**AES** **Add-on processor for RISC-V**

Project Report

Students:

David Dan

Dolev Vaknin

Supervisor:

Eric Herbelin

Table of Contents

[**Abstract** 4](#_Toc56888252)

[**List of abbreviations** 5](#_Toc56888253)

[**Introduction** 6](#_Toc56888254)

[RISC-V 6](#_Toc56888255)

[Rationale 6](#_Toc56888256)

[Specifications 6](#_Toc56888257)

[RI5CY 7](#_Toc56888258)

[PULP 8](#_Toc56888259)

[PULPino 8](#_Toc56888260)

[PULPenix 10](#_Toc56888261)

[AES 11](#_Toc56888262)

[AES ciphering process 11](#_Toc56888263)

[KeyExpansionUnit 13](#_Toc56888264)

[SubBytes 13](#_Toc56888265)

[ShiftRows 13](#_Toc56888266)

[MixColumns 13](#_Toc56888267)

[AddRoundKey 13](#_Toc56888268)

[**Possible solutions** 15](#_Toc56888269)

[1. Encryption from buffer 15](#_Toc56888270)

[2. Encryption command 15](#_Toc56888271)

[3. Encryption on the fly 16](#_Toc56888272)

[**Interface architecture** 17](#_Toc56888273)

[AES commands opcode 17](#_Toc56888274)

[AES Commands structure 18](#_Toc56888275)

[ AES REG: 18](#_Toc56888276)

[ AES KEY: 18](#_Toc56888277)

[ AES MEM: 18](#_Toc56888278)

[ AES\_RUN 19](#_Toc56888279)

[**Design and Implementation** 20](#_Toc56888280)

[Flow 20](#_Toc56888281)

[AES register file 21](#_Toc56888282)

[Simulate the AES register file 21](#_Toc56888283)

[AES engine 22](#_Toc56888284)

[Simulate the AES engine 23](#_Toc56888285)

[AES write-back 24](#_Toc56888286)

[AES engine with register file and write-back module 25](#_Toc56888287)

[Integrate the AES flow with RISC-V core 26](#_Toc56888288)

[**Simulate and Debug** 27](#_Toc56888289)

[**Performance** 29](#_Toc56888290)

[CoreMark 29](#_Toc56888291)

[**Power** 30](#_Toc56888292)

[Area 31](#_Toc56888293)

[Timing 32](#_Toc56888294)

[**Appendix A: User guide** 33](#_Toc56888295)

[Synthesis 34](#_Toc56888296)

[Simulation 36](#_Toc56888297)

[AES verification test 37](#_Toc56888298)

[Simvision 38](#_Toc56888299)

[Trace file 39](#_Toc56888300)

# **Abstract**

In our days, where we use sensors and processors anywhere and anytime, it’s getting very hard to keep the data we receive from those devices – safe.

There are a lot of ways to get data from devices that don’t belong to us, and that’s because, in most of the cases, the raw data sent from the sensors to a relatively far processor, and only after processing the information, the processor encrypts the processed data. Which means, there are a lot of chances to steal the information between collecting the raw data by the sensor, and encryption of that data by the processor.

The goal of this project is to design and implement an AES encryption (or any other safe encryption method) on the RISC-V processor, which will enable using the RICS-V for processing the information from the sensors as close to the sensor as can be and doing so safely.

In this project, we are using the RI5CY core embedded in the Pulpenix microcontroller.

# **List of abbreviations**

CPU – Central Processing Unit

ISA – Instruction Set Architecture

RISC – Reduced Instruction Set Computer

AES – Advanced Encryption Standard

SOC- System on Chip

PULP – Parallel Ultra-Low Power

LSU – Load Store Unit

ID – Instruction Decode

EX - Execute

WB - Write Back

RTL -Register Transfer Level

IoT – Internet of Things

SV- System Verilog

**Introduction**

RISC-V

The RISC-V project started in 2010 at Berkeley USA.

Since 2015 RISC-V under the supervision of the non-profit RISC-V Foundation, with over 200 member organizations, from both academia and industry, including Samsung, Google, NVIDIA, Qualcomm, Berkeley Architecture Research, ETH Zurich, and many more.

Rationale

Most of the semiconductor companies (such as MIPS, ARM, etc.) charge royalties for the use of their designs and patents, due to the high cost of designing a CPU.

