf, it can simply be loaded from the operand expression. When the left and right `InjectMask` is loaded, the pragma `intermediate_literal` defines how intermediate inputs are handled. When no valid inject values were loaded, the values are treated as completely precise. The relax value was set by the neglect value of the result variable, as mentioned before. The pragma `combine` defines the combination mode to decide which value (either the `InjectMask` or the `RelaxMask` is stored as result `NeglectMask` in the result expression.

3.2.1.6. Code Generation

Emit ALU Instructions If an expression of arithmetic calculations (here only addition, subtraction and multiplication) contains a `NeglectMask` and its approximation level is not full precise, then a new instructions will be emitted `CGExprScalar.cpp`.

Emit LUT Instructions There are two steps needed to emit a LUT instruction correctly. First, in `CodeGenFunction.cpp` the emitting of the function code will be prevented, if the function is marked as approximate function by the key-value `strategy`. Second, in `CGExpr.cpp` the `lute` and the `lute3` instructions are called. This instruction are printed instead of the original function within the intermediate code, which will go to LLVM for further processing. To store the uuid into these functions, the uuid is separated into four 32 bit values, because LLVM can not deal with 128 bit inputs in this version we use.

3.2.1.7. Test Code To test Clang, two test files has been added. The first file `ApproxGeneralAndALU.cpp` contains test code for general tests of the approx decorators and the approximate ALU. The second one `ApproxLUT.cpp` contains test code for the LUT related components. The test files can also be used as an example to show how approximate code is written correct.

3.2.2. LLVM Translation

This section covers the extension of the LLVM backend as an intermediate between the Clang C/C++ frontend and the machine code generation performed by the GNU binutils. All of our changes are done in the `riscv-tools-llvm` repository. For our extensions, we use LLVM intrinsics (see Appendix A.2) to represent the custom instructions added. This is possible as they are not affected by other instructions and do not affect them.

3.2.2.1. Adding Intrinsics, Translating to Instructions All modifications to LLVM were handled using intrinsics which are a lightweight way to pass new instructions through LLVM without much further processing.

Intrinsics are statements in the LLVM intermediate representation and get translated to assembly instructions.

Definition To add an intrinsic for use with RISC-V, it must first be defined in the file `IntrinsicsRISCV.td` following the format laid out by the intrinsics already defined there.

The intrinsic class accepts three arguments: The result value types, the argument value types and additional options. For available data types refer to Appendix A.2.

The final argument specifies additional options as a list of symbols, most notably `IntrNoMem`, specifying that the intrinsic will not interact with memory but rather work on the processor's register file only.

Translation: Instruction Format To output instructions in a new format as required for our approximate instructions, these formats must be specified in `RISCVInstrFormats.td`.

The best practice is to simply copy an instruction format type and adjust it to suit the needs of the new format. For further information on specifying instruction formats refer to Appendix A.2.

Translation: Instruction Instructions themselves are defined in `RISCVInstrFormats.td`.

They consist of a single declaration using an instruction format as described above.

The arguments in instantiating the instruction format can be literals, such as for mnemonics and instruction codes, or references to intrinsics and internal registers.

Further information can be found in Appendix A.2.

Translation: Instruction Info *The instruction info, implemented in `RISCVInstrInfo.td`, defines the opcode and the LLVM internal types of the input and output for the intrinsic instructions, which are implemented in `RISCVInstrFormats.td`. The LLVM internal types are either register types `GR64` and `GR32` for 64 bit and 32 bit input/output or an immediate value with a specific bit width.*

Translation: Operands *In the file `RISCVOperands.td` are the immediate operands defined, which can be used in the file `RISCVInstrInfo.td` and in the file `RISCVInstrFormats.td`. To define a new operand just copy any existing one and edit the name. If the bit width has to be set to a value which does not exist so far, a new immediate value has to be added in the same `RISCVOperands.td` and a new function for checking the bit width has to be implemented in `RISCVAsmParser.cpp`. This was the case for example for the variable `imm32zxtutroc` which has a bit width of just one bit, so a new definition for `U1Imm` was needed.*

Adding Test Cases Test cases are added to the directory `RISCV`. A test can simply be added by copying an already implemented one and just change the

instruction declaration and the intrinsic call to the needed input which was defined in the corresponding `.td` files. For complex tests, you can also use Clang with the compiler flags `-cc1 -emit-llvm`, if the new intrinsics were added in the correct parts in Clang. For further information on writing tests refer to Appendix A.2.

3.2.2.2. Selection DAG, Passes and UUID-Translation

Context The Selection DAG is an abstraction of code implementations which helps to identify the optimal implementation of an LLVM-IR code in assembler code. All nodes of the Selection DAG, which are instances of the class `SDNode`, perform all kinds of tests. These tests called Passes are responsible to make sure every input is correct. In some cases they also perform a correction (for example when a pseudo instruction is split into two normal instructions).

Since the UUID, which is used to identify the different functions which are approximated in the high level code, is a 128 bit value, it is impossible to store the whole value into an input for the new LUT instructions. Therefore, the UUIDs of all functions are translated into sequential indices. This was the reason for creating a new Pass.

TODO (MUST): figure: LLVM translation flow: Selection DAG and passes

Initialize the new Pass PACO To add a new Pass to LLVM, several additions were made: First of all a new header file is needed to define the Pass functions. This is done in `Paco.h`. Next steps consist of the registration of a new Pass module, which is done by defining the function which registers the new module to the header file `InitializePasses.h` and the implementation in the corresponding file `paco.cpp`. Here the registration for each function, in our case only `initializeLutTranslatePass`, is called, which is implemented in `LutTranslate.cpp`. After creating the function `initializePACO`. It can be called in `llc.cpp`. In `LutTranslate.cpp` a new module was implemented and also the function `runOnModule` which calls the translation pass for the LUT uuid.

Implementation of LUT Translation Pass The basic idea in the LUT translation Pass, which is implemented in `LutTranslate.cpp`, is to map the UUID to up counting sequential indices. Therefore all UUIDs are saved in a map and when they first occur, mapped to a new index. To identify the UUID, it is stored into a 32 bit array, because the UUID is transmitted from Clang to LLVM via four 32 bit values. This four values are compared to earlier existing UUIDs if any exist. When the uuid exist, it will be updated with the saved index, otherwise a new index will be created and the UUID is updated with it. The update stores the new value to the fourth input. This happens, because only one value can be used as input for the hardware. To prevent errors, the other values are stored with zeros.

3.2.3. Machine Code Generation

Machine code generation is the final stage in the compilation process, translating assembly code into binary code.

This is handled by the GNU binutils assembler which spans two repositories: `riscv-tools-opcodes` for the description of the new opcodes and `riscv-tools-gnu-toolchain` for the actual translation.

3.2.3.1. Modifying the Assembler

Defining the Opcodes Opcodes themselves are defined in the file `opcodes` as a single opcode per line consisting of the opcode mnemonic, the argument components and a number of fixed bits in the opcode.

For changes in the opcode definition file to take effect, the opcodes must be re-built by:

```
cd riscv-tools-src/riscv-opcodes
make
```

This step is always required, because assembly toolchain is built separately from the rest of the GNU toolchain.

Adding Opcodes to the Translator For new opcodes to be recognized by the parser, they must be registered in `riscv-opc.c` as part of the constant structure `riscv_builtin_opcodes`. The individual fields specify the mnemonic as defined in the opcodes (see above), the instruction set in terms of RISC-V instruction sets

CITATION NEEDED

the set of arguments, parsing function pointers and a flag mask.

The function pointers are auto-generated parsing instructions for detecting the respective instruction (`MATCH_` and `MASK_`) as well as a custom filter used to select instructions based on other factors than the static part of the opcode. This can simply be specified by `match_opcode` which does no special filtering.

The final flag mask accepts a single bit, 1 indicating the mnemonic is an alias for a different instruction.

Adding an Instruction Field If a new opcode requires a special field that has not been defined before, it must be defined as such both in the opcodes definition and the translator.

Note that an instruction field has two identifiers: One used in the opcode definitions and the other used by the translator.

Adding it to the opcodes definition is done in the python script `parse-opcodes` in the dictionary `arglut`, mapping field names to a tuple (*start*, *end*) defining the bit range comprising that field.

Adding it to the translator is done by adjusting the file `tc-riscv.c`. The new instruction field must be accepted as valid in the case structures in `validate_riscv_insn` and processed in `riscv_ip`. The identifier used here is a single character to be used in the opcode definition (see 3.2.3.1).

Finally the instruction field must be registered with the disassembler as well. This is done by adding a case into the `print_insn_args` method in `riscv-dis.c`.

3.2.4. Lookup Table Compilation

TODO (MUST): figure: internal tool flow, interoperation between compiler parts

The Lookup table compiler (LTC) is a command-line tool which translates a target function description into a lookup table description and configuration bitstream ready to use with our LUT hardware core.

During the compilation two principal actions are taken: *Segmentation* and *Approximation*.

Segmentation is the process of deriving the domain of the target function's input into intervals which form the individual segments of the LUT.

This is achieved by first determining the width of this domain and thereby selecting the exponent of the segment selection bits of input words.

After that, a user-selected segmentation strategy is executed to select intervals from the previously computed domain space.

Approximation operates on each segment individually and computes the affine linear function computing values in that segment by fixing the values on the leftmost and rightmost boundaries.

3.2.4.1. Usage The LTC handles three different file formats: *input files*, *intermediate files* and *output files*. It can read input or intermediate files and compile into output files or intermediate files.

To specify characteristics of the target LUT hardware cores, an additional *architecture file* can be provided.

The minimum input to the LTC is a single input file providing values for the keys `name`, `bounds`, `segments` and `approximation`, specifying the identifier used for the data sections of the LUT, the parts of the input domain for which approximation is required (all other inputs are interpreted as don't cares), as well as the segmentation and approximation strategy to be used, respectively.

In most cases an additional architecture file is used to tell the LTC about the layout of the LUT core to be instantiated. An example of how to write input files can be found in an application template in `lut-application`.

3.2.4.2. File Formats

Input Files An input file is an excerpt from the surrounding program in which the LUT resides. It contains a list of key-value pairs defining the compilation settings and properties of the LUT, the signature of its target function as well as C/C++ source code defining the target function. Input files are generated automatically by the PACO compiler during the compilation process of the surrounding program.

Intermediate Files Intermediate files contain a number of segments and configuration data for the LUT. It is generated by the LTC and can be read again, forming an entry point for external programs or manual intervention into the LUT generation process.

Output Files Output files contain only the bitstream used for configuring a LUT hardware core. This can be outputted either as C code containing a single constant definition of the bitstream data, or as an ELF which was generated from such code.

Architecture Files Architecture files are a list of key-values syntactically identical to the first section of input files, overriding default architecture-relevant settings (see section ??).

Weight Files Weight files are optional files used for assigning levels of importance to input values, thus prioritizing those with a higher number in approximation and segmentation.

