Paderborn CPU Core for Approximate Computing

Design Document

September 30, 2016

Contents

1 Introduction

The PACO (Paderborn CPU Core for Approximate Computing) development platform is a project group effort by students and advisors of the University of Paderborn. *Approximate Computing*, as we understand it, provides calculation facilities to application programmers that allow them to trade off precision or probability-of-correctness for other desired qualities: e.g. reduced delay, higher energy efficiency.
We perceive a lack of a complete stack of tools and general-purpose computing platform to evaluate new approximate implementations of functional units (hardware). With our development platform, we want to enable exploration of new approximate hardware components in an ordinary general-purpose processor.
This includes the whole stack:

- A modular RISC-V processor template with attached peripherals (memory, UART etc.) synthesizable to FPGA.
- An easy-to-extend compiler toolchain that allows hardware developers to create new (approximate) opcodes and programmers to write code (annotated C++) that can be compiled into machine code containing those approximate opcodes.
- A testing framework to test new instructions and hardware functional units in an automated fashion.
- Example applications that are compiled using our toolchain.
- A QEMU just-in-time binary translation testbench for approximate applications.
- Two complex approximate functional unit designs useful for approximation, integrated into the CPU pipeline as examples.

We have chosen two previously unexplored approaches of approximate functional units to integrate in the pipeline: One is an approximate ALU capable of common ALU operations (e.g. $+$, $-$, $*$) but with different possible degrees of precision, specified at application compile time. The other, less conventional, functional unit is supposed to evaluate complex functions approximately, functions which are not normally present as stand-alone instructions in a processor. Underlying technology is a Lookup Table.
This *Design Document* is not an all-encompassing high-detail specification of the complete platform, but a succinct description of our goals, critical design problems, proposed solutions and specification of interfaces between work packages, allowing us to compartmentalize implementation.
The next section “2 System Architecture Overview” contains an outline of the whole PACO platform, hardware and software, as well as references to sections explaining the details of each component. Section 3 – “Approximation Techniques” is a detailed description of the proposed inner workings of the proposed approximate functional units in the CPU as well as interfaces towards the compiler and the CPU. Section 4 – “Compiler/Language” – contains a specification of all compiler interfaces, towards the application programmer and the hardware. Section 5 is the Conclusion for our Design phase.

To external readers we would also like to point out that other documents precede this Design Documentation: The Project Plan (stating our timeline and detailing our avenues of exploration) and Exploration Documentation (summing up the results of our exploration of the design space and setting up our design decisions).

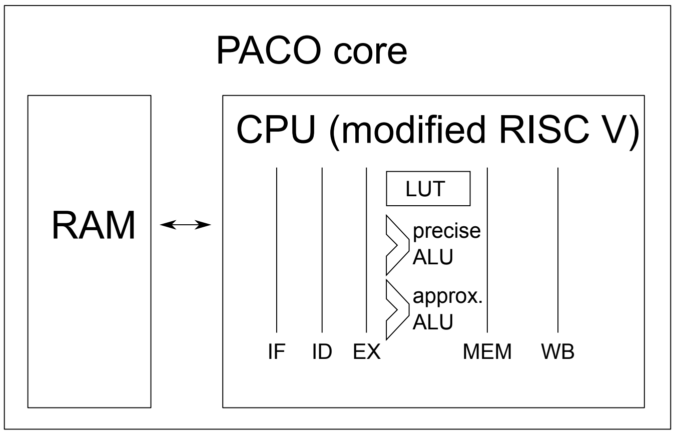


Figure 1: Overview of the hardware core design.

2 System Architecture Overview

The PACO Approximate Computing System laid out in this document is composed of hardware and software. The approximate hardware components created by our group are functional units in the execution stage of the pipeline, which can perform the approximate operations required by applications.
The software primarily consists of a compiler toolchain that allows a user to generate binaries that can compute approximate results on the hardware created in this project.

The rest of this section will outline the PACO system architecture as an orientation aid for the design description in sections 3 and 4.

2.1 Hardware-Side

An overview of the hardware has been presented in the Project Plan document. For reference, Figure 1 is included as an update of the core overview presented in the Project Plan: We have chosen the Rocket[1] chip implementation of the RISC-V ISA as a template for extension with approximate functionality. We must modify the ISA (see Sections 3.1, which means modifying the decoder, and we will add new functional units to the execution stage of the processor pipeline. These new functional units are an ALU capable of approximate calculations (see Section 3.2) and a Lookup Table unit capable of approximate evaluation of complex functions (see Section 3.3).

2.2 Software-Side

Our compiler system is built on the CLANG frontend for C/C++, an LLVM backend used for code optimization and target code generation as well as the binutils from the RISC-V toolchain that allow assembly and linking executables.
Every stage of this toolchain will be extended by us to support the approximation units added to the RISC-V processor, see Figure 2.

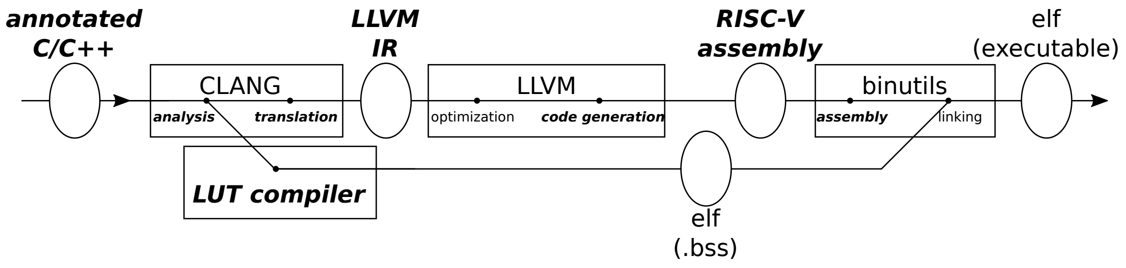


Figure 2: Overview of the compiler toolflow used by PACO. Parts that will be modified or created by us are set in bold italic.

Modifications to the CLANG frontend include analysis and translation stages (see Section 4.2). Frontend output is LLVM intermediate representation (IR) code (see Table 11), which is going to be extended to support additional instructions for our approximation units.
The LLVM backend (Section 4.3) optimization steps are mostly independent of the extensions made to the IR and thus do not need to be modified. In contrast, the code generation stage (Section 4.3.1) is extended to translate approximate instructions from the LLVM IR to RISC-V-targeted assembly code, requiring extension of assembly language mnemonics to include approximate instructions. These additional mnemonics are then converted to binary object code by the RISC-V binutils assembler. Finally the binutils linker (Section 4.3.3) must combine all object code into a single binary file.

Lookup Table Configuration Generator One of the proposed approximate functional units is a programmable Lookup Table (LUT) in the execution stage of the pipeline. This LUT is supposed to evaluate complex (normally not included in an instruction set) functions approximately. Configuration for the LUT functional units has to be generated on a per-application and per-complex-function basis. So, in parallel to the normal compiler toolchain, specifications for Lookup table usage must be extracted from source code in the CLANG frontend and processed in a separate program called the *LUT compiler* or *LUT configuration generator* (see Section 4.4). This program inputs source code representing a function to be approximated and outputs object code containing static data representing the configuration and ROM bitstreams of a Lookup table.

Immediately after loading a program the Lookup table hardware components are configured according to the previously generated bitstream (see Section 4.4.4). Since multiple compilation units may yield LUTs, knowledge of all the configurations is not present before the linking stage. Thus the linker must also manage LUT hardware cores and assign configuration bitstreams to them as well as translating LUT-related instructions to use the correct LUT hardware core.

3 Approximation Techniques

A useful method of approximate computing must create a benefit in exchange for the error introduced during computation. Approximation techniques allow this exchange, but the exchange rates depend on the application: e.g. one application might be tolerant of a certain percentage of calculations being plain wrong (error rate) while another might be tolerant of all calculations deviating by a certain amount from the correct result (deviation).

We have designed two approximation techniques trading off correctness for energy efficiency and/or reduction of computation delay. These will be specified in this section.

3.1 Description of Existing Core

We will implement our approximate functional units in a RISC-V processor. As a basis for understanding the proposed modifications, this subsection explains relevant structure and behaviour of the the Rocket System-on-Chip we will use as a template for PACO.

RISC-V The RISC-V processors are written in Chisel [1]. Chisel is a new open source hardware construction language developed at UC Berkeley that supports advanced hardware design in an extension of the Scala programming language. It raise the level of abstraction of a hardware design by providing concepts like object orientation, functional programming, parameterized types and type inference.
A hardware description provided in Chisel is generated into appropriate low-level Verilog code.