RISC-V was designed as an open-source ISA, that anybody can change and use for his personal or commercial uses, without paying royalties.

The original goals of RISC-V were [[[1]](#footnote-1)][[[2]](#footnote-2)]:

* ISA that supports a wide variety of practical uses
* Usable academically.
* Usable in any hardware or software design.

Specifications

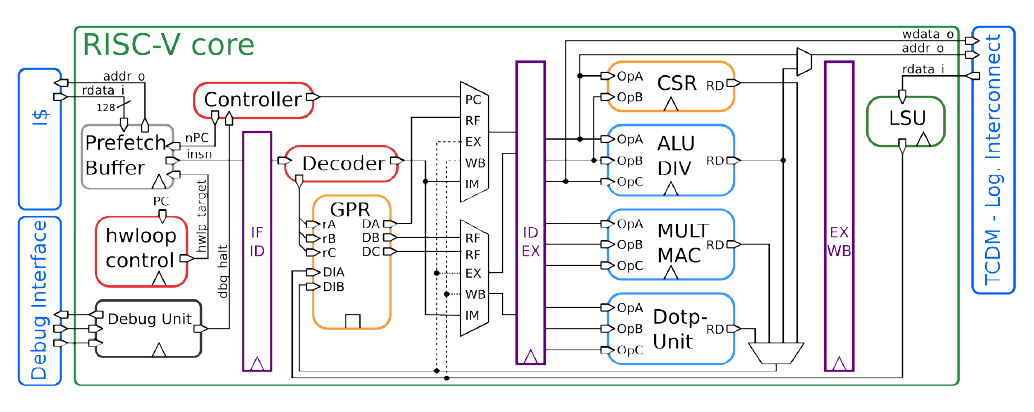
The RISC-V ISA is designed for a wide range of uses. It is variable-width and extensible so that more encoding bits can always be added. It supports three word-widths: 32, 64, and 128 bits, and a variety of subsets. The definitions of each subset vary slightly for the three word-widths [[[3]](#footnote-3)].

RISC-V has 32 (or 16 in the embedded version) [integer](https://en.wikipedia.org/wiki/Integer) registers, and 32 separate floating-point registers when the floating-point extension is implemented.

RISC-V is implemented in a [load-store architecture](https://en.wikipedia.org/wiki/Load%E2%80%93store_architecture), which means that instructions address only registers, with load/store instructions to read/write from/to the memory.

RI5CY

RI5CY is a 4-stage in-order 32-bit RISC-V processer[[4]](#footnote-4) , with some added instructions not in the standard RISC-V ISA. The top-level diagram of the core:



PULP

PULP is an open-source computing platform designed by ETH Zurich and the University of Bologna - started in 2013.

The processors embedded in the PULP range from single-core to multi-cluster (more than one processor).

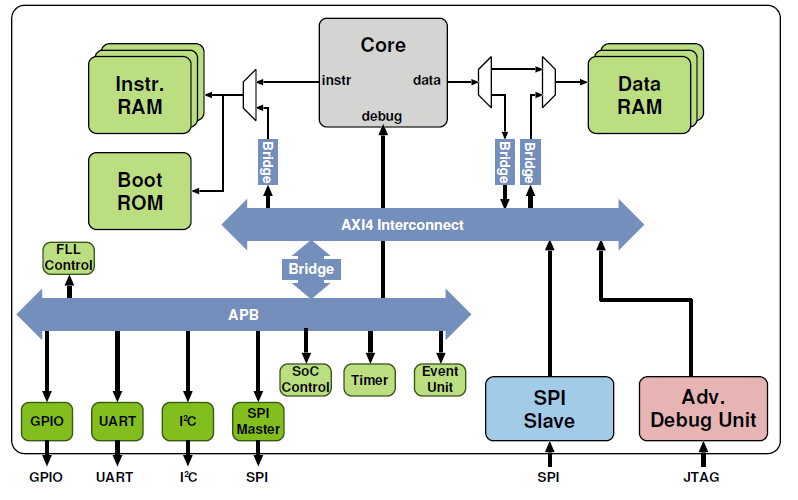
The PULP architecture targets IoT applications requiring low-power, flexible processing of data streams generated by multiple sensors, such as accelerometers, low-resolution cameras, microphone arrays, vital signs monitors.

PULP consists of an advanced microcontroller architecture representing a significant step ahead in terms of completeness and complexity.