3.2.4.3. Command-line Options With each invocation of the LTC, a single compilation process takes place. Therein the input, output and parameters must be specified along with an input file name:

- By default, the input is expected to be in input format. This can be switched to intermediate using the command-line switch `-c`.
- The output is generated as ELF by default. To generate the corresponding C++ code instead, add `-C` to the command-line. Specifying `-i` will generate intermediate code.
- To specify a weights file to be used for segmentation and approximation, use `-w` followed by the path to the weight file.
- An architecture file can be set with `--arch` followed by the architecture file name.

Further command-line options can be found by invoking the LTC with `-h`. These are not important for normal operation.

3.2.4.4. Segmentation Strategies (Primary) Primary segmentation strategies are used on an empty set of segments for a LUT and they generate a number of segments that may be subdivided further with a secondary segmentation strategy.

Uniform Uniform segmentation creates a set of equally-sized segments covering all the input domain as selected by the user. Therefore it first determines the minimum segment width feasible so that the maximum allotted number of segments is not exceeded this way.

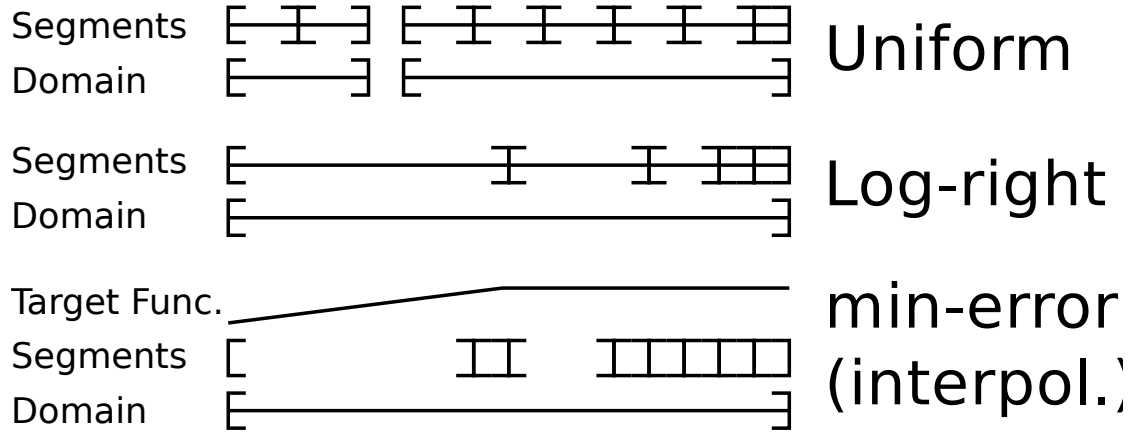


Figure 5: Examples of segmentation strategies. Each group shows the designated domain to be segmented and the segment intervals placed. Note that for error minimization, an approximation strategy must also be specified.

Logarithmic Logarithmic segmentation creates a set of segments incrementally increasing in width by a factor of two, covering the entire width of the input domain, disregarding intervals that are not part of the input domain requested by the user. There are two modes of operation for this strategy: Remainder-based and Binary representation-based. The former operates by starting with an interval width of 1 and adding intervals of increasing width until that width would exceed the remaining width to be covered. This remainder is then added. Binary representation-based segmentation on the other hand creates one segment for each bit in the binary representation of the domain width. Independent of the method used, for left-bound (right-bound) logarithmic segmentation, the generated segments are then sorted in ascending (descending) order.

Error Minimization This strategy starts by generating minimum-width segments for all the parts of the input domain that need to be covered. After that it successively combines two segments until the maximum number of segments was attained. Selection of the pair of segments to be joined is performed greedy by choosing the first operation exhibiting the minimum mean square error on the input domain (min-error strategy). To instead use the maximum gain in precision (maximum loss of error), use the min-error-gain strategy.

3.2.4.5. Segmentation Strategies (Secondary) Secondary segmentation is performed on each segment generated by the primary segmentation strategy, aiming to further subdivide segments, utilizing segments left over by the primary segmentation strategy.

The strategy is thus invoked once for each segment, distributing the number of allotted segments per subdivision step equally.

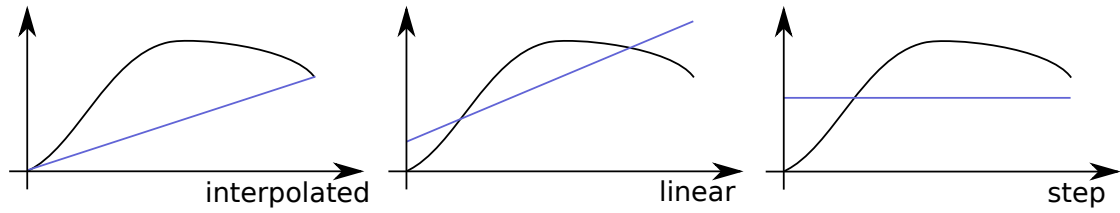


Figure 6: Examples of approximation strategies. The target function is shown in black, being approximated by an affine linear function (blue).

Uniform Uniform secondary segmentation operates exactly as it does in its primary form.

Logarithmic Logarithmic secondary segmentation operates exactly as it does in its primary form, using the width of the segment to be subdivided instead of the total domain width.

Error Minimization Error minimizing secondary segmentation operates on the entirety of segments instead of just one, splitting one segment at a time into two new segments until the maximum number of segments was reached. The selection process is identical to the one used in primary segmentation.

3.2.4.6. Approximation Strategies In the following all the available approximation strategies are listed.

Interpolated The simplest is the interpolation strategy. Here the target function is evaluated for the lower and upper segment boundaries and these values are used to fix the affine linear function of the segment, simply interpolating between those two values. While this may generate higher mean error rates within each segment, it is the only strategy that guarantees the absence of discontinuities between two adjacent segments.

Linear Linear approximation samples the target function at each possible input value and fits a linear function to that data set by minimizing the mean square deviation.

Step Similar to linear approximation, samples the target function at each possible input value and fits a constant function to that data set by minimizing the mean square deviation.

3.2.4.7. Implementation Overview The remainder of this section is to be regarded as an overview and explanation of the design and construction of the Lookup table compiler

to be read in conjunction with source code documentation provided in the code files itself.

TODO (MUST): refer to doxygen

An overview of the important files in the LTC source code can be found in figure 7. Building is performed through the sole makefile located in the source code directory `riscv-lut-compiler`, which first builds a dependency file `make.incl` by traversing the directory tree and listing all source code (`.c`, `.cc`, `.cpp`) and lexer (`.l`) files and generating makefile rules accordingly.

The LTC main routine is located in `riscv-lut-compiler.cpp`. The compilation process therein utilizes the two principal classes of the software: `WeightsTable` and `LookupTable`. The former is used to handle the weights functions to be specified by the user while the `LookupTable` class handles input/output of files, translation from segments to bitstreams and management of segments.

Segmentation and approximation strategies are added as a modular system by declaring them in `strategies_decl.h` and implementing them in a source file in the `strategies` directory.

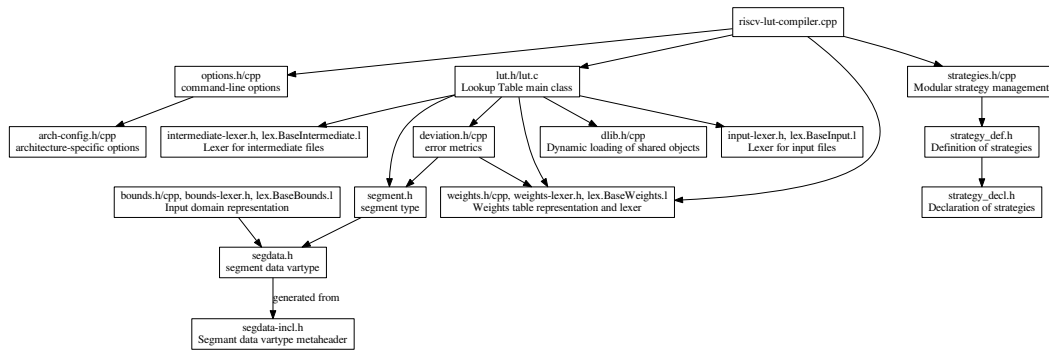


Figure 7: Overview of the important source code files of the LUT Compiler Tool.

Third-party Software

- **Alpha Framework:** Parts of the LTC depend on components from a code library called *alpha*. This library is a repository of code gathered throughout several years and was not written as part of this project group.
- **Flex Scanner Generator:** To read the different text-based formats used by the tool, the *flex* scanner generator was employed. It inputs lexer files (`.l`) and generates C/C++ source code used for scanning raw character streams. To improve usability, these scanners were created with source positions in mind, keeping a record of the location in the raw character stream to generate appropriate warnings and error messages where needed.

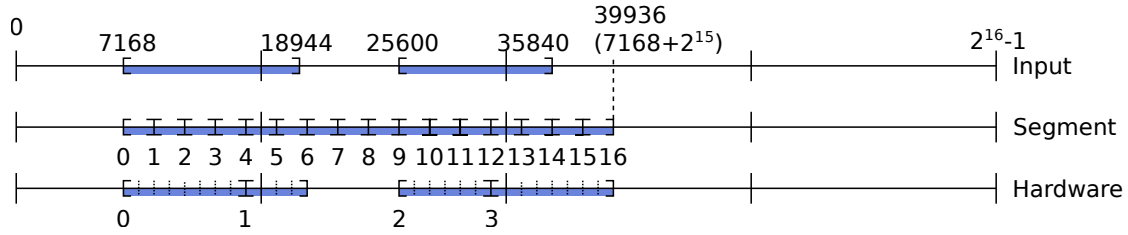


Figure 8: Coordinate systems used in the LUT Compiler Tool for an exemplary translation of the highlighted input domain. In this example the segment space is 2^{15} value wide with an offset of 7168. With 2 selector bits the segment space is subdivided into 16 principal segments. They are reduced to 4 segments (2 segment bits). Dotted lines in the hardware space indicate individual evaluable points which result from a single interpolation bit.

Coordinate Systems During the generation of segments from a target function, several different coordinate systems are used to represent positions in the target function’s domain.

- **Input Space:** This is the most basic coordinate system, being simply the domain of the target function itself.
- **Segment Space:** In segment space not the entire domain is represented but rather the part of it that is being mapped onto the LUT. Segment space is derived in a number of *principal segments*, that are segments of minimum expressable width (see `selectorBits` in section 3.2.4.7). A point in segment space is identified by a tuple (i, o) with $i \in \mathcal{N}, o \in [0, 1]$. i represents the index of the principal segment and o is an offset into that segment.
- **Hardware Space:** Hardware space is very similar to segment space, however instead of having an index into a principal segment and a real-valued offset it uses a segment (as generated in a segmentation strategy) and an integer (representing the interpolation bits, see section `refsec:luc-arch-config`). Therefore, hardware space can express exactly those points in the input space that can be evaluated in the LUT hardware core.

Architecture Configuration

TODO (MUST): add reference to bitstream layout figure

Architecture-specific values are stored in a structure defined in `arch-config.h`. The most important values are as follows:

- **selectorBits** This is the number of bits in an input word that are fed into the address translation PLA.
- **segmentBits** This represents the number of segments that can be stored in the LUT and thereby the number of outputs of the translation PLA.