ISA The RISC-V ISA is composed of modules [2]. Each module defines an interface for a functionality in the ISA. The RISC-V ISA is defined as a base integer ISA. The base integer ISA is very similar to that of early RISC processors like MIPS or OpenRISC. The difference is that RISC-V has no branch delay slots and supports optional variable-length instruction encodings. Each base integer instruction set is characterized by the width of the integer registers and the corresponding size of the user address space.
Each implementation of the RISC-V ISA has to implement the basic modules, either RV32I or RV64I. Any other module is optional. The base integer ISA can be extended with one or more optional instruction-set extensions, but the base integer instructions cannot be redefined.

Extensions RISC-V is divided into standard and non-standard extensions.
Standard extensions are useful for common applications and do not conflict with other standard extensions. Non-standard extensions are specialized and may conflict with other non-standard extensions as well as standard extensions, not needed for this extension. If possible, conflicts between standard extensions should be avoided.

[1]https://chisel.eecs.berkeley.edu/

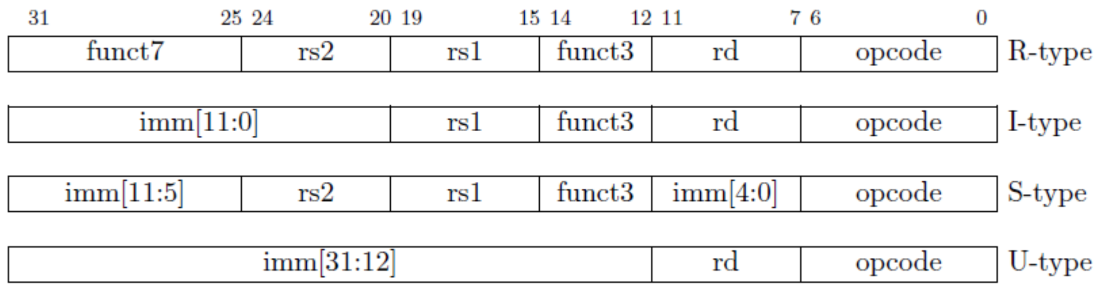


Figure 3: Base instruction formats of RISC-V. The opcode field selects the operation to be performed. "rd" is the destination register. "rs1" and "rs2" are the source registers. The "imm" field denotes the immediate value. "funct3" as well as optional "funct7" control operations for functional units with separate decoders. "funct7" is used in the R-type format, "funct3" and "funct7" together select the type of operation for R-type instructions. The formats in RISC-V are little-endian.

A set of standard extensions are defined to provide integer multiply/divide, atomic operations and single-double precision floating point arithmetic.
The base integer ISA are named I, prefixed with RV32 and RV64 (RV32I and RV64I) based upon the integer register width and contain integer loads, integer stores and control flow instructions.

Standard Extensions

- The standard integer multiplication and division extension is named 'M' and includes instructions to multiply and divide values stored in two integer registers.
- The standard atomic instructions are denoted with 'A'. This module contains instructions to atomically read, write and modify memory for inter-process synchronization.
- The standard single-precision floating point extension is denoted by 'F', and adds floating-point registers, single-precision computational instructions, and single-precision loads and stores.
- The double precision floating point extension is denoted by 'D' which expands floating pointing registers and adds double-precision computational instructions, the loads and stores.

Instruction formats In the base ISA, there are four core instruction formats (R/I/S/U) as shown in Figure 3. All are 32 bits fixed length formats. Figure 3 displays the R/I/S/U format:

- R-type instructions are register type instructions, most integer and floating point computations use R-type format for register-register operations. Along with integer and floating point computations, atomic memory operations (AMO) instructions are encoded in R-type instruction format.
- I-type instructions are shorthand for immediate instructions. Integer computations using register-immediate operations are encoded using I-type format. Apart from the integer computation indirect jump instruction (JALR), loads, shifts and system instruction uses I-type encoding.
- Store instructions are encoded using S-type format.
- Load upper immediate(LUI) uses U-type format. This instruction is used to build 32-bit constants and places the U-immediate value in the top 20 bits of the destination register, filling in the lowest 12 bits with zeroes. AUIPC (add upper immediate to PC[2]) also build PC-relative addresses using U-type format.

For our scalable ALU we plan to use R-type format for our instructions. The RISC-V ISA keeps the source (rs1 and rs2) and destination (rd) registers at the same position in all formats to simplify decoding. Immediate values are packed towards the leftmost available bits in the instruction and have been allocated to reduce hardware complexity. In particular, the sign bit for all immediate values is always in bit 31 of the instruction to speed up sign-extension circuitry.

Rocket Core The implementation of the RISC-V architecture, the Rocket Core, implements the 32-bit instruction set which can handle both 32-bit and 64-bit operands. The CPU contains 31 registers. The core works with six pipeline stages:

1. pcgen - generates the program counter value
2. fetch - fetches the instruction
3. decode - decodes the instruction
4. execute - the execution phase
5. mem - access to memory
6. commit - commits the result

A detailed overview can be found in ([1], page 13). All basic extensions are implemented in the Rocket Core, so all functions of the ISA are available and a floating point unit is included.
Some functions need more than one cycle. A scoreboard takes care of data hazards that could occur while using functions like ”DIV”. All computations are done in user mode. The supervisor mode will be used for trap and interrupt handling.

[2]program counter

3.2 Approximate Arithmetic Logic Unit (ALU)

To attain scalable approximation, we designed a scalable ALU unit inside the pipeline, specifically in the decode/execution phase. The basic idea for our scalable ALU unit is to approximate operands rather than ALU function itself, before processing the operands. The main idea is that a number of LSBs[3] can be neglected if the application tolerates some specific error rate. This depends on required quality of results specified by the programmer. The approximation of operands takes place before the execution phase and after the decode phase.

Arithmetic operations are good candidates for approximation since they are more complex and consume much more energy than logical and shift operations. Our design supports approximate instructions for the ADD, SUB and MUL operations. As neglecting bits in a division operation can produce undefined results, division is left out. Arithmetic operations with immediate values are also supported by the core, but are mostly used for control flow and less in normal arithmetic computations. Adding approximate instructions for them will add to the complexity of the decode phase without great promise. More details on approximate instructions are shown in Section 3.2.2.

Scalable ALU The design in Figure 4 shows the circuitry 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. Approximation is done by neglecting number of LSBs from each operand and replacing them with zeros. The graph shows the modification to one of the ALU input operands. A control signal, id_precsie, is set to zero if the operation is approximate. id_app_bits are the bits responsible for defining the degree of approximation.

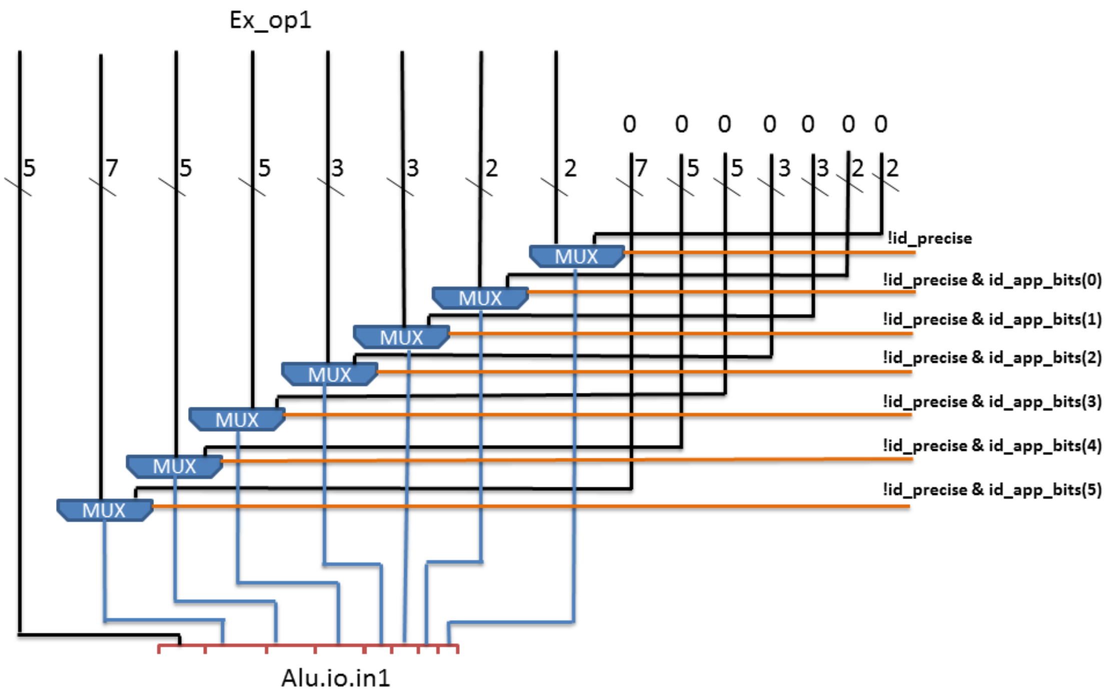
Table 1 gives information on the number of LSBs being ignored for each level of approximation and its corresponding encoding. The 7 encoding bits represent the 6 bits obtained from the instruction concatenated with the control bit (as the right-most). I.e. the control bit is always 0, which means the instruction is approximate. This design variant gives a maximum of 27 bits approximation and minimum of 2 bits.