PULPino

PULPino is a single-core System-On-a-Chip (“SOC”) built for the RISC-V RI5CY and ZERO-RI5CY core. PULPino reuses most components from its bigger brother PULP. It uses separate single-port data and instruction RAMs. It includes a boot ROM that contains a boot loader that can load a program via SPI from an external flash device. The SOC uses an AXI as its main interconnect with a bridge to APB for simple peripherals. Both the AXI and the APB buses feature 32-bit wide data channels. For debugging purposes, the SOC includes an advanced debug unit that enables access to core registers, the two RAMs, and memory-mapped IO via JTAG. Both RAMs are connected to the AXI bus via bus adapters [[[5]](#footnote-5)].

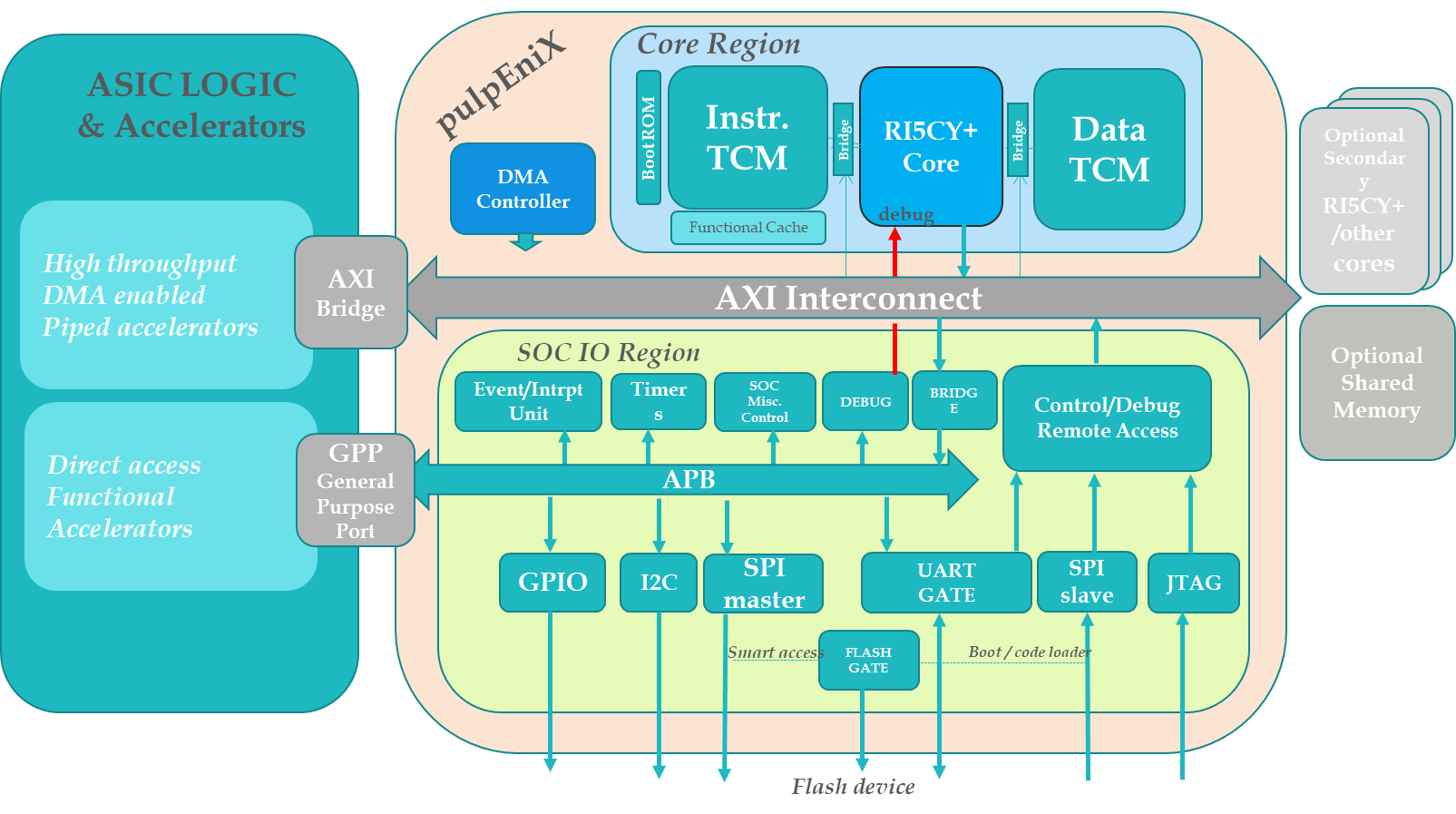
PULPino top level:



PULPenix

Developed in Bar-Ilan by Yehuda Kra, the PULPenix platform took the existing open-source implementation of the PULPino processor and added an interface and compiler system for the software so bare-metal C programs could be compiled and run on the processor. It also includes a set of scripts and tools to allow easy compilation and running of the program, debugging, making waveforms of the processor signals, and making a tracing of the assembly commands including the simulation times and register values. This project used PULPenix as the simulation environment. The existing software interface was used and expanded upon to learn about, and eventually modify the processor.

PULPenix top level:



AES

AES is a symmetric [block cipher](https://en.wikipedia.org/wiki/Block_cipher), adopted by the U.S. government and governments and organizations all around the world[[[6]](#footnote-6)].

The AES algorithm is a subset of the Rijndael block cipher, developed by [Vincent Rijmen](https://en.wikipedia.org/wiki/Vincent_Rijmen) and [Joan Daemen](https://en.wikipedia.org/wiki/Joan_Daemen), and chosen by NIST during the AES selection process in November 2001[[[7]](#footnote-7)]. Rijndael is a family of ciphers with different key and block sizes. For AES, NIST selected three members of the Rijndael family, each with a block size of 128 bits, but three different key lengths: 128, 192, and 256 bits[[[8]](#footnote-8)].

The AES algorithm is a [symmetric-key algorithm](https://en.wikipedia.org/wiki/Symmetric-key_algorithm), meaning the same key is used for both encrypting and decrypting the data, which makes it easier to send and receive an encrypted text.

AES is a block cipher, meaning it encrypting blocks (128-bit blocks) of the text each time.

AES ciphering process

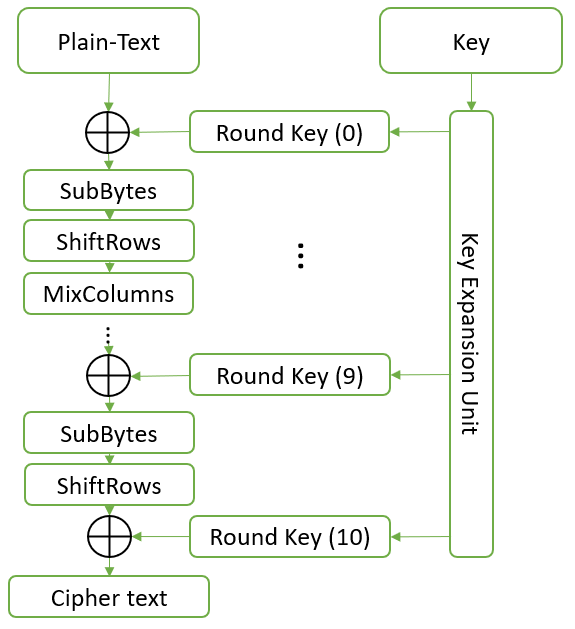
The AES consisting of 10, 12, or 14 rounds.

The first round is consisting of the steps: AddRoundKey, SubBytes, ShiftRows, MixColumns, AddRoundKey (in that order).

The next 8, 10, or 12 rounds consisting of the steps: SubBytes, ShiftRows, MixColumns, AddRoundKey.

The last round consisting the steps: SubBytes, ShiftRows, AddRoundKey.

AES diagram



KeyExpansionUnit

round keys are derived from the cipher key using the [AES key schedule](https://en.wikipedia.org/wiki/AES_key_schedule). AES requires a separate 128-bit round key block for each round plus one more (before the first round).

SubBytes

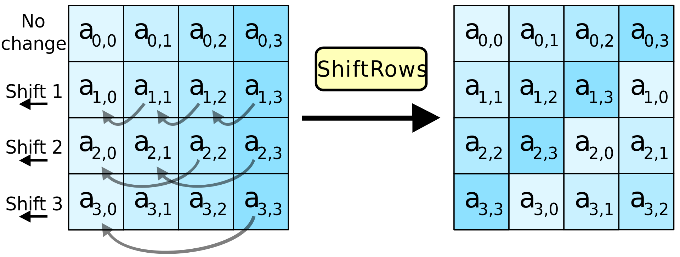
A non-linear substitution step where each byte is replaced with another according to a lookup table (AES S-box).