- **interpolationBits** Number of additional bits fed into the interpolation multiply/add unit. The total number of bits is the sum of **selectorBits** and **interpolationBits**.
- **plaInterconnects**: Number of interconnection rows between the and and or planes of the translation PLA.
- **base_bits**: Number of bits for the base value of each segment. A segment is an affine linear function attaining the base value at the lower bound and increasing by a factor of **incline**.
- **incline_bits**: Number of bits for the incline value of each segment. A segment is an affine linear function attaining the base value at the lower bound and increasing by a factor of **incline**.
- **delay_addressTranslator**, **delay_controller**, **delay_inputDecoder**, **delay_interpolator**: These settings specify how many clock cycles of delay are introduced in each of the configurable pipeline stages of the LUT hardware. In most cases all of these are 0 (default). For more information refer to section 4.2.

When specifying key-values in an architecture file, the name of the key-value is identical to the field in the `arch_config_t` structure.

3.2.5. Error Handling and Testing

3.2.5.1. Exception classes Defined in `error.h` there are a number of exception classes:

- **FileIOException**: A file could not be read or written.
- **SyntaxError**: An input file is invalid in terms of its syntax. This is only thrown when reading an input file or buffer.
- **RuntimeError**: Some user-specified semantics do not check out. This can never be a programming error.
- **CommandLineError**: An invalid command-line was specified when running the program.
- **HWResourceExceededError**: If the hardware resources as specified by the architecture configuration are exceeded during compilation, an instance of this exception is thrown.
This can occur if segmentation yields a sequence of segment boundaries that is too complex to be mapped with the number of PLA interconnects available in the LUT.

In addition to these, asserts are used throughout the program to emit a critical failure when invariants are unmet that are a result of program design rather than user input.

3.2.5.2. Logging System The LTC uses a logging system from the alpha framework which distinguishes between *error*, *warning*, *info* and *debug* messages, printing them onto standard error / output using ANSI color codes.

3.2.5.3. Unit Tests Utilizing a minimalistic unit test feature from the alpha framework, critical components of the program are tested in-code. These tests are executed before the main routine runs and thus should be excluded from a production build of the LTC. This can be done by specifying

```
-DALPHA_UNITTESTS=0
```

when running `make`.

3.2.5.4. System Tests Testing the overall behavior of the LTC, a number of system tests were written, located in `tests`. To execute them, simply run:

```
make tests
```

This will compile the LTC itself, if not done already, and run all of the tests in arbitrary sequence, aborting if one of them fails.

3.3. Limitations and improvement-worthy parts

3.3.0.1. Floating-point Support In accordance with overall project group progress, support for floating-point approximation was not implemented.

3.3.0.2. Target Function Evaluation Currently the target function is evaluated by running an external compiler program and loading the result as a shared object into the LTC's memory. This process is somewhat unstable and could be better solved by using a JIT compiler such as Clang.

3.3.0.3. Default Strategies Currently the LTC does not have any default values for strategies.

4. Approximation Hardware

4.1. Environment in Rocket chip

We extended the Rocket Chip, which is an implementation of the RISC-V architecture, written in Chisel (cf. figure 9).

Our extensions consist of an approximate ALU and a Lookup-Table (LUT), both instantiated alongside the precise ALU of the Rocket Chip. The LUT itself is a hardware component described in VHDL, thus it is instantiated in the Rocket Chip pipeline via a black box interface.

TODO (MUST): figure/table: directory structure of rocket-chip, describing what directories do, how important they are and where programmers need to look.

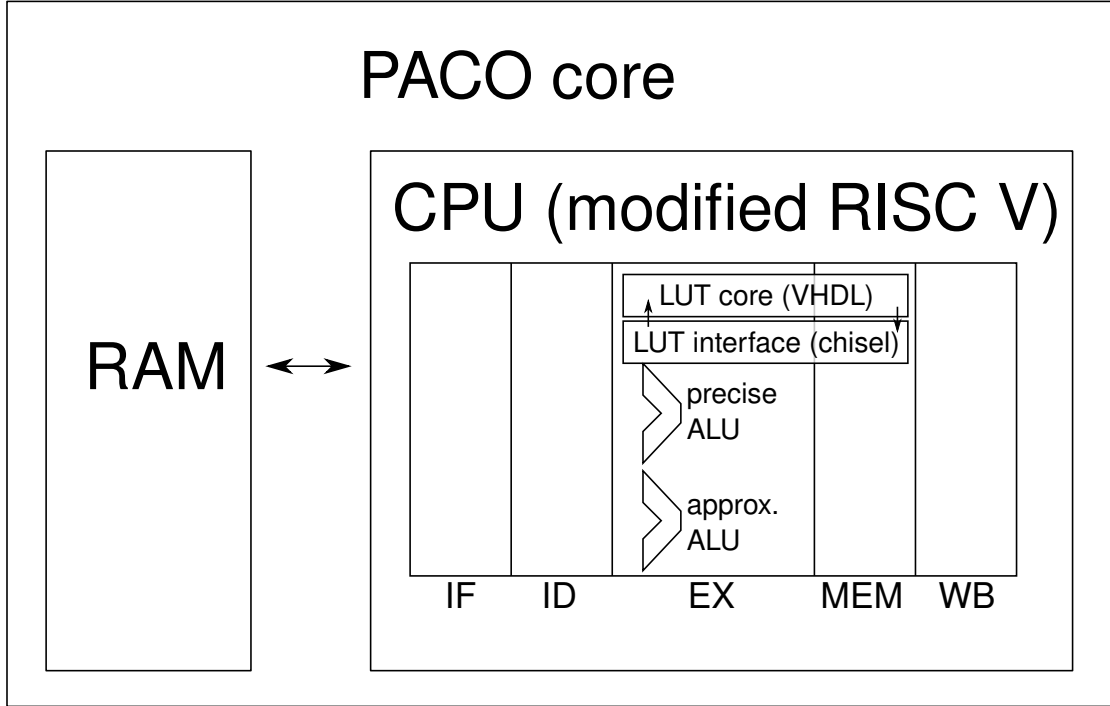


Figure 9: Overview of the PACO CPU core. The Rocket Chip implementation of the RISC-V CPU core is extended with an approximate ALU and a Lookup-Table (LUT). The LUT core itself is written in VHDL while all of the remainder is described in Chisel.

4.1.1. Chisel

Chisel is a hardware construction language developed by UC Berkeley for embedded programming using the Scala programming language. Scala or Scalable Language is a statically typed language that provides functional and object oriented methods of writing code. Scala runs on the Java Virtual Machine (JVM), making it platform independent.

Chisel is created by extending Scala in order to describe hardware; thus enabling programmers unfamiliar with the embedded domain to program hardware fast and easy. Chisel is open source and has a growing community of developers contributing to it.

The appendices A.4 and A.5 provide links to get started with Chisel and Scala respectively.

4.1.2. Decoder

The purpose of the decoder is to interpret the fetched instruction and to send control signals to the CPU in order to execute it. The decode stage in the rocket core is responsible for obtaining the operands, either from the bypass sources or from the

register file, for it to be executed by the ALU/FPU. Additionally, the decoder looks up a configurable table, wherein each row represents the signals required to control the functional units corresponding to the interpreted opcode.

The `rocket.scala` file describes 5 out of 6 stages of the rocket core's in-order pipeline in the **rocket** class. The missing stage - the Instruction Fetch stage, is instead detailed in `frontend.scala`. The configurable table of control signals looked up by the decode stage is listed in the `idecode.scala` file.

In order to implement the approximate ALU part of the PACO Core, the decoder had to be extended to include **ADDAPPROX** and **MULAPPROX** instructions. The following steps were needed to modify the decoder:

- A control signal **precise** was added to every instruction in the **XDecode** object in the `idecode.scala` file. This control signal determines if an instruction executed precisely (represented by a 'Y') or approximately (an 'N') and activated the approximation logic in the core explained in section 4.3.1.
- A boolean variable `val precise = Bool()` was declared in the `IntCtrlSigs` class, referring to non-floating point instructions.
- The same variable was appended to the array **sigs** in the function **decode**.
- The default macro value of the variable was added to the list - **decode_default**.

This results in the mapping of the variables with its corresponding values during the decode stage in the `decode.scala`.

Other than the control signals, the decoder was modified similar to the Approximate ALU in order to accommodate the LUT instructions. The control signals added for the LUT instructions are `alu_lut_sel`, `lut_ex`, `lut_wr`, `lutl` and `luts`, which can be better understood on reading the section 4.2.2.

These suggestions can ,therefore, be taken into account while modifying the core in the future to add further approximate instructions.

TODO (MUST): figure: decoder control signals

4.2. LUT

4.2.1. Overview

As specified in design document Look up table hardware is intended to perform complex computation of approximate functions, at the end aiming for considerable speeds in computing.

The LUT hardware unit is planned to be integrated inside the pipeline as keeping it as coprocessor is not feasible and may cause delays. For the LUT to execute inside pipelines two major LUT instructions are designed which are LUTL for loading the configuration data and LUTE for doing look up for values in the LUTE. The detailed description of the instructions are provided in the design document. Since the LUT

hardware is a part of execution phase inside the pipeline, the LUTL and LUTE instructions will execute the LUT hardware residing inside pipeline, if the execution of LUT takes more than one cycle, pipeline need to be stalled. For now it is been tested that the LUT will not take cycle delays when executed inside FPGA, hence the stalling is postponed, unless the need arises.

Problem: Is it still the case that the LUT is tested? Aren't we in a state to give more information about what would be the best implemented size for the LUT and how much cycles to take are reasonable?

Architecture *The LUT hardware is implemented as a blackbox Chisel, which gets instantiated inside the generated verilog source with input and output port values. The LUT hardware core is implemented and described in VHDL which is integrated with Chisel blackbox receiving the inputs based on instruction from rocket and sending the output to the rocket which at the end write back to the memory at the write back stage. Within the hardware team, the responsibilities are shared as follows:*

Problem: *Why are responsibilities still in here? It seems to be part of the intermediate report.*

- *The VHDL implmentation of the LUT core is assigned to one person.*
- *Chisel blackboxes and addition of LUT instructions is being assigned to two persons*
- *Downloading the rocket and LIT core inside FPGA is being done by 1 person.*

Problem: Why are this TODOs commented out? If they are done, please delete them.

4.2.2. Chisel Interface

Modifications The Chisel interface includes the blackbox units and the control signals from the decoder in the rocket-rocket repository.
The modified files are rocket.scala and idecode.scala.

idecode.scala Following control signals are added in the decoder.

- **alu_lut_sel:** This control signal signify whether an instruction is LUT instruction or non LUT instruction. When **alu_lut_sel** is true it means it is one of the LUT instruction.
- **lutl:** For LUTL instruction, LUTL (lut loads) is set to true else for other instructions it is always false.
- **lut_ex:** Set to true iff the current instruction shall start a computation on a LUT. This is used to distinguish between LUTE and LUTW instructions.
- **lut_wr:** Set to true iff the current instruction shall modify input registers of the LUT. As a LUT core can have more input registers than the CPU can transfer in a single clock cycle, a buffer is used for all of the LUT core inputs. If the **lut_wr** signal is false, no buffers are written to.