From Table 1 it can be concluded that we never approximate last 5 bits (27 to 31) as they form the MSB of the number – a 100% approximation can be achieved by leaving out the instruction.

3.2.1 Control Parameters for the ALU

As already mentioned, the approximation logic is provided with 6 bits defining the approximation level. These must be present in the instruction format of every approximate

[3]The least significant bit is the lowest bit in a series of numbers in binary.



Ex_op1
0 0 0 0 0 0 0
5
7
5
5
3
3
2
2
7
5
5
3
3
2
2
MUX
!id_precise
!id_precise & id_app_bits(0)
!id_precise & id_app_bits(1)
!id_precise & id_app_bits(2)
!id_precise & id_app_bits(3)
!id_precise & id_app_bits(4)
!id_precise & id_app_bits(5)
Alu.io.in1

Figure 4: approximating input operands of ALU.

Encoding bits	No of bits approximated	
level 1	1111110	2 lsb neglected
level 2	1111100	4 lsb neglected
level 3	1111000	7 lsb neglected
level 4	1110000	10 lsb neglected
level 5	1100000	15 lsb neglected
level 6	1000000	20 lsb neglected
level 7	0000000	27 lsb neglected

Table 1: Encoding of approximation bits.

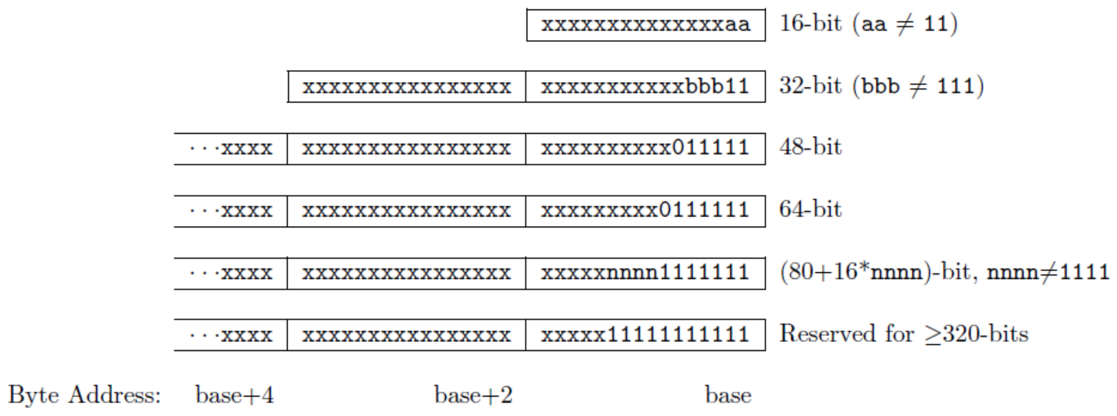


Figure 5: RISC-V instruction length encoding [2]

operation. They can be calculated if the programmer provides the compiler with suitable error bounds i.e the deviation a result can tolerate and the range in which the variable lies. As the ALU neglects selected blocks of bits, the amount of error equals $2^{n+1} - 1$ at maximum, for n neglected bits. The compiler also provides separate opcodes for approximate operations. These are used in the decode phase to produce the control bit.

3.2.2 Changes to the Instruction Set

The current ISA of RISC-V supports different instruction length encoding as you can see in Figure 5. Based on this, the proposed ISA of 32-bit instructions needs the first two bits to encode that it is a 32-bit instruction and the following five bits define the opcode. The opcode determines the instruction format as well as which operation is computed. In all instructions which use the ALU, a “funct3” input (bits 12-14) defines the exact instruction format. For some functions, the “funct3” field is not sufficient, for these the R-format provides an additional seven bit “funct7” (bits 25-31) field (“funct2” [bits 25-26] for R4-format). In all encoding variants, all bits are used to depletion. Therefore new instructions are added for the scalable ALU. In order to maintain consistency the R-type instruction format is used to define approximate instructions for our scalable ALU. Table 2 illustrates the new instructions. Three bits are not enough to encode all functions, so the fourth bit is obtained from the field “funct7” in R format. The rest of the “funct7” field is used to specify the approximation level. The new design for the ALU does approximation on 32-bit operands. Table 3 lists all functions which are considered for approximation. They are encoded in the “funct7” field.

XXXXXX	X	XXXXX	XXXXX	XXX	XXXXX	0001011
approx 31-26	funct4 25	rs2 24-20	rs1 19-15	funct3-1 14-12	rd 11-7	opcode 6-0

Table 2: New opcode for the scalable ALU

Function	Opcode (Funct4)
ADD.APPROX	0000
SUB.APPROX	0001
MUL.APPROX	0010
FADD.S.APPROX (Single precision Add floating point)	0100
FSUB.S.APPROX (Single precision Sub floating point)	0101
FMUL.S.APPROX (Single precision Mul floating point)	0110
FSQRT.S.APPROX (Single precision Sqrt floating point)	1100
FADD.D.APPROX (Double precision Add floating point)	1000
FSUB.D.APPROX (Double precision Sub floating point)	1001
FMUL.D.APPROX (Double precision Mul floating point)	1010
FSQRT.D.APPROX (Double precision Sqrt floating point)	1101

Table 3: Function opcodes for the ALU

3.3 Lookup Table-based Approximation Unit (LUT)

3.3.1 Proposed HW Unit

The LUT unit is used to approximate arithmetic functions. The input range of a function (domain) will be divided into segments. The value of the function will be approximated for each segment either as a constant value or an affine function (defined by slope and base).

The LUT unit supports integer and fixed-point operations. For the current design only integers are considered. Extending the unit to process floating-point values will need extra hardware. This is left to later phases of our work. The LUT unit consists of three reconfigurable logic planes and a small ROM. An overview of the LUT unit architecture can be seen in Figure 6.

The ROM will store the approximated values of the arithmetic function for different segments of the domain. As a starting point for our design, the memory is addressed with eight bits, has 256 entries and implemented in SRAM technology. It has one port for reading and writing.

In order to achieve approximation of arithmetic functions with piecewise affine functions, two values, slope and offset, are stored for each segment. The slope is multiplied by the input value and then the offset is added. This will need extra hardware; namely two ROMs to store slope and offset values, a multiplier and an adder. The Architecture of the component is shown in Figure 7.

