Introduction
The aim of this project is to create a custom RISC-V core by transforming functions in a C library into custom instructions.

Vitis HLS
The functions in the C library provided by the user will be converted into VHDL modules using Vitis HLS. It is important to note that these modules will be added to the RISC-V core as R-type instructions. Therefore, each module should consist of two inputs and one output. The inputs should be named op1 and op2, while the output should be named op3. Additionally, the module name and file name should not contain uppercase letters. These are critical requirements, as the core may not function properly if these conditions are not met.

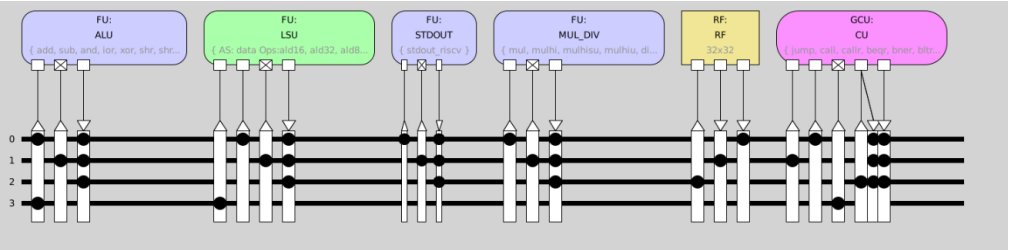
Using TCL, the C functions are fetched one by one, and a folder consisting of VHDL files is created. This folder will be later integrated into the ALU of the RISC-V core. The OpenASIP project was used to create the RISC-V core.

Docker
The TCL script generates VHDL files and then invokes Docker. Within Docker, there are prepared scripts and tools to be used (such as ghdl, Riscv GNU Toolchain, OpenASIP, etc.). After generating the core inside Docker, the prepared C codes are executed. Then, the execution returns to TCL to convert the prepared core into a Vivado project. The user can open the project and make desired modifications.

OPENASIP
OpenASIP is an open application-specific instruction-set processor (ASIP) toolset for design and programming of customized co-processors (typically programmable accelerators). The toolset provides a complete retargetable co-design flow from high-level language programs down to FPGA/ASIC synthesizable processor RTL (VHDL and Verilog generation supported) and instruction-parallel program binaries. Processor customization points include the register files, function units, supported operations, and the datapath interconnection network. The internal processor template of OpenASIP is based on the energy efficient and modular Transport Triggered Architecture (TTA), which is still its default target programming model for static multi-issue designs. OpenASIP, however, also has initial support for other programming models such as standard operation-based VLIW (demonstrated in Blocks CGRA) and since 2.0 it received the first features to support customizing RISC-V ISA based processors.[1]

How to generate a RISC-V core with OpenASIP
OpenASIP tools use some files to generate a RISC-V core. The descriptions of these files are summarized below.

Architecture Definition File (ADF)
Filename extension: .adf
Architecture Definition File (ADF) is a xml file for defining target processor architectures, e.g. which function units are used and how they are connected.

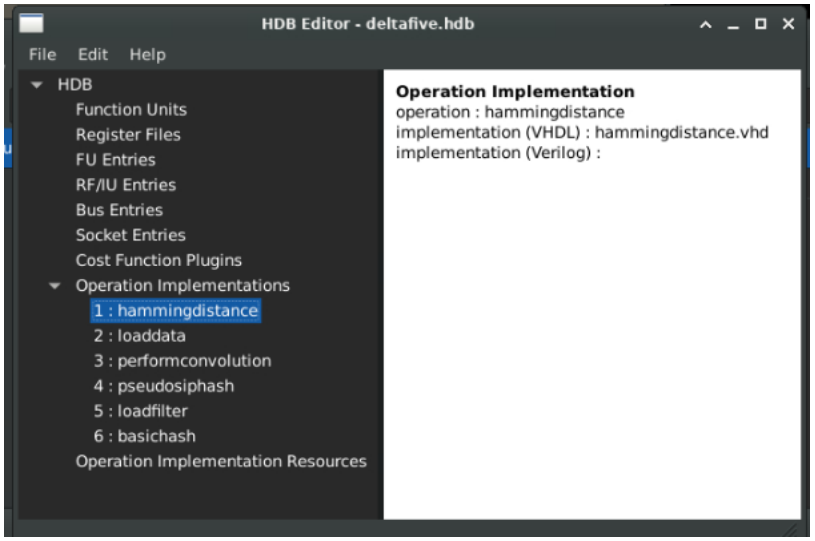


.adf file

Hardware Database (HDB)

Filename extension: .hdb

Hardware Database (HDB) is the main SQLite database used by the Processor Generator. The data stored in HDB consist of hardware description language definitions (HDL) of components (function units, register files, buses and sockets) and metadata that describe certain parameters used in implementation.