- **luts**: luts is set high for LUTS instruction.

All the LUT instructions are distinguished from each other by using the above control signals

rocket.scala The interface for LUT hardware core is implemented in rocket.scala in the form of blackbox. The blackbox unit is named as **lut_core**. It has following input and output ports and communication of rocket and LUT hardware core occurs via this blackbox and its io ports.

- **data_i, data2_i**: Data inputs to the LUT hardware core. Up to two LUT core inputs are updated in a single clock cycle. The selection of which input registers to update is done via the **charm_i**, **strange_i** and **lutsel_i** inputs.
- **charm_i** Single bit input, selects whether to write to two input registers (true) or a single one (false).
- **strange_i** Two bits input. Select which LUT input register(s) the CPU register(s) are written to. For further information refer to the design document.
- **data_we_i**: Set to true to enable writing to the LUT core's internal input registers.
- **id_rst_i**: Reset signal denoting whether to reset the configuration data or not, the field is present in LUTL instruction. This is 1 bit wide input signal.
- **id_cfg_i**: Configuration signal denoting whether to load the configuration data or not, this field is present in LUTE instruction.
- **id_stat_i**: Status signal for retrieving the status of the LUT. This field is present in the LUTS instruction. This is 1 bit wide input signal.
- **id_exe_i**: Execute signal for computation in the LUT hardware. When this signal is high, LUTE instruction is executed allowing computation and look up on the LUT hardware. This is 1 bit wide input signal.
- **data_o**: Output from the LUT hardware core which is then written to the destination register rd. This is as wide as length of the register.
- **data_valid_o**: This is one bit output signal, outputting the validity of the data.
- **error_o**: This is one bit wide output signal. Used in case of any error sequence.
- **status_o**: This signal outputs the status from the LUT hardware and is as wide as length of the register.

Problem: If TODOs are done, please delete them, not just commenting them out.

4.2.3. Hardware Core

The LUT Hardware core is a hardware component described in VHDL that interfaces with the black box as described in the previous section. It can be found in the repository `rocket-lut` together with its testing environment.

4.2.3.1. Component Overview

TODO (MUST): update figure contents, split apart for better readability

An overview of the LUT hardware core can be seen in figure ?? which is described below.

Inputs and Outputs The LUT core inputs at most a single instruction to be executed per clock cycle (reset, status retrieval, configuration and execution/writing) which is one-hot encoded via the input signals `id_rst_i`, `id_stat_i`, `id_cfg_i` and `id_exe_i`, respectively.

As depicted in figure 11, up to two registers of arguments can be supplied for the instruction, given by `data_i` and `data2_i`, corresponding to the first (r_{s1}) and second (r_{s2}) registers.

Most commands take zero or one inputs, however the execute/write instruction handles one or two inputs. Internally the LUT core operates on three input registers, of which one or two are updated at a time through the execute/write instruction. To select which register(s) are written to, the signals `charm_i` and `strange_i` are used to select one or two register input (`charm_i`) and which one/two register to write to (`strange_i`).

States and Interaction The LUT core resides in one of the following states: *RAM configuration*, *chain configuration*, *ready*, initially occupying the RAM configuration state.

The RAM and chain configuration states each accept a single word delivered by the configure instruction and move to an error state if execution was Requested.

During the RAM configuration state the Segment data looked up by the hardware core is written sequentially (via `ramcfg_o` as seen in figure ??) and after all cells were written, the chain configuration state is assumed. This state causes the words delivered by the configuration instruction to be forwarded via `cfg_o`.

After a sufficient number of words were written in the chain config state, the ready state is assumed in which only execution is permitted and an error state is assumed if an attempt at writing further configuration data is made.

At any point in time a reset or status instruction can be requested. The former re-initializes the core, resetting the state to RAM configuration and the RAM cell counter to 0, effectively unsetting LUT configuration.

The status instruction delivers a single word of information via the `status_o` output port containing information about the state of the controller: Error flags and the total number of configuration words utilized since the last reset instruction.

Once this status word contains an empty error mask and the number of utilized registers equals the number of required registers, execute instruction may be requested.

Internal Components and Data Path *The core is subdivided into five components: The LUT controller handles interpretation of requested instructions and the state machine as well as the first step in the data path(processor data path; if you mean the new implementation of the LUT datapath, please use a different word express it somehow).*

Problem: *Is the LUT controller part of the decode phase, so isn't it part of the decoder? The first step of the data path is always instruction fetch, this confuses a little bit. Please make it more precise in which part of the pipeline the five LUT components are implemented (this also applies for the following components).*

The data path continues through the input processor selecting desired bits from an execution instruction's input; The address translator performs an $n : m$ translation of some of the picked bits yielding the LUT segment address; In the next stage this address is used by the block RAM controller to look up the base and incline of the addressed segment. Finally the interpolator performs a multiply-and-add stage yielding the final result.

Problem: *Maybe an enumeration would be better, with further explanation, like: The components are: 1, 2, ... and they do this and this*

Delay *At most steps of the data path as described above an arbitrary number of delay cycles to be inserted if the path delay exceeds the clock period of the processor.*

Problem: *Same here, which data path?*

Problem: *So the LUT stalls dynamic depending on the needed time, do i understand it correct? If not, please clarify it.*

A single mandatory delay step is inserted during block RAM lookup which thus is fixed to a single clock cycle delay. The other four components can operate with zero or more delay steps.

Problem: *Which other four components? Do you mean the controller watches the delay for the other components? Then please name the controller as the one component here which is not included in the phrase "other four".*

4.2.3.2. Architecture Parameters

CLARIFY: The individual components are described in detail in the respective code files. Discussing this here would introduce duplication of text.

Solution: A splitting in general overview and implementation would be the best. The general overview was already done in the data path paragraph (if some information is added). If you do that, you can delete the empty paragraphs here. But please make sure to name the components the same way. E.g. Input decoder != Input processor

4.2.3.3. LUT controller

4.2.3.4. Input decoder

4.2.3.5. Address translator

4.2.3.6. Lookup Table

4.2.3.7. Interpolator

4.2.3.8. Testing Testing of the LUT Hardware core is divided into three paradigms: VHDL test benches simulate individual components and offer a means of manually debugging and testing them. Hardware tests are synthesizable top-level modules interfacing a single component via a UART transceiver and work in conjunction with a python script that performs automatic testing. Finally, integration of the entire LUT Hardware core is tested with a system test.

Test Benches *VHDL test benches exist for each of the five components of the LUT hardware core: Input processor, address translation, RAM control, interpolation and the LUT controller.*

Problem: *Please use the same order as above and the same names*
Each of these test benches is located in a file prefixed with tb_ . While these test benches can be used to debug and test components manually, they can not provide a guarantee that the synthesized circuit is executed correctly when instantiated on an FPGA.

Hardware Tests Hardware tests start off where test VHDL benches end: Instantiating a single component on an FPGA and interfacing it via UART, hardware tests deliver a definitive answer to whether or not a circuit is operational. Analogously to test benches, hardware tests are prefixed with ht_. Each of the tests consist of a top-level VHDL module (extension .vhd), a python class interfacing with the top-level module (appropriate .py file in the htlib directory) and a test script (extension .py).

To execute a hardware test, simply implement the corresponding top-level module, download the bitstream onto an FPGA and execute the respective script, adjusting the UART port if necessary.

To allow for automatic testing using hardware tests, a simulated implementation of each component as well as the entire hardware unit was performed in the python language. These scripts provide reference input-output pairs for each of the hardware tests, which are then checked against computations performed on the respective hardware component.

Hardware tests are available for the address translator, input processor, interpolator and the LUT controller as well as the entire LUT Hardware core. Due to the trivial nature of the RAM controller implementation, no hardware test was written for it.

Problem: *Think of same names and order as above.*

System Test Complementing (and building on) the hardware tests, a single system test was written which tests the correct configuration and computation of Lookup tables by interfacing with a LUT core embedded in the RISC-V processor pipeline. This is achieved through a RISC-V program located in system-test that utilizes input-output pairs provided by the python simulation and compares them with results of LUT instructions executed on the RISC-V instantiation.

4.2.3.9. Configuration Bitstream The configuration bitstream for each LUT consists of three blocks: RAM data, Input decoder crosspoints (connection plane) and PLA crosspoints (AND and OR planes). For a graphical overview, please refer to figure 12. Each block is made of a sequence of *bitvectors* that may or may not exceed the length of a word of the RISC-V processor.

Each bitvector is expressed with as many words as they are required to cover all bits of the bitvector, remaining bits are left as padding.

Example: Let the RISC-V word size be 64 bit and have the PLA use 80 interconnects between the AND and OR planes. Then each column in the OR plane is a 80 bit vector. Thus each of these is expressed as two configuration words with bits 80 through 127 being don't cares.

Ordering and Alignment Words within a bitvector are ordered *lowest word first* while the bytes within each individual word are ordered *in the processor's endianness*. As the RAM block is written front-to-back and the remainder is part of a daisy chain, the words making up the Input decoder and PLA crosspoints must be written in *reverse* order, thus if the blocks have n RAM words, m input decoder words and o PLA words, the bitstream is stored as:

$RAM_1, RAM_2, \dots, RAM_n, PLA_o, PLA_{o-1}, \dots, PLA_1, IDEC_m, IDEC_{m-1}, \dots, IDEC_1.$

RAM Bitvectors The RAM consists of $2^{\text{SEGMENT_BITS}}$ bitvectors (one for each addressed segment) of $\text{BASE_BITS} + \text{INCLINE_BITS}$ bits. The lowest INCLINE_BITS bits are the *two's complement* representation of the segment's incline and the remaining BASE_BITS bits are the *two's complement* representation of the segment's base value.

Input Crosspoints The input decoder is configured with one bitvector for each selector and interpolation bit, each being three times as wide as words in the processor (192 bits). Each bit in the bitvector represents one crosspoint in the input selection matrix. Thus it should be a one-hot encoding of the bit of the input words to be used as the respective bit. The lowest bit of the bitvector represents the lowest bit of the first input word (rs_1), incrementing in the obvious manner.

The first $\text{INTERPOLATION_BITS}$ bitvectors define the crosspoints for the interpolator and the remaining SELECTOR_BITS bitvectors handle the selector bits, in both instances starting with the lowest significant bit.

PLA Crosspoints The input I to the PLA is the selector bits as relayed by the input decoder expanded by its complement: $I = s_1 s_2 \cdots s_n \cdot \overline{s_1} \overline{s_2} \cdots \overline{s_n}$, s_1 being the lowest significant bit of the selector.

The configuration block is split in two segments: AND and OR plane bitvectors. Each bitvector in the AND plane holds the bits for the crosspoints of one interconnect between the AND and OR planes: It is AND-combined with the input I and the AND-reduction of this combination is fed into the OR plane, thus it has $2 \cdot \text{SEGMENT_BITS}$ bits.