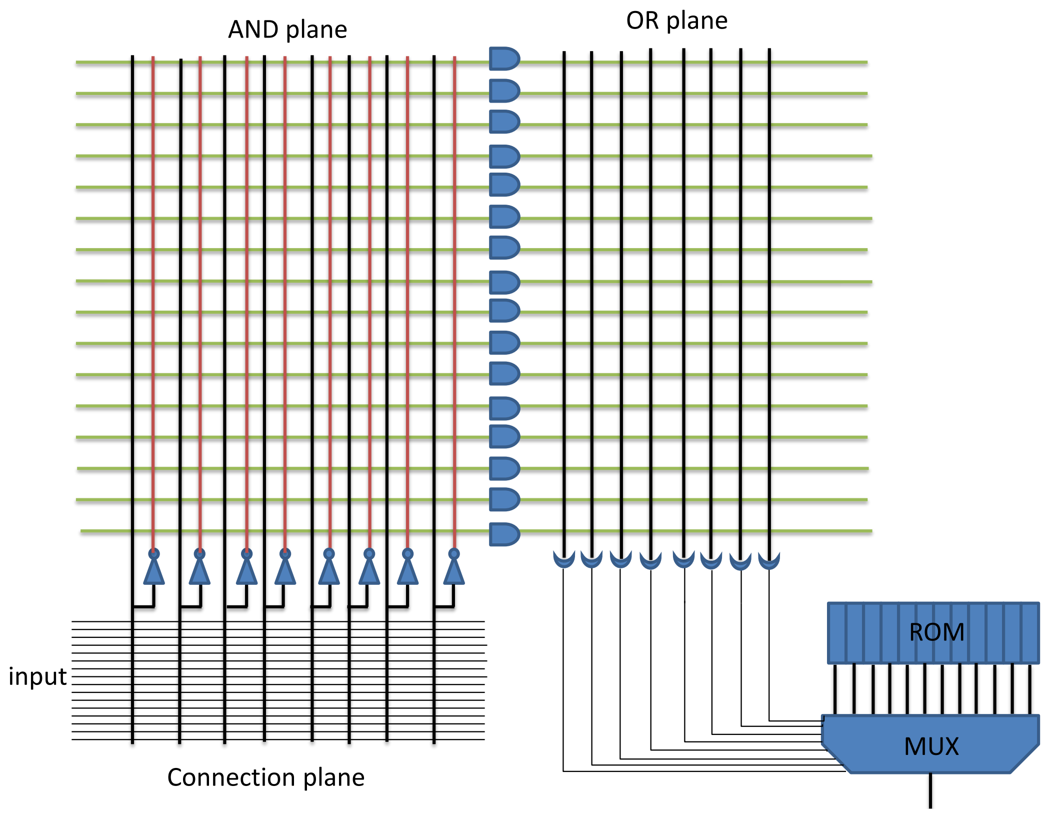


Figure 6: Architecture overview of the LUT unit

As Figure 6 shows, the signals from the OR-plane are used to address the ROM. This means only eight OR gates are needed. Ignoring optimization effects, the number of AND gates corresponds to the number of segments where the arithmetic function is sampled. For our current design, the AND-plane has eight inputs, along with their inverted values, and 256 gates. To select the eight relevant bits from the 32-bit inputs, the selection-plane is used.

The states of the cross points, connected or disconnected, in the three reconfigurable planes have to be stored in some memory structure. As this memory will not be accessed by the CPU memory address scheme, a simple structure of shift registers as shown in Figure 8 is sufficient. Data is copied to the registers using a simple memory interface. If the "write enable" signal is enabled, new data is written to the upper register, and older data is shifted to the next register. The connection-plane, AND-plane and OR-plane have 32x8, 16x256 and 8x256 cross points respectively. These 6400 values are generated by the compiler and loaded to main memory at program load time.

After implementing the LUT unit, the time delay of the complete unit will be estimated to decide if the lookup operation can finish in one cycle. If not, the lookup can be implemented as a multi-cycle operation and the CPU pipeline will be stalled.

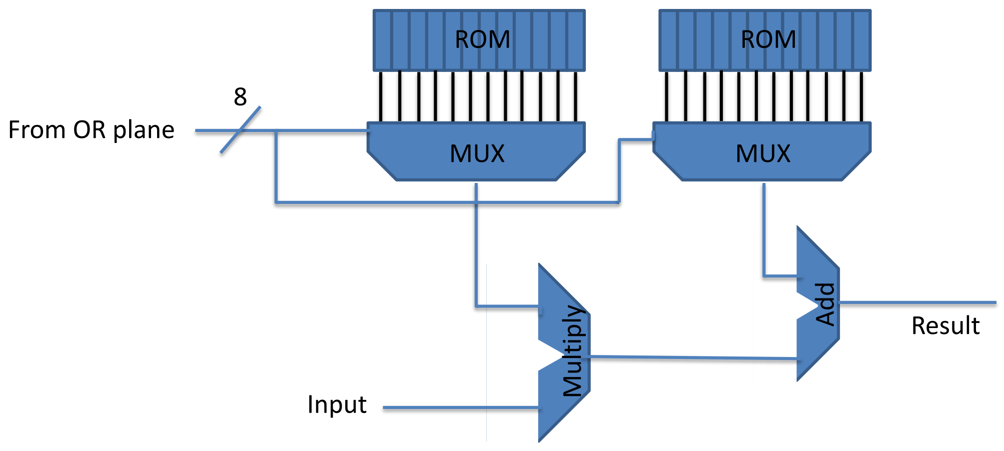


Figure 7: Using 2 ROMs to store values of slope and offset in one segment. 32 MUXs are used for 32 bit values stored in ROM, only one is shown here.

3.3.2 Compiler Interface

For the configuration of the LUT, the configuration data is generated by the compiler and loaded into main memory at program start time. It is later copied to the LUT in corresponding fields, the ROM and the shift registers, probably using DMA. This data contains 256 approximated values of the original arithmetic function. These values are to be stored in LUT memory. The configuration data also contains a bit stream representing the states of the cross points in the three reconfigurable planes. This is then copied into the corresponding shift registers shown in Figure 8.

3.3.3 Changes to CPU Core

The LUT is part of the execution stage of the pipeline. Its input can handle 32-bit operands (may be extended to 64-bit) from a register. But not all input bits from the register can be used as AND plane input. The input is limited by the size of the LUT reconfigurable planes. The output contains a 32-bit (or, extended, 64-bit) operand. Depending on the size of the LUT (the proposed values are only starting points for the

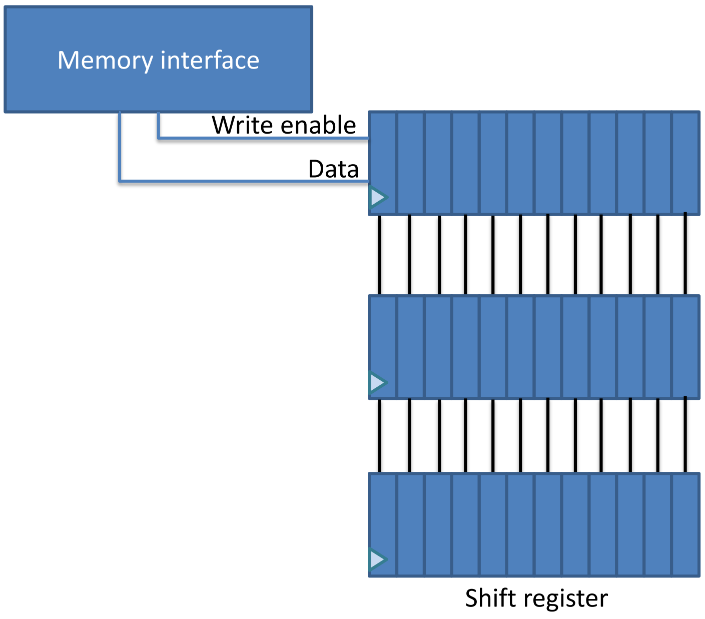


Figure 8: Memory interface to the shift register storing cross points states

implementation), it will take more than one cycle to look up the result. An acceptable trade-off of size, accuracy and space has to be evaluated during implementation. If the lookup takes more than one cycle, the processor has to use stalls. The LUT will be used for complex functions, so potential for improvement remains, compared to the computation the processor would do without using the LUT.
All LUTs will be programmed by loading the values from the memory by initialization of the LUT at program load time. Execution on the LUT has to wait until the LUT is configured. Initialization time also depends on the size of the LUT. If easily achievable with the Rocket SoC, the data is loaded with DMA, to prevent the CPU from blocking during configuration time.

3.3.4 Changes to the Instruction Set

The LUT unit expects four instructions to be supported by the processor.
Each of the instructions operates on a single LUT hardware core and thus has a selection field specifying which LUT core to use.
LUTL (table 4) performs a reset or configuration operation, transmitting a single word of configuration data.

LUTS (table 5) queries the status of a LUT hardware core writing the result into a destination register.
LUTE (table 7) starts a computation on a LUT hardware core writing the result into a destination register. It accepts one or two input registers.
LUTW (table 9) performs identically to LUTE, however no computation is performed.

Table 4: LUTL

`lutl rs1, lutsel, 0, reset/conf`

00	000	XX	XXXXX	XXXXX	XXX	XXXX	X	0101011
func2	reserved	lutsel[4:3]	reserved	rs1	lutsel[2:0]	reserved	reset/conf.	opcode
31-30	29-27	26-25	24-20	19-15	14-12	11-8	7	6-0

“funct2” selects the function, lutsel selects which LUT to configure, “rs1” is the source register for the LUT programming (ignored if reset/conf is set to 1), “reset, conf.” is to indicate whether to reset or configure the LUT, resetting the LUT core if set to 1 and loading a configuration word otherwise.