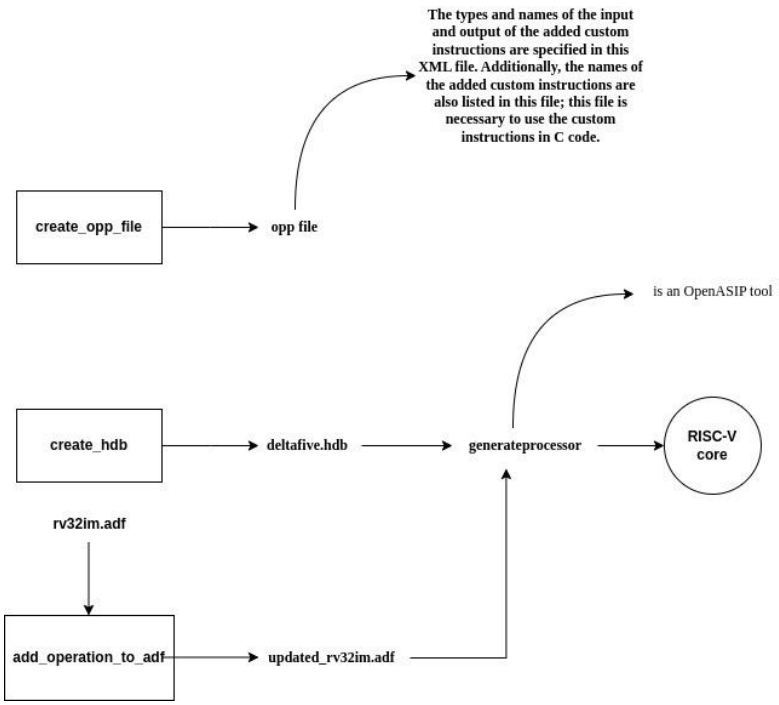


.hdb file

OPP FİLE

The .opp file is an XML file for defining the static properties of operations (for example, how many inputs and outputs an operation has).

In order to generate a core, OpenASIP utilizes some GUI tools. However, in this project, the process of core formation is completely carried out using terminal commands, and Python scripts have been written to automate this process. The diagram of the python script that provides this process can be seen below.



The rv32im.adf file has been obtained from OpenASIP as the default. This file is sufficient to generate a RISC-V core. Depending on the custom instructions the user wants to add, this adf file will be updated and output as updated_rv32im.adf.

We mentioned that the HDB file needs to have the necessary components to generate a core file. According to the desired custom instructions, the names of the custom instructions should be added to the "operation_implementation" table of the SQLite3 file, and the VHDL files of the custom instructions should be added to the "operation_implementation_source_file" table.

Using these two files, the core is generated with the following command.
$ HDBS=asic_130nm_1.5V.hdb,generate_base32.hdb,generate_lsu_32.hdb,deltafive.hdb
$ generateprocessor --hdb-list=HDBS -o core_name -t updated_rv32im.adf

Here HDBS variable contains some hdb files provided by OpenASIP for core generation and hdb file we produced.

As a final step, we need to create the instruction memory and data memory using the OpenASIP tool provided below.
$ generatebits -x core_name rv32im.adf

Compile and run c codes

We have prepared a bash script named "run_c.sh" to execute C code in the generated core. This script uses an OpenASIP tool called "oacc-riscv" command. The oacc-riscv project utilizes the LLVM compiler and OpenASIP configures this compiler for our custom instructions. The resulting .img file from this command is copied to the instruction memory and data memory inside the core, then compiled and simulated. The ghdl tool was used for compilation and simulation purposes.

Using Custom Instruction

In this guide, we will explore how to integrate a custom instruction into an existing C code. We will use an example where we have added a custom instruction for calculating the Hamming distance to an existing RISC-V processor implemented in VHDL.

Step 1: Backup the Original Code

Before making any changes, it is essential to create a backup of the original C code. This allows us to revert to the original version if needed.

$ cp hammingdistance.c custom_hammingdistance.c

Step 2: Understanding Custom Instruction Usage

Custom instructions are invoked using specific macros or intrinsics. The format for using custom operations is as follows:

_OA_<opName>(op1, op2, op3);

Here, **<opName>** represents the name of the custom operation. Input operands are op1, op2 and op3 are output operand. For our example, we have defined a custom operation named "HAMMINGDISTANCE". The intrinsic calls for these operations are as follows:

_OA_HAMMINGDISTANCE(op1, op2, op3);

Step 3: Modifying the C Code

To integrate the custom instruction into the C code, we need to make several modifications.