The lowest significant bit of each bitvector is the one AND combined with the lowest significant bit of the input I , and so on.

The OR plane bitvectors each combine all interconnects in a similar fashion as the AND plane combines all input bits: The interconnect bits are AND-combined with the bitvector and OR-reduced, making up the output bit, thus each bitvector has PLA_INTERCONNECTS bits.

Formally, the following formula express the bits and bitvectors used in the PLA:

$$\begin{aligned}
n &:= \text{SELECTOR_BITS} \\
m &:= \text{PLA_INTERCONNECTS} \\
o &:= \text{SEGMENT_BITS} \\
s &:= \sum_{i=1}^n 2^{i-1} s_i && (\text{selector (input)}) \\
a &:= \sum_{i=1}^m 2^{i-1} a_i && (\text{address (output)}) \\
c &:= \sum_{i=1}^o 2^{i-1} c_i && (\text{interconnects}) \\
I &= (i_1, i_2, \dots, i_{2n}) = (s_1, s_2, \dots, s_n, \overline{s_1}, \overline{s_2}, \dots, \overline{s_n}) && (\text{expanded input}) \\
c_j &:= \prod_{k=1}^{2n} i_k \cdot \text{AND}_{j,k} \\
a_j &\equiv \sum_{k=1}^o c_k \text{OR}_{j,k} \pmod{2}
\end{aligned}$$

Here, $\text{AND}_{j,k}$ ($\text{OR}_{j,k}$) represents the k -th bit of the j -th AND (OR) bitvector. The resulting configuration block then consists of these bitvectors:

$$\text{AND}_1, \text{AND}_2, \dots, \text{AND}_m, \text{OR}_1, \text{OR}_2, \dots, \text{OR}_o$$

4.2.4. Design Space

CLARIFY: should this go into an evaluation document?

Solution: The complete design decisions and tests yes, but a general overview about

what is the problem with design space, what options and what influences it would be good. But not that detailed like the evaluation document would be, so maybe leave out the paragraphs exploration and conclusion and reference instead to the evaluation document

4.2.4.1. Parameters, metrics

4.2.4.2. Exploration

4.2.4.3. Conclusion

4.3. Approximate ALU

4.3.1. Overview

The design document describes an approach towards a Scalable Approximate ALU that combines circuitry for clock gating and for scaling the approximation level. The latter has not deviated in its implementation however the former has changed considerably and has been redesigned and adjusted during the implementation phase. Clock gating certain bits in the ALU input registers require accessing and manipulating write-enable signals of single flip-flops. This level of granularity is not supported by the Chisel language. A work-around was designed by introducing two extra registers to store the operands of each arithmetic operation until the next cycle. If the next instruction is approximate, MSBs of the new operands are appended to LSBs of the previous ones. This means that LSBs maintain the same value in two consecutive cycles, hence the logic responsible for operating on them does not have any switching activity or consumes energy. 7 Multiplexers were also added to achieve 7 approximation levels. This design introduced a lot of overhead that would consume more energy than it would save. In addition, the complexity of the design introduced hazards that caused the precise operation of the pipeline to malfunction. The current implementation of scalable approximate ALU provides a platform which can achieve different levels of approximation. This would allow specialists from different domains to test applications with different level of approximation.

Approximating operands The idea behind the scalable ALU is to approximate operands of certain arithmetic operations before feeding them to the ALU or the Multiplier unit. Approximation of operands is scalable by different levels according to 6 approximation bits present in the instruction format of approximate instructions. As stated above in section 4.3.1, the aim of this strategy is to create a platform to serve as a multi-level approximate processor. This is done by neglecting number of LSBs⁵ from each operand. The desired quality of result must be specified by the

⁵The least significant bit is the lowest bit in a series of numbers in binary.

programmer in the code using approx decorators. For more details regarding approx decorators refer to the compiler section 3.

4.3.2. Original code base

rocket.scala *The Rocket chip CPU has a single-issue, in-order, five-stage-pipeline. Figure 13 shows a schematic of this design. This figure only shows details of the the decode and execute phase since only these are relevant for the modification to come. Every instruction has at most two input registers (rs, rt) as well as one result register (rd). The register's content is loaded from a 32 entry registerfile (Regfile) and these are sent to three functional units: ALU, Integer Divide (IDIV), and integer multiplier (IMUL). The ALU takes one cycle to complete an operation and the other two take several cycles. All outputs of the functional units are connected to a bypass network which allows new instructions to take their inputs from later phases of the pipeline, instead of the registerfile, in case of data dependency between consecutive instructions. A more detailed description of Rocket can be found in the links of the Appendix A.7.*

The controlpath for the pipeline can be found in the file `rocket.scala`. The ALU can be found in the file `dpath_alu.scala` and the IMUL functional unit can be found in the file `multiplier.scala`, which contains the IDIV as well.

4.3.3. Modified Code Base

Changes to the ALU The components modified for the ALU are listed below:

1. `idecode.scala`: The changes to the ALU requires a modification to the decoder. In order to execute the approximate instructions, a control signal - **precise**, has been added in the decoder to all the instructions. When the **precise** control signal is set to true (symbolised as 'Y'), the instruction is executed precisely, otherwise it is executed with scalable approximation 4.3.1.
The instructions **ADD** and **SUB** are executed approximately and are thus conveniently named **ADD_APPROX** and **SUB_APPROX**. These instructions have been added to the `idecode.scala` file. All the control signals of the **ADD_APPROX** instruction remains the same as the **ADD** instruction except for the precise signal (represented with an 'N'). This signal is set to false, activating the approximation logic of the core. The **SUB_APPROX** instruction has also been added similarly. For more details on how to extend the decoder, please read section 4.1.2.
2. `rocket.scala`: This file describes the implementation of the pipeline stages of the rocket-core. The operands are approximated before being sent to the ALU for execution (read section 4.3.1). The approximation logic is implemented in the execute stage of the ALU as shown in figure 13. The approximate logic is implemented in the class - **Rocket**. All the registers and the wires used in every pipeline stage is defined in this class. The following changes are made to the `rocket.scala` to incorporate the approximation logic:

- *Approximation Determination: In order to determine if an instruction needs to be executed precisely or approximately, the control signal identifying this is obtained in the wire `ex_ctrl.precise`. Whereas, the instruction to be executed is stored in the `ex_reg_inst` register. Bits 26 to 31 of this instruction identify the amount of approximation. This approximation value is then gathered in a bundle of wires named, `ex_app_bits` before being sent to the approximation logic. For more information on the instruction format, refer to the Design Document.*

Solution (workaround): Refer this to a sentence and place it in the footer

- *Approximation logic: As seen from the figure 13, the approximation logic is implemented in the execute stage of the pipeline. Since the operands are approximated before operation, following lines of code explain the relevant changes. **Solution (workaround):** Instead, explain an if block of the code line by line. Say, line 1 what it means, line 2 so on. Not the whole code. Just one block*

```
val ex_app_bits = ex_reg_inst(31,26)

when (!ex_ctrl.precise && (ex_app_bits.toUInt === 63 )){
  alu.io.in1 := Cat(ex_op1(xLen-1,2),Bits(0,2))
  alu.io.in2 := Cat(ex_op2(xLen-1,2),Bits(0,2))
}
.elsewhen(!ex_ctrl.precise && (ex_app_bits.toUInt === 62 )){
  alu.io.in1 := Cat(ex_op1(xLen-1,4),Bits(0,4))
  alu.io.in2 := Cat(ex_op2(xLen-1,4),Bits(0,4))
}
.elsewhen(!ex_ctrl.precise && (ex_app_bits.toUInt === 60 )){
  alu.io.in1 := Cat(ex_op1(xLen-1,7),Bits(0,7))
  alu.io.in2 := Cat(ex_op2(xLen-1,7),Bits(0,7))
}
.elsewhen(!ex_ctrl.precise && (ex_app_bits.toUInt === 56 )){
  alu.io.in1 := Cat(ex_op1(xLen-1,10),Bits(0,10))
  alu.io.in2 := Cat(ex_op2(xLen-1,10),Bits(0,10))
}
.elsewhen(!ex_ctrl.precise && (ex_app_bits.toUInt === 48 )){
  alu.io.in1 := Cat(ex_op1(xLen-1,15),Bits(0,15))
  alu.io.in2 := Cat(ex_op2(xLen-1,15),Bits(0,15))
}
.elsewhen(!ex_ctrl.precise && (ex_app_bits.toUInt === 32 )){
  alu.io.in1 := Cat(ex_op1(xLen-1,20),Bits(0,20))
  alu.io.in2 := Cat(ex_op2(xLen-1,20),Bits(0,20))
}
.elsewhen(!ex_ctrl.precise && (ex_app_bits.toUInt === 0 )){
  alu.io.in1 := Cat(ex_op1(xLen-1,26),Bits(0,26))
  alu.io.in2 := Cat(ex_op2(xLen-1,26),Bits(0,26))
}
.otherwise {
  alu.io.in1 := ex_op1.toUInt
}
```

```

alu.io.in2 := ex_op2.toUInt
}

```

In the above code snippet, `alu.io.in1` and `alu.io.in2` are the operands supplied to the ALU and these are approximated. The logic checks if the `precise` signal is set to false and then truncates the Least Significant Bits of the operands, replacing them with 0's based on the approximation bits supplied.

Changes to the Multiplier The multiplier has been modified in a similar way to the ALU for approximation. The components listed below were modified:

1. `idecode.scala`: Implementing the `MUL_APPROX` instruction follows the same steps as with the `ADD_APPROX` instruction. The control signals of the `MUL_APPROX` is similar to that of the `MUL` instruction except for the `precise` signal being set to false.
2. `rocket.scala`: Similar to the approximate ALU, approximation of the multiplier requires the operands to be approximated before execution. The following code snippet describes the changes:

```

when (!ex_ctrl.precise && (ex_app_bits.toUInt === 63 )){
  div.io.req.bits.in1 := Cat(ex_rs(0)(xLen-1,2),UInt(0,2))
  div.io.req.bits.in2 := Cat(ex_rs(1)(xLen-1,2),UInt(0,2))
}
.elsewhen(!ex_ctrl.precise && (ex_app_bits.toUInt === 62 )){
  div.io.req.bits.in1 := Cat(ex_rs(0)(xLen-1,4),UInt(0,4))
  div.io.req.bits.in2 := Cat(ex_rs(1)(xLen-1,4),UInt(0,4))
}
.elsewhen(!ex_ctrl.precise && (ex_app_bits.toUInt === 60 )){
  div.io.req.bits.in1 := Cat(ex_rs(0)(xLen-1,7),UInt(0,7))
  div.io.req.bits.in2 := Cat(ex_rs(1)(xLen-1,7),UInt(0,7))
}
.elsewhen(!ex_ctrl.precise && (ex_app_bits.toUInt === 56 )){
  div.io.req.bits.in1 := Cat(ex_rs(0)(xLen-1,10),UInt(0,10))
  div.io.req.bits.in2 := Cat(ex_rs(1)(xLen-1,10),UInt(0,10))
}
.elsewhen(!ex_ctrl.precise && (ex_app_bits.toUInt === 48 )){
  div.io.req.bits.in1 := Cat(ex_rs(0)(xLen-1,15),UInt(0,15))
  div.io.req.bits.in2 := Cat(ex_rs(1)(xLen-1,15),UInt(0,15))
}
.elsewhen(!ex_ctrl.precise && (ex_app_bits.toUInt === 32 )){
  div.io.req.bits.in1 := Cat(ex_rs(0)(xLen-1,20),UInt(0,20))
  div.io.req.bits.in2 := Cat(ex_rs(1)(xLen-1,20),UInt(0,20))
}
.elsewhen(!ex_ctrl.precise && (ex_app_bits.toUInt === 0 )){
  div.io.req.bits.in1 := Cat(ex_rs(0)(xLen-1,26),UInt(0,26))
  div.io.req.bits.in2 := Cat(ex_rs(1)(xLen-1,26),UInt(0,26))
}
.otherwise {
  div.io.req.bits.in1 := ex_rs(0)
  div.io.req.bits.in2 := ex_rs(1)
}

```


In the above code snippet `div.io.req.bits.in1` and `div.io.req.bits.in2` are the operands fed to the multiplier. The rest logic is implemented the same way as the approximate ALU.