Table 5: LUTS

`luts rd, lutsel`

10	000	XX	XXXXXXXXXX	XXX	XXXXX	0101011
func2	res	lutsel[4:3]	reserved	lutsel[2:0]	rd	opcode
31-30	29-27	26-25	24-15	14-12	11-7	6-0

“funct2” selects the function, lutsel selects which LUT to retrieve the status from and “rd“ indexes the register to store the status in.

The instructions are encoded to fit into the R-format of the RISC-V ISA like most functions in the execution pipeline stage of the Rocket Core do. It also reduces the size of the decoder to fit into one instruction format. Since there is a little difference to the R-format we name this format the A (for approximate) format. If configuration of the LUT with the DMA works fine, the first instruction will only be a pseudo-instruction. The size of LUT units is subject to the trade-off between energy consumption and level of approximation (number of segments). Depending on this, it will be possible to implement more or less LUTs. If the LUT is configured by DMA can be used to address more LUT units. This has to be evaluated during the implementation.

Table 6: LUTE

```
lute rd, rs1, lutsel, strange
lute2 rd, rs1, rs2, lutsel, strange
```

01	X	XX	XX	XXXXX	XXXXX	XXX	XXXXX	0101011
func2	charm	strange	lutsel[4:3]	rs2	rs1	lutsel[2:0]	rd	opcode
31-30	29	28-27	26-25	24-20	19-15	14-12	11-7	6-0

Table 7: "funct2" and "opcode" identify the operation, "lutsel" selects which LUT to use, "rs1" and "rs2" are source registers for LUT input and, "rd" is the destination register. "charm" determines if two source registers are used (1) or just "rs1" (0). "strange" is used to determine what LUT inputs are supplied with "rs1" and "rs2". The following table shows how the three input words of the LUT core are affected by this instruction. "keep" signifies that the previously used word is kept as input.
The "charm" bit is not part of the instruction's mnemonic and is instead encoded in its name: use "lute" for 0 and "lute2" for 1.

charm	strange	input 1	input 2	input 3
0	00	rs1	keep	keep
0	01	keep	rs1	keep
0	10	keep	keep	rs1
0	11	keep	keep	keep
1	00	rs1	rs2	keep
1	01	keep	rs1	rs2
1	10	rs2	keep	rs1
1	11	keep	keep	keep

3.4 Testing

Hardware Approximate ALU and LUT units must be tested to check if they function properly and produce the expected results. The approximate results obtained are then compared to the accurate results to check if error margins specified by the programmer are violated.

After implementing an ALU approximation technique, the design can be tested with different operations, different operands and different values of neglect-amount. Test data will be prepared manually at first. It does not have to wait for or depend on the tool-chain. Scripts will automate feeding of data to the implemented hardware, collect results, compare to accurate results and return a human-readable summary to the hardware developer.

The same applies for the LUT unit. Values stored in ROM and states of connection-

Table 8: LUTW

```
lutw rs1, lutsel, strange
lutw2 rs1, rs2, lutsel, strange
```

11	X	XX	XX	XXXXX	XXXXX	XXX	00000	0101011
func2	charm	strange	lutsel[4:3]	rs2	rs1	lutsel[2:0]	res	opcode
31-30	29	28-27	26-25	24-20	19-15	14-12	11-7	6-0

Table 9: "funct2" and "opcode" identify the operation, "lutsel" selects which LUT to use, "rs1" and "rs2" are source registers for LUT input. "charm" determines if two source registers are used (1) or just "rs1" (0). "strange" is used to determine what LUT inputs are supplied with "rs1" and "rs2". The following table shows how the three input words of the LUT core are affected by this instruction. "keep" signifies that the previously used word is kept as input. The "charm" bit is not part of the instruction's mnemonic and is instead encoded in its name: use "lutw" for 0 and "lutw2" for 1.

charm	strange	input 1	input 2	input 3
0	00	rs1	keep	keep
0	01	keep	rs1	keep
0	10	keep	keep	rs1
0	11	keep	keep	keep
1	00	rs1	rs2	keep
1	01	keep	rs1	rs2
1	10	rs2	keep	rs1
1	11	keep	keep	keep

Table 10: LUTE3

```
lute3 rd, rs1, rs2, rs3, lutsel
```

"LUTE3" is intended as a pseudo-instruction, to be translated with:

```
lutw2 rs1, rs2, lutsel, 0
lute rd, rs3, lutsel, 2
```

plane cross points are generated manually for certain arithmetic functions. Then different lookup operations can be performed to check if the hardware design functions properly.

In a later phase of our work when the complete tool-chain is developed, standard test benches will be used to measure the degradation of precision corresponding to different levels of approximation.

Applications For the applications we want to run on PACO, we will create a QEMU emulation of our system so we can quickly check if the application is processed correctly when approximate instructions are binary-translated to a precise analogue of their functionality. This will speed up development and bug-hunting since it enables us to provide error-free test cases for the hardware with higher probability.

4 Compiler/Language

The PACO language/compiler needs to offer programmers access to approximate computation units via an interface.

To this end we add annotations specifying types and functions as approximable to the CLANG C/C++ frontend. These annotations also specify additional parameters used to determine the configuration of applicable approximation techniques.

On the backend side of the compiler, binaries will be created, integrating additional sections for Lookup Table configuration data as well our new approximate instructions.

4.1 Programmer's Interface

For an approximate interpretation of the source code, a developer needs to annotate types or whole sections of code as approximate. The design of these annotations is the topic of this section. In general, we introduce type annotations and pragma commands, extending the CLANG C/C++ LLVM frontend.

Type annotations are used to derive characteristics of fine-grained approximation techniques such as the approximated ALU implemented in our core. Annotating functions/invocations serves to mark entire sections of code as approximable which by default is translated to a Lookup Table approximation unit.

The language extension itself is designed with extendability in mind: All annotations follow the same syntax using a construct called the *approximation decorator*, or just *decorator* for short. Such a decorator is a list of named arguments called *key-value* pairs or simply *key-values* that specify method and degree of approximation to be used.
Any later additions to the set of approximation units may use the same decorator and simply change the set of values it contains.

4.1.1 Predefined Preprocessor Macros

The extended CLANG compiler defines the macro `__paco` in order to allow distinguishing between PACO-aware and PACO-unaware compilers and thus write code compilable by both kinds of compilers.

4.1.2 Pragma Directives

Additional configuration can be done to set up compiler behavior. This is done through pragma statements:

- `#pragma paco combine <mode>`: When an operation is performed on operands with different levels of precision, a resulting precision must be derived to configure the underlying approximate functional unit and to infer the result type.
  The method to be used in such a scenario can be set with this pragma instruction. Permissible values for `mode` are:

  - `least_precise` Combines all operands' precision levels so that the result will be no more precise than any of the operands. If for example the first operand is precise in any but the least four bits and a second operand is imprecise in exactly the seventh-least significant bit, the result will be imprecise in the four least significant bits as well as the seventh-least significant bit.
    In terms of ALU instructions, the precision masks of the operands will be *and*-combined (see Table 1).
  - `most_precise` Combines all operands' precision levels so that the result will no less precise than any of the operands. If for instance the first operand is imprecise in four least significant bits and the second one is imprecise in two, the result will be imprecise in only the two least significant bits.
    In terms of ALU instructions, the precision masks of the operands will be *or*-combined (see Table 1).
  - `error` Operation on inputs of differing precision is considered an error.

  The value of this pragma defaults to `error`.

- `#pragma paco intermediate_literal <mode>` Sets what precision a literal should have that occurs alongside a decorated type as input to an arithmetic expression. This scenario occurs on binary operators and the ternary comparison operator.
  For binary operators the imprecise type to mimic in a literal is well-defined as there is only one other operand. For the ternary comparison operator, the first operand holds a special purpose as the selector thus the type to mimic is also well-defined.
  Possible values for `mode` are:
  - `precise` All literals are of their respective precise type. This is the default setting.
  - `mimic` A literal assumes the same type as an imprecise type occurring along with it.

4.1.3 Decorators

Besides selecting modes of operation through pragma directives as discussed in the previous section, the main way to exploit approximate functional units is use of *decorators.* A decorator is a set of key-value pairs attached to a type / function definition or a function invocation.
The available keys and the acceptable type and meaning of assigned values depend on the approximation unit to be used and will be explained further in the appropriate sections for instruction-level approximation using the approximate ALU (Arithmetic operation approximation) and function-level approximation using Lookup tables (Function approximation).