-> hammingdistance function:

```
int hammingdistance(int op1, int op2) {
   int n = op1 ^ op2;
   int dist = 0;
   for (int i = 0; i < 32; i++) {
      dist += (n >> i) & 1;
   }
   return dist;
}
```

-> custom hammingdistance function:

```
int hammingdistance(int op1, int op2) {
   int output;
   _OA_RV_HAMMINGDISTANCE(op1, op2, output);
   return output;
}
```

In this example code, we have modified the original functions to use custom instructions for specific operations. Each modified function now invokes the corresponding custom instruction using the provided macros. The output values from the custom instructions are returned as the function results.

If the user wishes, they can define the custom instruction in the specified format within the main function and call it, assigning it to a desired variable and using it as they wish.

Modifications and Performance Analysis for RISC-V Core

To enable the functionality of the extracted RISC-V core, several modifications have been made. The fu_alu.vhd file received from OpenASIP only accepts HDL snippets. In this project, scripts were written to facilitate the conversion of the C code-based module into a format compatible with the fu_alu.vhd file.

One of our goals was to write a script that makes certain modifications within the fu_alu.vhd file located in the core. The purpose of this script is to change the positions of desired code lines. To achieve this, we used the Python programming language to read the fu_alu.vhd file. We then extracted the code lines consisting of "signal" commands and wrote them to the desired lines.

Another goal was to write a script that calculates the performance and efficiency of the C code generated from the core. For this task, we used the Python programming language. Each C code performs differently in terms of performance. The report for this information can be found in files with the .dump extension. The information about how many cycles it took for the C code to complete all operations is stored in a file named cycles.dump. We wrote a script that compares the numbers in the cycles.dump files to determine the difference in cycles between the previously generated C code and the newly generated one. Furthermore, this script reads the riscv_instruction_trace.dump file, which contains the instruction sets processed within the core, and performs certain operations. The first task is converting hexadecimal numbers to binary numbers. The purpose of this conversion is to facilitate the identification of 7-bit opcodes within the RISC-V instruction format, whereas 2-digit hexadecimal numbers consist of 8 bits. In short, the goal here is to make it easier to identify the 7-bit opcodes from hexadecimal numbers. The script reads the opcodes of instructions one by one and counts which instructions access the ALU. There are specific opcodes that determine which instructions access the ALU, such as ADD, SUB, XOR, OR, AND, SLL, SLR, and other basic instruction commands. For example, the ADD instruction has a 7-bit opcode "0110011". The script we wrote counts the opcodes sent to the ALU, enabling a performance comparison between the two C codes. The C code with fewer instructions sent to the ALU is considered more efficient. Additionally, the information about how many register files different C codes have can be found in a file named rf.dump. This script compares the number of register files each C code has. In general, this script prints the difference between the cycle numbers of the C codes, how much efficiency is achieved between the cycle numbers, the difference between the number of accesses to alu, how much efficiency is obtained between the numbers of accesses to alu and the difference between the number of register files to the "statistics.txt" file.

An example of statistics.txt

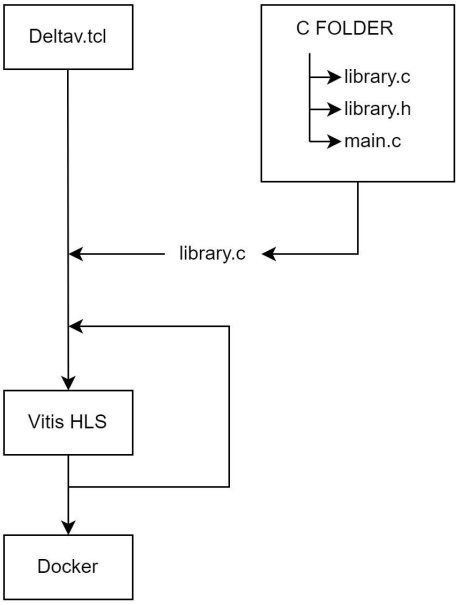
STATISTICS OF TOTAL CYCLES: 97 cycles less and %0.326 efficient.
STATISTICS OF STALL CYCLES: 41 cycles less and %0.352 efficient.

STATISTICS OF ACCESS TO ALU: 43 accesses less and %0.318 efficient.

STATISTICS OF REGISTER ACCESS: Previous: 11024   Next: 10987

Mechanism

The following diagram illustrates the main mechanism described. At the end of this flow, a RISC-V core is generated.



The image below demonstrates the correction scripts for the core generated from Docker, as well as the execution mode for C code. If desired, the user can run their preferred C code on the core after it is created.