4.4. Power estimation

TODO (MUST): state the objective

The energy consumption of the PACO core has to be estimated in order to compare its savings over typical precise cores. Instructions such as approximate ADD and MULTIPLY and their precise counterparts are compared on the original core. This data can be obtained using FPGA or ASIC tools for power estimation. Xilinx FPGA tools are available and are easy to use however the power estimation is that of an FPGA implementation of the core, which differs from the performance of an ASIC Chip. ASIC tools used by UC Berkley are not open source or freely available.

TODO (MUST): discuss difficulty

4.4.1. Xilinx Power Analyzer

TODO (MUST): describe the tool

Xilinx Power Analyzer(xpa) is a tool available with Xilinx ISE to obtain the power results post placement and routing stages. It requires net lists and saif files as input. Saif files are generated after simulation and include switching activity for estimation of dynamic power consumption.

TODO (MUST): state experiments conducted

The FPGA Toolchain was used to obtain static and dynamic power estimates. The following steps were followed for the power results:

- A dummy design of an Adder was synthesized, placed and routed using the Xilinx ISE tool. These steps produced net lists in the form of .ncd files.
- A testbench with varying inputs was run using isim and an .saif file was generated. This file was generated with the command

```
saif open
```

in the *isim* console before the start of the simulation and

```
saif close
```

after the end of simulation.
- These steps record the switching activity from the runs in the saif file.
- The .ncd and the .saif have to be input to the xpa for estimating the static and the dynamic power. This can be done by opening *X power Analyzer* from the tools tab in the ise.

TODO (MUST): state results

The above design of the 32-bit Adder yielded the following estimates:

- For a testbench of multiple input pairs spanning 6 cycles:
 - static power consumption was 3.423
 - dynamic power consumption was 0.016
- For a testbench of multiple input pairs spanning 1000 cycles:
 - static power consumption was 3.423
 - dynamic power consumption was 0.040

The reason for the static power consumption being considerably larger than the dynamic power consumption could be due to the adder being small compared to the overall size of the FPGA.

4.4.2. Synopsis Tool

TODO (MUST): describe the tool

Another option would be to use the Synopsis Tools in order to synthesize the estimate the power with the ASIC tool flow. This suite consists of several tools such as:

- Synopsis' Verilog Compiler and Simulator (VCS) is specifically designed to simulate and debug ASIC designs. VCS compiles the Verilog source into an object source or C source files. The C compiler is invoked to generate the simulation executable. This executable is used to simulate the designs.
- Synopsis Design Compiler: This tool is used to synthesize the design and it transforms RTL into gate level net lists.
- Synopsis Formality: This tool is used to verify that the RTL and gate level models match. VCS is used once again to simulate the gate level model.
- Synopsis IC compiler: Once gate level netlist is obtained IC Compiler is used for placement and routing of the design. VCS is used to simulate the final gate level netlist.
- Prime Time PX: This tool takes the gate level netlist and switching activity from the simulation and provides accurate numbers on power consumption.

No experiments have been conducted so far with the Synopsis tools as they are not yet available. Some tools may be obtained in the near future. The next step would be to obtain the missing ones or find a work around without using them.

TODO (MUST): state experiments conducted

TODO (MUST): state results

TODO (MUST): add evaluation of both alu an lut using different applications

TODO (MUST): all details regarding evaluation and performance metrics

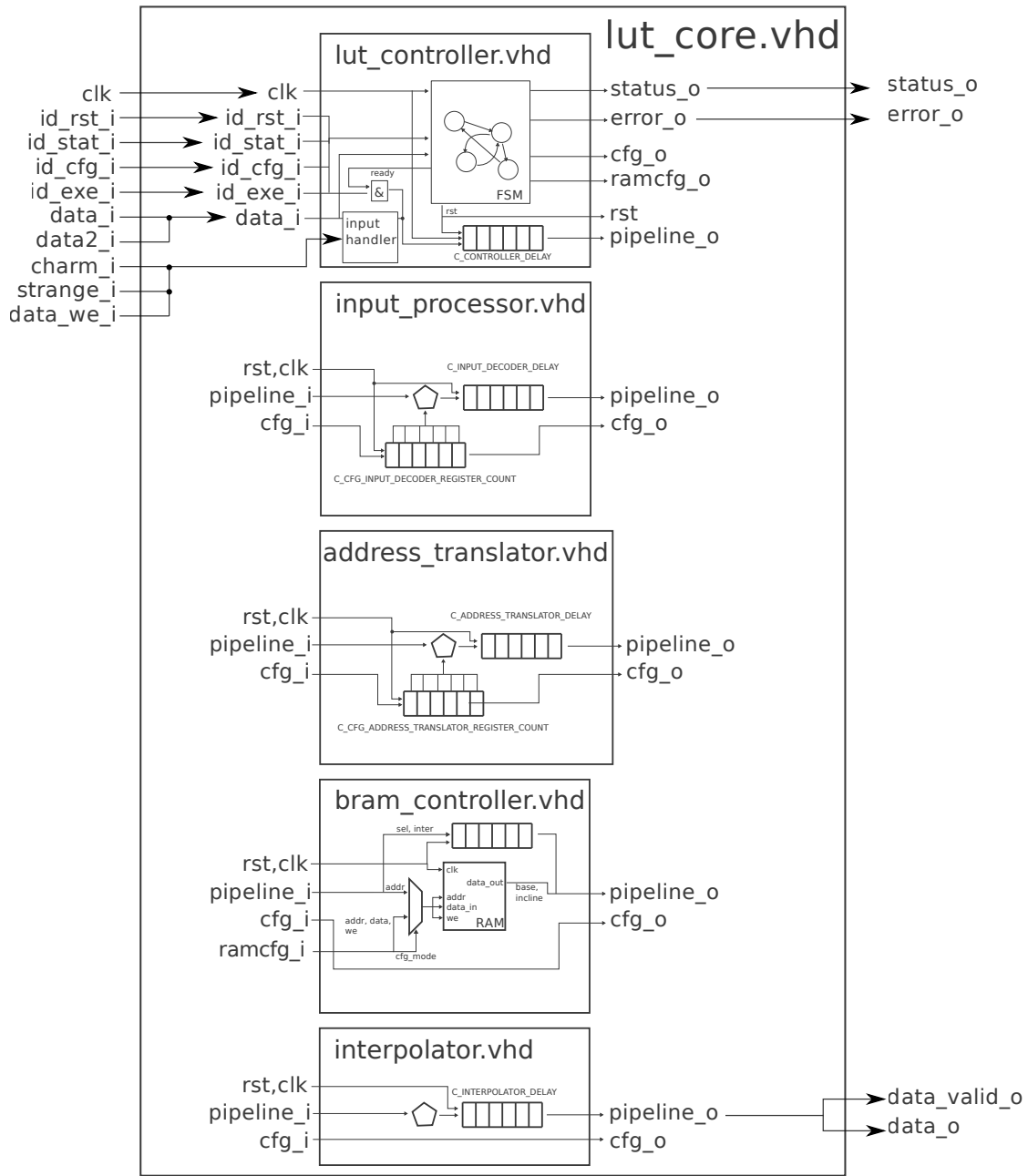


Figure 10: Overview of the LUT Hardware core. Each box shows a single component found in a single VHDL file. Connections of reset and clock signals are implemented trivially and thus not shown here. Signals pipeline and cfg are connected in a daisy-chain manner and the ramcfg_i input of the bram controller is connected with the ramcfg_o output of the lut controller.