Approximation decorator To mark a syntax element of the source code as approximable and specify parameters on its approximation, an *approximation decorator* is used. This is a set of key-value pairs enclosed in `approx()` . The value assigned to a key must be a literal:

```
approx_decorator ::= ( 'approx' '(' approx_keyvalue+ ')' )+
approx_keyvalue ::= ident '=' literal
```

Example:

```
approx( neglect_amount=2 inject=1 )
```

(Note that the key-value pairs in this example do not reflect actual key-values used, but serve to illustrate the syntax only.)

The set of available keys and the expected types depends on the context in which the decorator is used. If multiple approximation decorators are specified, they are evaluated in order of appearance. This means that any key-value specified in a preceding decorator is superseded by values assigned to the same key in any succeeding decorator.

The syntax of the approximation decorator has been chosen explicitly to allow annotated code to be compiled with any C compiler, ignoring approximation annotations. To do this, a simple macro resolves all approximation decorators to nothing:

```
#ifndef __paco
#define approx(...)
#endif
```

Approximate types An approximate type is any simple data type decorated with an approximation decorator. It is used to specify approximation parameters for individual variables that is used to derive approximable operations.

```
approx_type ::= type_declaration approx_decorator
```

Example:

```
typedef int   approx( neglect_amount=2 inject=1 ) int2;
typedef float approx( neglect_amount=4 ) float4;
typedef int4  approx( neglect_amount=7 inject=0 relax=1) int7;
```

If the base type is already approximate, any additionally specified keys supersede those defined in the original type definition. In the above example, `int8` keeps the key-value `neglect_amount=4` from its `int4` base type but overrides `inject=1` to `inject=0`.

Approximate functions To utilize coarse-grained approximation facilities such as Lookup Tables or Neural Processing Units, entire blocks of code must be evaluated at once. This can be achieved by decorating a function definition, thereby applying approximation to the entire function body. Approximate functions are defined by appending an approximation decorator after the argument list. Example:

```
float function1(float a) approx( strategy="lut" ) {
  return sqrt(a);
}
```

Approximate invocations If only a specific invocation of a function is to be approximated or it is to be approximated *differently* than it was specified in the function definition, the invocation itself can be decorated with an approximation decorator, too. The effect is identical to copying the called function definition and applying the additional decorator to the duplicate function definition.
Decorating a function invocation follows the following syntax:

```
approx_invoc ::= identifier approx_decorator '(' argument_list ')'
```

Example:

```
float a=approx( strategy="lut" ) sqrt(b);
```

4.1.4 Arithmetic Operation Approximation

One approximation technique of the PACO core is an approximate ALU which allows to neglect a number of bits of the operands. Neglecting bits in this case leaves the output of the operation in those bits as undefined. Usage of this is determined by the types of inputs and the assignment target of an arithmetic operation.
Literal inputs are considered precise while variable inputs may have decorated types. Operation approximation understands these key-values:

- `neglect_amount`: An integer specifying the number of bits to neglect. Valid numbers are: 2, 4, 7, 10, 15, 20 and 27. For further information, refer to Table 1. Note that this cannot be used in conjunction with `mask` (see below).
- `mask`: A 6 bit integer encoding the bit neglect mask according to Table 1. Note that in contrast to using `neglect_amount`, this way neglected bits can be selected more freely allowing intermediate bits to be neglected and less significant ones to be left precise. Note that this cannot be used in conjunction with `neglect_amount`.
- `inject`: If set to a non-zero integer, this variable is considered to be imprecise thus operations having this as an input can operate approximately. This is used to denote that the source of data found in variables of this type are expected not to deliver exact values thus approximation can be performed even if the result is to be handled as precise.
- `relax`: If set to a non-zero integer (default), operations assigning to a variable of this type can be regarded as approximable. This is used to denote that the result of an operation does not need to be exact thus even with precise inputs approximation is performed.

Considering an expression tree involving decorated types as inputs and as assignment target, translation occurs as follows:
First, any `neglect_amount` specification is translated into a neglect `mask` according to Table 1.

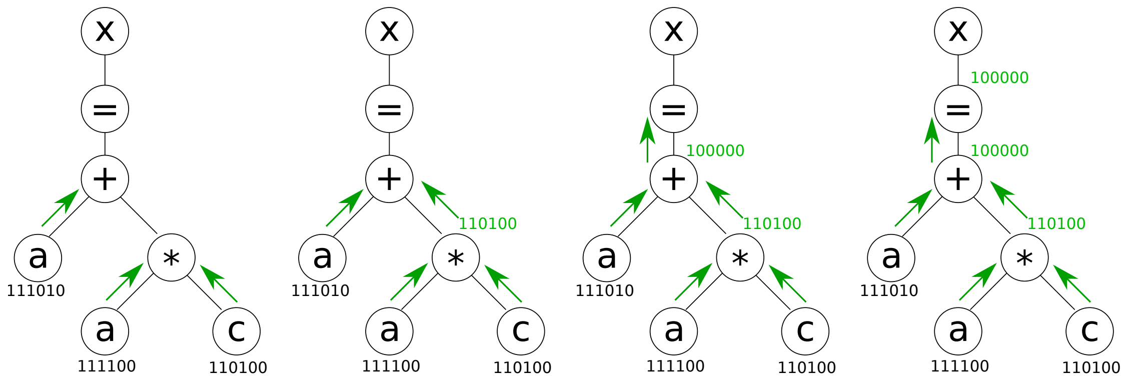
If the assignment target is annotated with the `relax` key-value, its neglect mask is applied to the root operation of the expression tree and propagates downwards towards the leaves of the expression tree.
Together with this back-propagated neglect, a forward-propagated neglect is computed for each node of the expression tree by combining the neglect masks of the immediate inputs starting at the leafs.
If both operands have an identical neglect mask, it is applied to the operation as is.
If the neglect masks are different, they are combined according to the `combine` pragma.
A literal assumes a neglect mask according to the `intermediate_literal` pragma.
An illustration of forward and backward propagation can be seen in Figure 9 for an expression tree of $x = a + (b \cdot c)$.
After both the forward and backward propagated neglect values are computed, they are combined by picking the least precise of the two. This decision is sound as can be seen by the following elaboration:
If the forward-propagated neglect allows more precision than is required by the back-propagated neglect, the additional precision is unnecessary by definition.
If the forward-propagated values are less precise than the back-propagated neglect accepts, additional precision in the operations yields random data as the input was already random.

For each operation that can be approximated, error propagation must be computed. This is a function that dictates the neglect mask of the result type of this operation depending on the neglect masks of the operation itself. Alongside this, the inverse function needs to be computed as well in order to allow back-propagation to compute what neglect mask the operation can use if the output needs to match a given neglect mask.

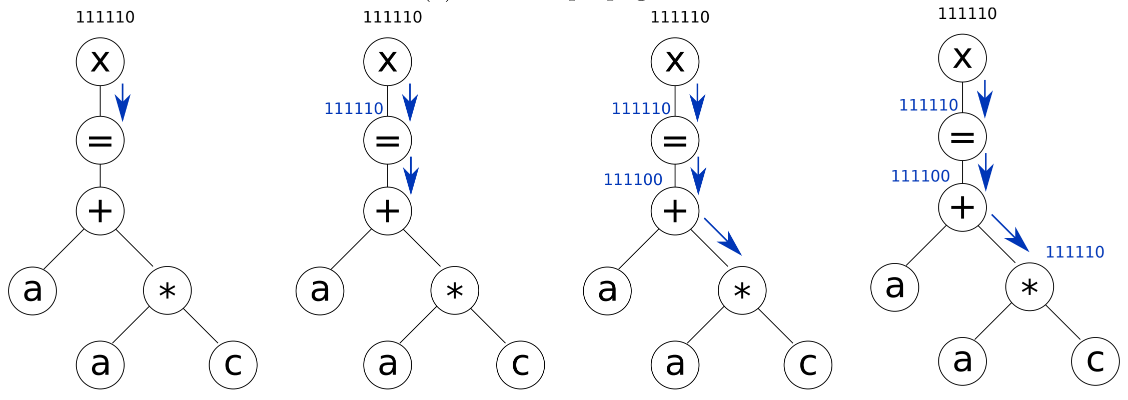
4.1.5 Function Approximation

LUT approximation only promises speed-up and reduction of energy usage for more complex functions. Therefore it is applied on blocks of code rather than single instructions. These blocks of code are identified as functions with no side effects, annotated with an approximation decorator. Each invocation of this annotated function is then replaced with an invocation of a corresponding LUT.
To allow for different approximation parameters on a single function definition, the invocation of a function may also be decorated with an approximation decorator, resulting in a combined set of key-values from the decorator used on the annotated function and the decorator used on the invocation.
Note that having multiple invocations with different sets of key-values yields multiple LUT configurations.