A picture containing crossword, clock, black, hanging

Description automatically generated

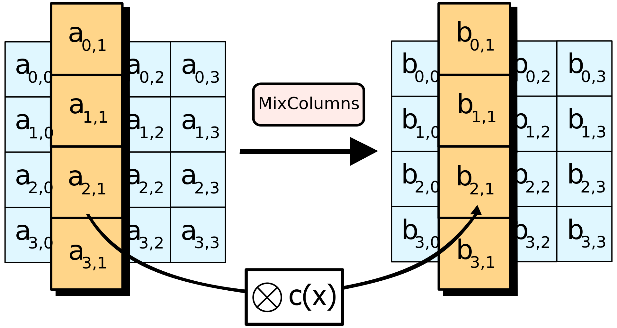
ShiftRows

A transposition step where the last three rows of the state are shifted cyclically a certain number of steps.



MixColumns

A linear mixing operation that operates on the columns of the state, combining the four bytes in each column.



AddRoundKey

Each byte of the state is combined with a byte of the round key using [bitwise xor](https://en.wikipedia.org/wiki/Bitwise_xor).

A close up of a clock

Description automatically generated

# **Possible solutions**

There was 3 optional solution on the table:

1. Encryption from buffer

In this solution, the RISC-V core will hold 9 new AES registers, holding 4 data registers, 4 key registers, 1 write-back register. There will be a special command to write data to those registers, and running the command:

1. Storing 4 key registers in the AES register file.
2. Storing write-back register in the AES register file.
3. Storing 4 data registers in the AES register file.
4. Calling the AES command:

The command will cipher the AES data registers using the AES key registers, and store the ciphered 4 data registers to the memory (starting in the address given in step b.)

Advantages:

* + Relatively easy to implement.
  + Relatively secure – once the 4 registers hold the new data, the ciphering process can start.

Disadvantages:

* + Wasteful in terms of area - requires 9 new registers, especially for the AES.
  + Need to encrypt 4 registers at every encryption process.

1. Encryption command

In this solution, the command will be the same as “Store”, but instead of writing the data directly to the memory, the data will be ciphered before:

* 1. Storing the key inside special 4 registers (hardware protected, nonvolatile).
  2. Calling the AES command (equivalent to “Store” command):

The command will load the 4 data registers from the memory (from the address given in step b., but after alignment to 128-bit), decipher them, replace the correct (old) register with the new register, run the AES algorithm, and store the ciphered data back to the required address in the memory (read-modify-write).

Advantages:

* + Can store one register (instead of 4 at a time).

Disadvantages:

* + Not very secure – the data will wait in the core, un-ciphered, till finishing loading the 4 registers from the memory and deciphering them.
  + Hard to implement – there is a need for deciphering unit.
  + Wasteful in terms of time, power, and area - requires 4 loads from the memory and 4 stores to the memory.

1. Encryption on the fly

In this solution, we don’t use the AES algorithm, but using another cyphering method, that can cipher 32-bit block, so we could cipher data before writing it to the register, and storing ciphered data without loading data from the memory first (read-modify-write):

* 1. Storing key inside register (hardware protected, nonvolatile)
  2. Using encryption command, equivalent to “Store” command (or R type command):

The command will cipher the data and store it into the memory (or register).

Advantages:

* + Easy to implement.
  + The memory (including the registers) would hold only ciphered data.
  + Economical in terms of time and power (only one register is ciphered at any time, with a less complex ciphering algorithm.).

Disadvantages:

* + Not secure – due to the use of a less complex ciphering algorithm.

After discussing the advantages and disadvantages of the 3 solutions, we decided to implement solution number one, which is Encryption from a buffer.