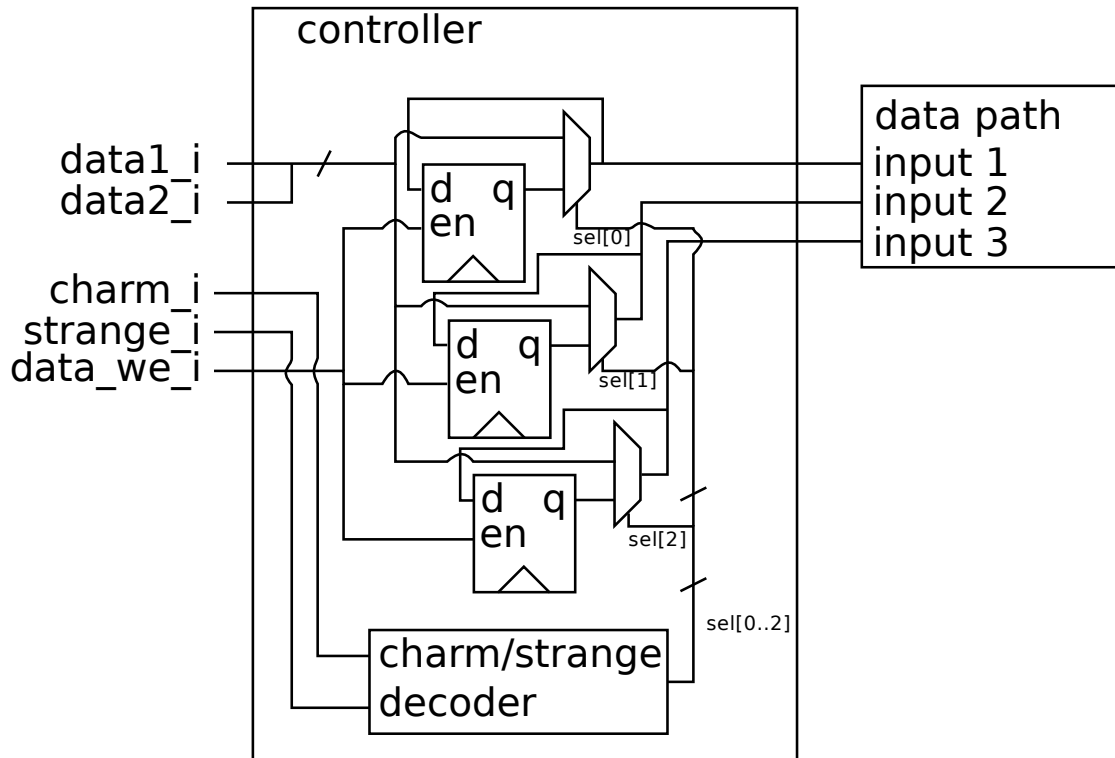
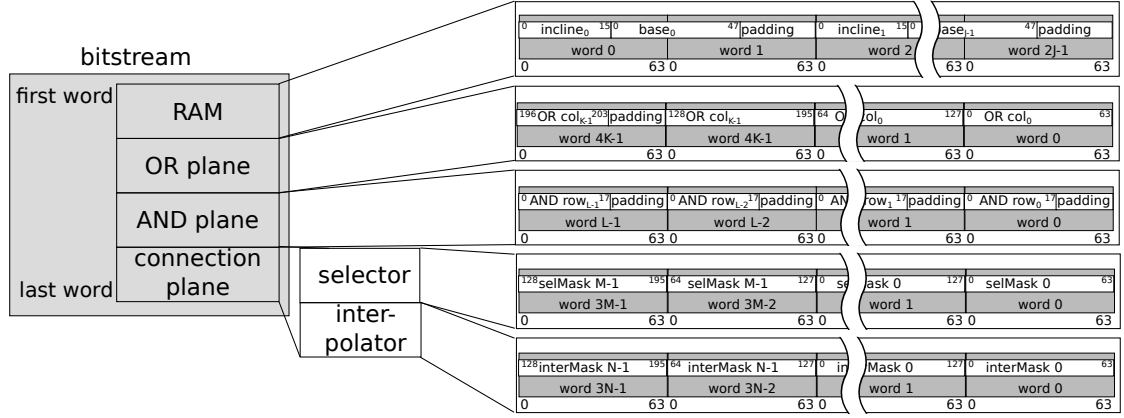
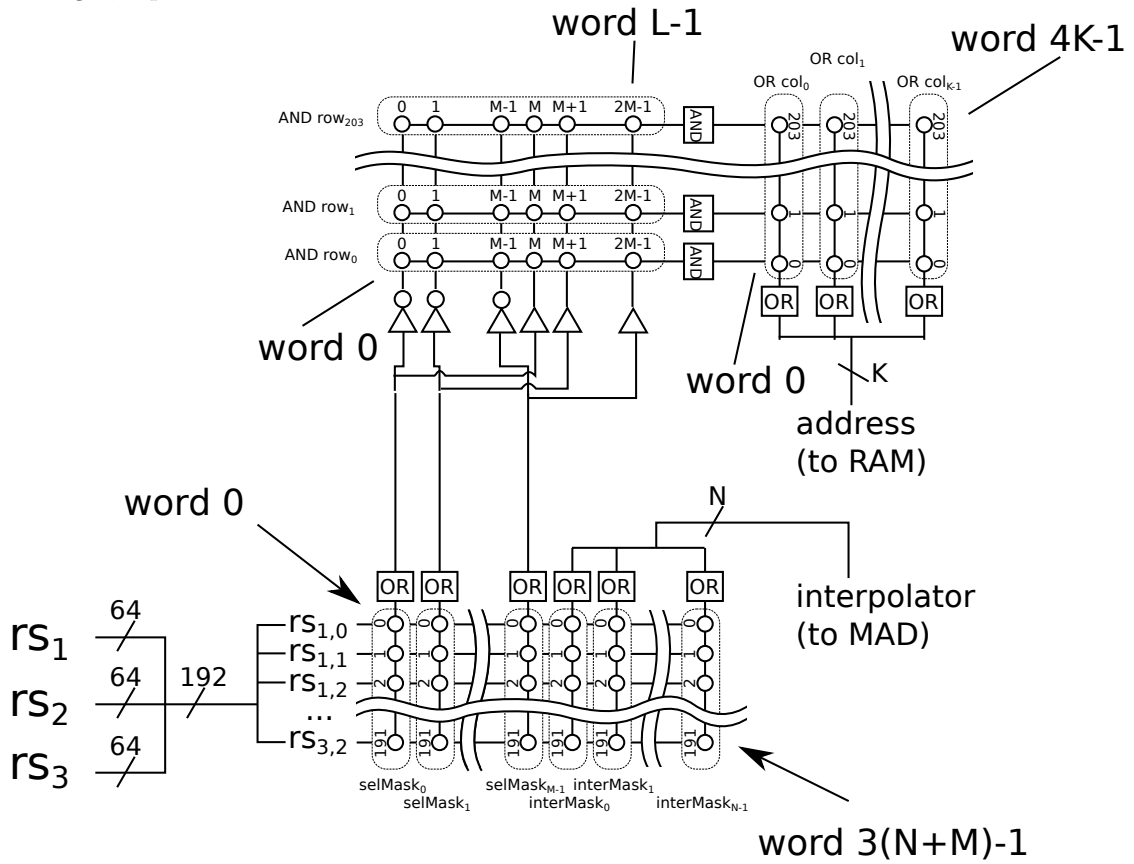


Figure 11: Processing of input data into the LUT core. data1.i and data2.i are fed via an instruction and are used to update internal input registers, driving the data path inputs.



(a) Overview of bitstream components. The order of configuration words is to be read left-to-right, top-to-bottom.



(b) Structural depiction of the LUT data path excluding the interpolator and RAM, showing the mapping of bitstream data to crosspoints in the connection plane and PLA. Word indices are relative to the respective bitstream component, crosspoints are labeled with the bit index of the value corresponding to the group of crosspoints encompassed with a dotted frame. Note that the interconnection plane combines both the selector and interpolator bitstream components.

Figure 12: Depiction of the LUT Hardware core bitstream. Values shown are corresponding to a LUT core configuration with K segment bits, $J = 2^K$ segments, L PLA interconnects, $M = 9$ selector bits and N interpolation bits. Each segment is represented by 16 incline bits and 48 base bits.

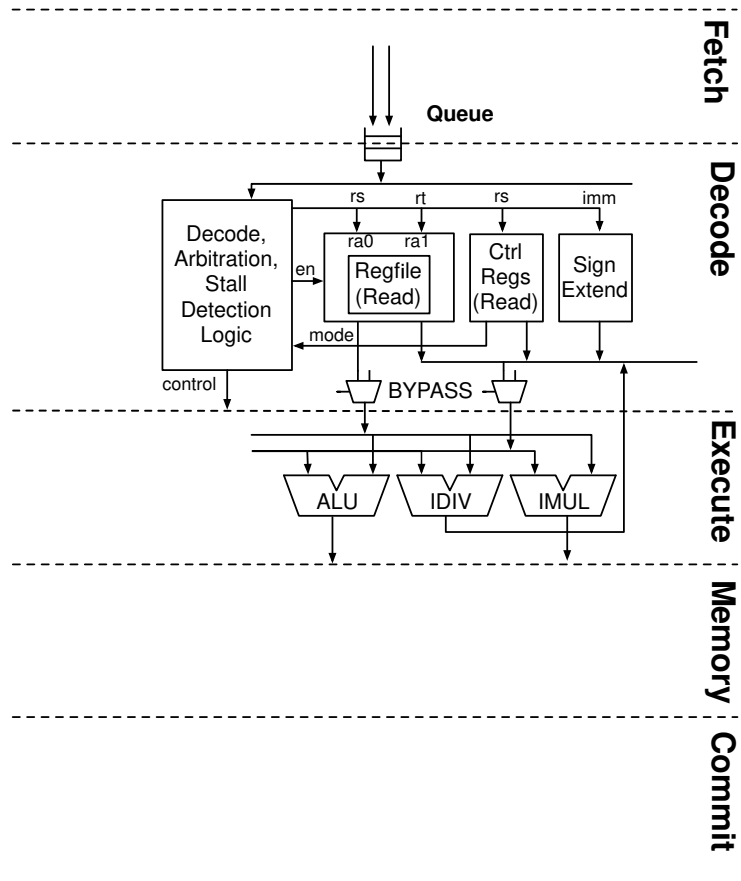


Figure 13: Rocket core pipeline stages

Appendices

A. External resources

- Design Document

TODO (MUST): add links to external material, website link

TODO (MUST): download this material (if appropriate), store it in /mat and refer to it as well

A.1. QEMU

Useful material for understanding QEMU can be found in these external resources:

- The official manual in form of a wiki: <http://www.qemu.org/Manual>
- Additional information about the internals of QEMU:
<https://qemu.weilnetz.de/qemu-tech.html>

A.2. Clang/LLVM

Useful material for understanding LLVM can be found in these external resources:

- The official manual in form of a wiki: <http://llvm.org/docs/>
- Documentation on intrinsic functions:
<http://llvm.org/docs/LangRef.html#intrinsic-functions>
- Additional information on data types can be found here:
<http://llvm.org/docs/LangRef.html#t-firstclass>
- Additional information about adding a backend can be found here:
<http://llvm.org/docs/WritingAnLLVMBackend.html>
- Additional information about adding intrinsic functions can be found here:
<http://llvm.org/docs/ExtendingLLVM.html>
- Additional information about writing test cases can be found here:
<http://llvm.org/docs/TestingGuide.html>

A.3. GNU/Binutils

A.4. Chisel

- The official homepage of Chisel: <https://chisel.eecs.berkeley.edu/>
- Documentation of Chisel:
<https://chisel.eecs.berkeley.edu/documentation.html>

A.5. Scala

- Official Scala Website <http://www.scala-lang.org/>
- The official documentation of Scala:
<http://www.scala-lang.org/documentation/>
- Scala School : https://twitter.github.io/scala_school/

A.6. RISC-V

A.7. Rocket Chip

A.8. Rocket SoC

List of Tables

1.	PACO hardware component overview: Short descriptions of major hardware components in the PACO approximate computing core, as well as the FPGA interface needed to run applications it. The last column contains references to more detailed descriptions of the components.	9
2.	PACO software component overview: Short descriptions of major components needed to compile/generate software for the PACO approximate computing core. The last column contains references to more detailed descriptions of those components.	10
3.	Table containing the original repositories without any modification and the place where they should be checked out.	29

List of Figures

1.	An overview over the workflow in the PACO tools. On the left side are the Hardware descriptions and how they can be transformed into three different platform. The right side shows the workflow to create a program to be run on these platforms	21
2.	A directory tree of the most important files and directories with a short description on their purpose. Git repositories are marked with a git symbol, folders with a folder symbol, and build or installation folders have a gear symbol. Every directory surrounded by a solid white box is entirely written for PACO, every directory surrounded by a grey box contains modified code from the original repositories, and every other directory is untouched.	28
3.	Toolflow for generating executable utilizing the PACO language extensions and hardware cores.	42
4.	Example of PACO tree for approximate computation analysis	47
5.	Examples of segmentation strategies. Each group shows the designated domain to be segmentated and the segment intervals placed. Note that for error minimization, an approximation strategy must also be specified.	54
6.	Examples of approximation strategies. The target function is shown in black, being approximated by an affine linear function (blue).	55
7.	Overview of the important source code files of the LUT Compiler Tool.	56
8.	Coordinate systems used in the LUT Compiler Tool for an exemplary translation of the highlighted input domain. In this example the segment space is 2^{15} value wide with an offset of 7168. With 2 selector bits the segment space is subdivided into 16 principal segments. They are reduced to 4 segments (2 segment bits). Dotted lines in the hardware space indicate individual evaluable points which result from a single interpolation bit.	57