These configurations are derived by extracting the annotated / invoked function source code as well as the set of key-values and invoking an external LUT Configuration Generator with these. This Generator then builds a configuration bitstream for the corre-



(a) Forward propagation



(b) Backpropagation

Figure 9: Exemplary neglect propagation for an expression tree of $x = a + (b \cdot c)$. Node annotations represent the neglect mask as it could be specified via the `mask` key-value, actual numbers are purely exemplary.

sponding LUT which is combined with regular code in the linking stage of compilation. For further information refer to Section 4.4.

4.2 Compiler Frontend

A compiler toolchain is divided into the following stages:

4.2.1 Lexical/Syntactical Analysis

The lexical and syntactical analysis stages consist of reading the source code and deriving a syntax tree from it. This tree is used throughout the following compilation stages to analyze the code and finally find an intermediate representation used to generate binaries.
For our modifications we need to add grammar rules and syntax tree components to accommodate approximation decorators.

4.2.2 Semantic Analysis

This stage interprets data that was structured in the syntactical analysis phase. As part of this, identifiers are resolved and the types of variables and intermediate expressions are resolved.
Modifications in this phase are the following:

1. Definition of unrecognized key-values should yield a warning or an error, configurable through command-line flags to the compiler frontend.
2. Assigning an unexpected type to a key-value has to be reported as an error.
3. Backpropagation of neglect in expression trees must be performed according to the selected backpropagation mode.
4. All Lookup Table based approximations must be listed by enumerating different approximation configurations in invocations of approximable functions.

These invocations are then provided with unique identifiers and passed to the LUT Configuration Generator using these identifiers as their names.

4.2.3 Translation

In this stage the analyzed syntax tree is translated from its source language-specific language to the LLVM intermediate language. During this step, function invocation nodes calling approximated functions are replaced with LUT invocation nodes.
To add instructions to LLVM IR (Intermediate Representation) many compiler files need modifications. These files are not discussed here any further, as it is too much implementation detail. A step by step introduction can be found in LLVM documentation [4].

[4] `http://llvm.org/docs/ExtendingLLVM.html`

4.2.4 Code Generation

In this stage the translated syntax tree is translated into LLVM IR for the LLVM backend to work on and apply optimizations.
In addition to generating code for our custom instructions we also need to invoke the LUT Configuration Generator to find their configuration to be used in the final executable.
Table 11 shows the PACO extension to the LLVM IR. The first part represents instructions intrinsics corresponding to approximate ALU instructions as defined in Table 3 and the second part represents LUT-related instructions as defined in Tables 4 and 7. Instruction intrinsics are functions in LLVM IR which always start with the prefix "llvm." and can be replaced by a backend which recognizes them with custom instructions. Every other backend simply generates a function call.

Instruction	Description
llvm.riscv.add.approx(rs, rt, rd, nm)	Addition of rs and rt, with neglecting bits according to the neglecting mask (nm) and storing the result in rd
llvm.riscv.sub.approx(rs, rt, rd, nm)	Subtraction of rs by rt, with neglecting bits according to nm and storing the result in rd
llvm.riscv.mul.approx(rs, rt, rd, nm)	Multiplication of rs and rt, with neglecting bits according to the neglecting mask (nm) and storing the result in rd
llvm.riscv.fadd.s.approx(rs,rt,rd,nm)	Single-precision floating-point addition
llvm.riscv.fsub.s.approx(rs,rt,rd,nm)	Single-precision floating-point subtraction
llvm.riscv.fmul.s.approx(rs,rt,rd,nm)	Single-precision floating-point multiplication
llvm.riscv.fsqrt.s.approx(rs,rt,rd,nm)	Single-precision floating-point square root
llvm.riscv.fadd.d.approx(rs,rt,rd,nm)	Double-precision floating-point addition
llvm.riscv.fsub.d.approx(rs,rt,rd,nm)	Double-precision floating-point subtraction
llvm.riscv.fmul.d.approx(rs,rt,rd,nm)	Double-precision floating-point multiplication
llvm.riscv.fsqrt.d.approx(rs,rt,rd,nm)	Double-precision floating-point square root
llvm.riscv.lutl(rs,li,roc)	Transmit a single configuration ($roc = 1$) word to LUT core li or reset it ($roc = 0$).
llvm.riscv.luts(rd,li)	Get the status word of LUT core li and store it at rd.
llvm.riscv.lute(rs,rd,li,strange)	Execute a single computation on LUT core li, using rs as input and rd as output. *Strange* selects which internal input register to use.
llvm.riscv.lute(rs,rt,ru,rd,li)	Execute a single computation on LUT core li, using rs, rt, ru as inputs and rd as output.

Table 11: PACO extension to LLVM IR

4.3 Compiler Backend

4.3.1 Assembly Code Generation

After LLVM has finished optimizations, machine-specific assembly code is generated from LLVM IR using intriniscs. In this step the back-end of LLVM will translate the LLVM IR to those understood by the assembler. Our part consists of extending this translation process with the mnemonics described in Table 11. In LLVM the folders AsmParser and Bitcode (both contained in lib) are relevant for our modifications in reference to the machine-specific assembly code. This link [5] gives a general overview of the directory layout of LLVM.

4.3.2 Assembly

Starting from machine-specific assembly code, the assembler builds object code in a binary format specific to our architecture. Each of our mnemonics of Table 11 belongs to one opcode specification of Table 3. The following examples shows how the LLVM IR mnemonics for the approximate addition and subtraction function are correlated to their opcode specifications:

llvm.riscv.add.approx(rs, rt, rd, nm) → ADD.APPROX (0000)
llvm.riscv.sub.approx(rs, rt, rd, nm) → SUB.APPROX (0001)

4.3.3 Linking

Finally, all assembled object files are linked together to a single executable.
We need to configure the linker to do two things:
First it needs to know about addresses used in LUT configuration loading in order to translate them. Secondly, LUT configuration data must be included in appropriate sections. This is done by encoding the LUT configuration data as ELF, thereby enabling any linker to combine them into the final executable.
At the linking stage, LUT hardware cores are also assigned: Up to the linking stage, the toolchain does not know how many LUT configurations are used throughout the entire program. Thus the job of the linker is to resolve symbolic LUT names in load and invocation instructions and issue LUT hardware indices accordingly. (If more LUT configurations are present than there are LUT hardware cores, a linker error must be generated.)

4.4 LUT Configuration Generator

The process of deriving LUT configuration data from an annotated function or invocation is handled by an external program which is fed the source code of the function in

[5]`http://llvm.org/docs/GettingStarted.html#general-layout`

question and parameters in the form of key-values.

The program operates in two stages: First the given key-values and source code are evaluated and a LUT specification is generated from them as a piecewise affine function. In the second step such a piecewise affine function is read and compiled into a LUT bitstream ready for configuration of a LUT hardware unit.

LUT Configuration Name To properly link the final executable, each individual LUT configuration is assigned a *globally* unique identifier. This is an internal value determined by the compiler and passed to the LUT Configuration Generator. It can be provided via command-line option, as key-value `name` or a `name` directive in the intermediate format (see below).

4.4.1 Input Format

Combining key-values and source code in one format, the following syntax is used as an interchange format between the main compiler frontend (CLANG) and the LUT Configuration Generator.
The file format is text-based with UTF-8 encoding, split into three blocks: Key-values, target signature and code.
The key-value section contains one key-value pair per line, separated with equal signs. The key and value parts are the same syntactical constructs as they are in the key-value definition in the C/C++ code.
A single line containing %% seperates the three sections from one another, just like it is done in flex input files.
The second block contains only the signature of the target function in the form `identifier argument_type -> result_type`.
Finally the third block contains one or many function definitions, one of which having the same identifier as specified in the signature section which will be evaluated for approximation. This allows for functions to be approximated that invoke other functions defined in source code.

4.4.2 Intermediate Format

In order to intervene in the process of compiling a LUT configuration, an intermediate format can be used as input and output of the LUT compilation program.
In this format, the LUT is specified as a list of segments. It is a line-based text file in UTF-8 encoding accepting double slashes and hashtags as initiators of line comments. Any non-comment line may contain a single directive, of which there are three: Name directives, domain directives and segment directives:

1. Name directives are required to find the name of the current compilation unit and they are provided as

   ```
   name <name>
   ```

With `<name>` being a string literal enclosed in double quotes. Specification of zero or more than one name directives is considered an input error.