9.	Overview of the PACO CPU core. The Rocket Chip implementation of the RISC-V CPU core is extended with an approximate ALU and a Lookup-Table (LUT). The LUT core itself is written in VHDL while all of the remainder is described in Chisel.	60
10.	Overview of the LUT Hardware core. Each box shows a single component found in a single VHDL file. Connections of reset and clock signals are implemented trivially and thus not shown here. Signals pipeline and cfg are connected in a daisy-chain manner and the ramcfg_i input of the bram controller is connected with the ramcfg_o output of the lut controller. . .	76
11.	Processing of input data into the LUT core. data1_i and data2_i are fed via an instruction and are used to update internal input registers, driving the data path inputs.	77
12.	Depiction of the LUT Hardware core bitstream. Values shown are corresponding to a LUT core configuration with K segment bits, $J = 2^K$ segments, L PLA interconnects, $M = 9$ selector bits and N interpolation bits. Each segment is represented by 16 incline bits and 48 base bits. . . .	78
13.	Rocket core pipeline stages	79

Special files

env.sh Bash script to be sourced before the PACO environment can be used. Sets up.
7, 11, 30, 31

qemu/images/bbl Default bootloader image for use with emulating RISC-V in QEMU.. 32, 35

qemu/images/rootfs.ext2 Default root file system for use with emulating RISC-V in QEMU.. 31, 32, 35

qemu/images/vmlinux Default kernel image for use with emulating RISC-V in QEMU.. 31, 35

qemu/target-riscv/instmap.h Instruction decode header for the RISC-V QEMU target.. 36, 37

qemu/target-riscv/translate.c Complete QEMU intermediate code generation from RISC-V instructions.. 36, 37

qemu/tcg/README Reference for the QEMU intermediate code (TCG).. 36

riscv-tools . 7, 11, 12, 40

riscv-tools-src/build.sh Bash script building RISC-V toolchains fesvr, isa-sim, gnu-toolchain, pk and riscv-tests.. 30

riscv-tools-src/riscv-gnu-toolchain/binutils/gas/config/tc-riscv.c . 51

riscv-tools-src/riscv-gnu-toolchain/binutils/opcodes/riscv-dis.c . 52
riscv-tools-src/riscv-gnu-toolchain/binutils/opcodes/riscv-opc.c . 51
riscv-tools-src/riscv-llvm/include/llvm/InitializePasses.h . 50
riscv-tools-src/riscv-llvm/include/llvm/IR/IntrinsicsRISCV.td . 49
riscv-tools-src/riscv-llvm/include/llvm/Transforms/PACO/Paco.h . 50
riscv-tools-src/riscv-llvm/lib/Target/PowerPC/AsmParser/RISCVAsmParser.cpp . 49
riscv-tools-src/riscv-llvm/lib/Target/RISCV/RISCVInstrFormats.td . 49
riscv-tools-src/riscv-llvm/lib/Target/RISCV/RISCVInstrInfo.td . 49
riscv-tools-src/riscv-llvm/lib/Target/RISCV/RISCVOperands.td . 49
riscv-tools-src/riscv-llvm/lib/Transforms/PACO/LutTranslate.cpp . 50
riscv-tools-src/riscv-llvm/lib/Transforms/PACO/paco.cpp . 50
riscv-tools-src/riscv-llvm/riscv-clang/include/clang/AST/Decl.h . 44
riscv-tools-src/riscv-llvm/riscv-clang/include/clang/AST/Expr.h . 47
riscv-tools-src/riscv-llvm/riscv-clang/include/clang/Basic/DiagnosticParseKinds.td . 43
riscv-tools-src/riscv-llvm/riscv-clang/include/clang/Basic/DiagnosticSemaKinds.td . 43
riscv-tools-src/riscv-llvm/riscv-clang/include/clang/Basic/LangOptions.def . 43
riscv-tools-src/riscv-llvm/riscv-clang/include/clang/Basic/PACO.h . 43
riscv-tools-src/riscv-llvm/riscv-clang/include/clang/Basic/TokenKinds.def . 43, 45
riscv-tools-src/riscv-llvm/riscv-clang/include/clang/Sema/Sema.h . 45
riscv-tools-src/riscv-llvm/riscv-clang/include/clang/Sema/SemaExpr.cpp . 47
riscv-tools-src/riscv-llvm/riscv-clang/lib/CodeGen/CGExpr.cpp . 48
riscv-tools-src/riscv-llvm/riscv-clang/lib/CodeGen/CGExprScalar.cpp . 48
riscv-tools-src/riscv-llvm/riscv-clang/lib/CodeGen/CodeGenFunction.cpp . 48
riscv-tools-src/riscv-llvm/riscv-clang/lib/Parse/ParseDecl.cpp . 43, 44
riscv-tools-src/riscv-llvm/riscv-clang/lib/Parse/ParseExpr.cpp . 46

riscv-tools-src/riscv-llvm/riscv-clang/lib/Parse/ParsePragma.cpp . 45
riscv-tools-src/riscv-llvm/riscv-clang/lib/Parse/ParsePragma.h . 45
riscv-tools-src/riscv-llvm/riscv-clang/lib/Parse/Parser.cpp . 45
riscv-tools-src/riscv-llvm/riscv-clang/lib/Parse/ParseStmt.cpp . 45
riscv-tools-src/riscv-llvm/riscv-clang/lib/Sema/SemaDecl.cpp . 44
riscv-tools-src/riscv-llvm/riscv-clang/test/PACO/ApproxGeneralAndALU.cpp . 48
riscv-tools-src/riscv-llvm/riscv-clang/test/PACO/ApproxLUT.cpp . 48
riscv-tools-src/riscv-llvm/test/CodeGen/RISCV . 49
riscv-tools-src/riscv-llvm/tools/llc/llc.cpp . 50
riscv-tools-src/riscv-lut-compiler . 56
riscv-tools-src/riscv-lut-compiler/arch-config.h . 57
riscv-tools-src/riscv-lut-compiler/error.h . 58
riscv-tools-src/riscv-lut-compiler/make.incl . 56
riscv-tools-src/riscv-lut-compiler/riscv-lut-compiler.cpp . 56
riscv-tools-src/riscv-lut-compiler/strategies . 56
riscv-tools-src/riscv-lut-compiler/strategies_decl.h . 56
riscv-tools-src/riscv-lut-compiler/tests . 59
riscv-tools-src/riscv-opcodes/opcodes . 51
riscv-tools-src/riscv-opcodes/parse-opcodes . 51
riscv-tools-src/riscv-tests/isa/ . 17
riscv-tools-src/riscv-tests/isa/macros/scalar/test_macros_appr_scalar.h . 34
riscv-tools-src/riscv-tests/isa/rv64ui/add_approx.S . 33
riscv-tools-src/riscv-tests/isa/rv64ui/Makefrag . 34
riscv-tools/bin/riscv-uart-flash . 19, 39
riscv-tools/py . 31
riscv-tools/riscv64-unknown-elf/share/riscv-tests/isa/ . 17
rocket-chip/ . 17, 34

rocket-chip/emulator/ . 17
rocket-chip/emulator/output/ . 17
rocket-chip/fsim/generated-src/Top.PACOConfigFPU.v . 30
rocket-chip/fsim/Makefile . 30, 41
rocket-chip/rocket/src/main/scala/decode.scala . 61
rocket-chip/rocket/src/main/scala/dpath_alu.scala . 70
rocket-chip/rocket/src/main/scala/frontend.scala . 61
rocket-chip/rocket/src/main/scala/idecode.scala . 61, 62, 70, 72
rocket-chip/rocket/src/main/scala/multiplier.scala . 70
rocket-chip/rocket/src/main/scala/rocket.scala . 61, 62, 70, 72
rocket-soc/rocket_soc/fw/boot/linuxbuild/bin/bootimage.hex . 31
rocket-soc/rocket_soc/fw/boot/src/ . 30
rocket-soc/rocket_soc/fw/boot/src/main.c . 30
rocket-soc/rocket_soc/fw/boot/src/trap.c . 30
rocket-soc/rocket_soc/fw_images/ . 31
rocket-soc/rocket_soc/lib . 19, 40
rocket-soc/rocket_soc/lib/templates/ . 13
rocket-soc/rocket_soc/lib/templates/lut-application . 52
rocket-soc/rocket_soc/prj/rocket_soc.bit . 13
rocket-soc/rocket_soc/prj/rocket_soc.xise . 13
rocket-soc/rocket_soc/rocket-lut/htlib . 66
rocket-soc/rocket_soc/rocket-lut/system-test . 67
rocket-soc/rocket_soc/rocketlib/misc/nasti_uart.vhd . 39
rocket-soc/rocket_soc/rocketlib/Top.GnssConfigNoFPU.v . 7, 12, 30, 41

Repositories

qemu Git repository `qemu`. 29, 35

riscv-tools-src Git repository `riscv-tools`. 29, 30, 33

riscv-tools-src/riscv-fesvr Git repository `riscv-tools-fesvr`. 29

riscv-tools-src/riscv-gnu-toolchain Git repository `riscv-tools-gnu-toolchain`. 29, 51

riscv-tools-src/riscv-isa-sim Git repository `riscv-tools-isa-sim`. 29

riscv-tools-src/riscv-llvm Git repository `riscv-tools-llvm`. 29, 42, 48

riscv-tools-src/riscv-llvm/tools/clang Git repository `riscv-tools-llvm-clang`. 29, 42

riscv-tools-src/riscv-opcodes Git repository `riscv-tools-opcodes`. 29, 51

riscv-tools-src/riscv-pk Git repository `riscv-tools-pk`. 29

riscv-tools-src/riscv-tests Git repository `riscv-tools-tests`. 29, 31, 33, 34

riscv-tools-src/riscv-tests/env Git repository `riscv-tools-tests-env`. 29

rocket-chip Git repository `rocket`. 29, 34

rocket-chip/chisel Git repository `rocket-chisel`. 29

rocket-chip/context-dependent-environments Git repository `rocket-context-dependent-environments`. 29

rocket-chip/dramsim2 Git repository `rocket-dramsim2`. 29

rocket-chip/groundtest Git repository `rocket-groundtest`. 29

rocket-chip/hardfloat Git repository `rocket-hardfloat`. 29

rocket-chip/junctions Git repository `rocket-junctions`. 29

rocket-chip/rocket Git repository `rocket-rocket`. 29, 30, 62

rocket-chip/torture Git repository `rocket-torture`. 29

rocket-chip/uncore Git repository `rocket-uncore`. 29

rocket-chip/zscale Git repository `rocket-zscale`. 29

rocket-soc Git repository `rocket-soc`. 29, 40

rocket-soc/rocket_soc/rocket-lut Git repository `rocket-lut`. 64