2. Domain specifications give the width of the approximated domain and its offset. The width is expressed as an exponent of 2:

```
domain <exponent> <offset>
```

3. Interval specifications are of the form:

```
interval <start> <end> <val1> <val2>
```

Such an interval specification maps the interval $[start, end]$ to the function

$$x \mapsto \frac{x - start}{end - start} * (val2 - val1) + val1$$

As an intermediate specification is the representation of a Lookup table and by that a function, no input value must be mapped to more than one output value: The set of intervals defined in an intermediate specification must be disjoint.

4.4.3 Weights Format

The weights file is a line-based text file in UTF-8 encoding accepting double slashes and hashtags as line comment initiators.
Each non-comment line is of the form `range = expression` mapping weights to one or more input values.
A *range* is either a number of an interval specified as a tuple:

```
range ::= number | '(' number ',' number ')'
```

If the interval is empty (the second bound being less than the first bound), it is never used but generates no error.
An expression is either a number or an expression understood by a Lua interpreter. In addition to number literals it offers the following functions and constants:

- `x` (The input itself)
- `abs(x)`
- `e`, `pi`
- `sin(x)`,`cos(x)`,`tan(x)`

- `sinh(x)`,`cosh(x)`,`tanh(x)`
- `sqrt(x)`,`pow(base,exponent)`, `exp(x)`
- `floor(x)`,`ceil(x)`,`round(x)`

The weight specifications are read in order of occurrence in the weights file. If a multiple weights are specified for the same value, the last one is used. This is particularly useful for specifying weights over an interval with a few exceptions.

4.4.4 Output Format

The final data produced by the LUT Configuration Generator is the bitstream needed for configuring a single LUT hardware unit.
This data can be issued either as an ELF or C code The only section contained in either format is a static data section containing the raw binary data (connection point configuration and ROM data).
This raw data stream is labeled with the name of the LUT as specified in its input or intermediate format.

4.4.5 Architecture Specification Format

In order to configure the LUT generation tool for different LUT architectures without recompiling it, a single architecture specification file can be provided that overrides hard-coded architecture configuration.
It uses the same format as the intermediate file without any code segments.
Key-value pairs available for overriding depend on the implementation of the LUT Configuration Generator, however no compiler error must be generated for missing or unrecognized key-values. If a key-value is not present, it shall be left at the hard-coded default value. An unrecognized key-value shall generate a warning.

4.4.6 Approximating Strategy

Approximating the input function is done in two steps: First the input domain is split into segments according to a *segmentation strategy*, then each of these segments are assigned a piecewise affine function through the *approximation strategy*.
The input domain is derived from the data types of the input of the approximated function. However, it can be overridden with the `bounds` key-value, which is a comma-separated list of intervals $[min, max]$, $(min, max]$, $[min, max)$ or (min, max) of boundaries.
Furthermore a *weights* file may be used to weight input values. This can be seen as a grading of importance (non-positive weights representing exclusion) of input ranges and are used by some of the segmentation and approximation strategies.
The number of segments to be used by the LUT defaults to the maximum number available in the LUT hardware, however this may be overridden by the `num-segments` key-value.

Utilizing these pieces of information, seven segmentation strategies are available which are selected with the `segments` key-value:

uniform . This strategy partitions the domain into segments of equal length ignoring any weights that might have been specified.

log-left Subdivide the domain into a sequence of segments that grow exponentially in width from the lower end of the domain to the upper end.

log-right Subdivide the domain into a sequence of segments that grow exponentially in width from the upper end of the domain down to the lower end.

min-error . This strategy joins or splits segments in order to achieve the correct number of segments representable by the LUT HW core. It does so by greedily picking an operation exhibiting least error.

explicit segments . If the segments to be used are known beforehand, the programmer can specify them as a comma-separated list of intervals $[min, max]$, $(min, max]$,$[min, max)$ or (min, max), similar to the specification of the input domain.

As an experimental feature, two-stage segmentation can be selected by using a second keyvalue, named `segments2` which specifies another segmentation strategy to used for subdividing segments found in the first stage.
Note that for using two-stage segmentation, the `min-error` and `min-error-weighted` segmentation strategies cannot be used for the first stage and no explicit segments must be specified.
If two-stage segmentation is used, the `num-primary-segments` key-value has to be used to specify how many segments shall be generated in the first stage. The total number of segments generated adheres to the `num-segments` key-value.

Using segment boundaries, three approximation strategies are available, which are selected with the `approximation` key-value:

interpolated . Simply connect the function values at each lower and upper segment bound with a straight line.

linear . Minimize the mean square deviation of the line approximation.

step . Minimize the mean square deviation of a constant approximation.
Note that error-minimizing strategies, both in segmentation and approximation stages, are particularly time-consuming as they are performed numerically.

4.4.7 Keyvalue Summary

As the LUT Configuration Generator is modular, specifying any key-value that is not recognized must not cause a compiler error (but may be reported as an info or warning message).
Key-values understood by the LUT Configuration Generator are:

- `name`: This must be specified by the compiler and should generate a compiler error if the programmer sets this key-value in the source code. The name itself must be a globally unique symbol identifier different from any other symbol identifier exported in any other binary produced in the compilation process.
- `numSegments`: An integer specifying the number of segments to use in the LUT. It defaults to the maximum number of intervals expressible in the target architecture.
- `segments`: Strategy to use for partitioning the input domain into intervals.
- `explicitSegments`: List of intervals making up an explicit segmentation. Cannot be used with `segments`.
- `segments2`: To use two-stage segmentation, another segmentation strategy can be specified in this key-value.
- `numPrimarySegments`: Sets the number of segments to be generated in the first stage of two-stage segmentation. If only one segmentation stage is used, this key-value is ignored.
- `weights`: Name of a file to be used as weights of input values.
- `approximation`: Strategy to be used to obtain affine functions for each segment.
- `bounds`: List of intervals comprising the input domain.

4.4.8 Command-line Options

To control the behavior of the LUT Configuration Generator, a number of arguments can be specified on the command line preceding a single input file name:

- `-i` or `--intermediate`: Output an intermediate representation of the generated LUT instead of building its bitstream.
- `-a` or `--assembly`: Output an assembly file (`.s`) of the generated bitstream instead of an ELF object file.
- `-c` or `--compile`: The input is expected to be in intermediate format instead of source input.
- `--arch <file>`: Specify a file to read architecture-specific parameters from it and use them to override the default parameters hard-coded into the program.

- `-n <name>` or `--name <name>`: Specify the name of the LUT. This overrides any name specified in the input. If this is specified, a missing name in the input will not generate a compiler error.
- `-o <name>` or `--output <name>`: Override the output file name to be generated. By default the output file is `<id>.lut` for intermediate files, `<id>.s` for assembly output and `<id>.o` for ELF object files, `<id>` being the name of the compilation unit.

If instead of a file name a single hyphen (`-`) is specified as input, it is read from standard input instead of the file system. Likewise, specifying a single hyphen as output file name, the output is written to standard output instead.

4.5 Testing Regime

In addition to the compiler toolchain, a test suite will be implemented. This test suite is comprised of a number of source files to be exposed to the compiler. The compiler output, warnings and error is then compared to an expected result and the resulting executable, if any, is executed on a hardware implementation or emulator and its result is again compared to the expected result.
As the language extension done by us do not infringe on native C/C++ functionality, all existing test suites designed for the C/C++ language are considered part of the testing regime.
Before the hardware functional units have been synthesized for FPGA, a QEMU just-in-time binary-translation emulator will allow us to check compiler output against simulated hardware.

5 Conclusions

- We perceive the largely unknown inner workings of the Rocket chip as the biggest technological risk.
- We perceive the functional approximation of complex arithmetic functions as the (mathematical) problem with the most upside potential.
- We will try to implement this Design during our Implementation phase spanning to the end of July.

List of Tables

List of Figures

References

[1] Ben Keller. RISC-V, Spike, and the Rocket Core. `http://www-inst.eecs.berkeley.edu/~cs250/fa13/handouts/lab2-riscv.pdf`, 2013. [Online; accessed 16-Mar-2016].

[2] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The risc-v instruction set manual, volume i: User-level isa, version 2.0. Technical Report UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014. [Online; accessed 16-Mar-2016].