# **Interface architecture**

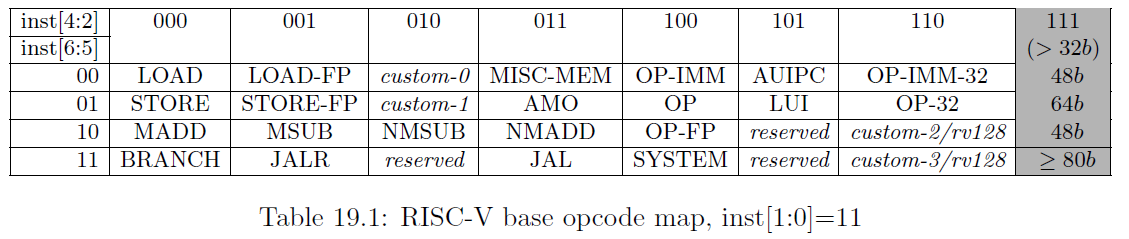
To implement our solution to encryption command in RISC-V, we decided to use the following command structure (from R type commands):



Where the opcode will be the same to all AES commands, while funct3 will distinguish between the AES commands, rd will be AES register (data or key), and rs1 will be RISCV register.

## AES commands opcode

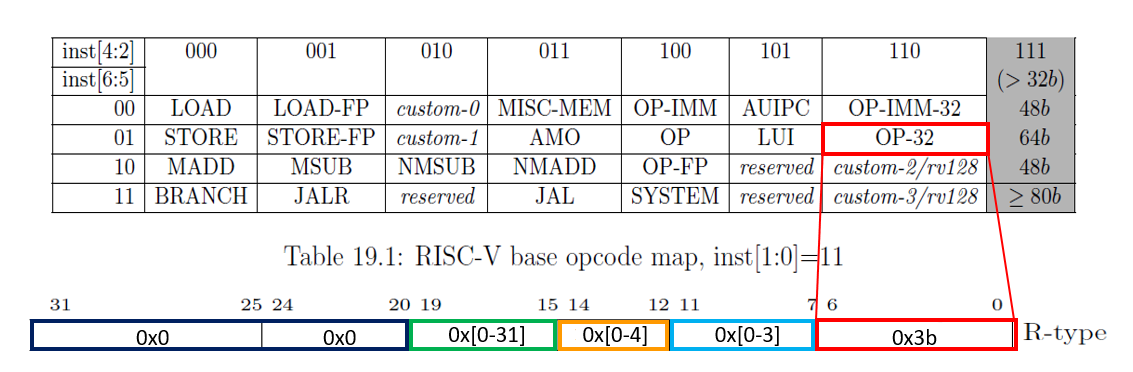
The RISC-V opcode map:



In the PULPenix RI5CY core, some of the opcodes in the table are not in use (not necessarily the “custom” opcodes).

We discovered that 0x3b opcode (OP-32 in the table) is not in use.

We decided to use 0x3b as AES commands opcode:



Opcode

AES register (key or data)

AES function

RISCV register

Not in use

### AES Commands structure

To run the AES encryption command, first, the AES register file must hold the correct data, key, and write-back register.

Writing to the AES register file is done using the commands:

* AES REG:

Writing data to AES data registers, it receives as parameters AES data register ( d[0:3] ) and RISC-V register.

Funct3 = 0

Command structure:



0x3b

0x[0-3]

0x00

0x[0-31]0

0x00

0x00

For example:

AES REG d0, t1

* AES KEY:

Writing key to AES key registers, it receives as parameters AES key register ( k[0:3] ) and RISC-V register.

Funct3 = 1

Command structure:



0x3b

0x[0-3]

0x10

0x[0-31]0

0x00

0x00

For example:

AES KEY k1, t3

* AES MEM:

Writing an address to the AES WB register, which holds the write-back address. It receives as parameter RISCV register.

Funct3 = 2

Command structure:



0x3b

0x0

0x20

0x[0-31]0

0x00

0x00

For example:

AES MEM t5

After storing the data, key, and write-back address in the AES register file, we can use the AES RUN command (with no parameters), to start the ciphering process:

* AES\_RUN

Funct3 = 4



0x3b

0x0

0x40

0x0

0x00

0x00

For example:

AES RUN

At the end of the ciphering process, the AES WB module will write the ciphered data to the memory in 4 blocks of 32-bit, to the 4 memory blocks that start at the address stored in the write-back address.

# **Design and Implementation**

## Flow

The flow if the AES implementation inside the RISC- core is divided into three modules:

* AES register file (“riscv\_aes\_registers”):

Contains 4 data registers, 4 key registers, and 1 write-back address register.

It receives AES instruction number, and instruction parameters (if any are given), and according to the instruction it manipulates the given parameters and sends the correct signals out.

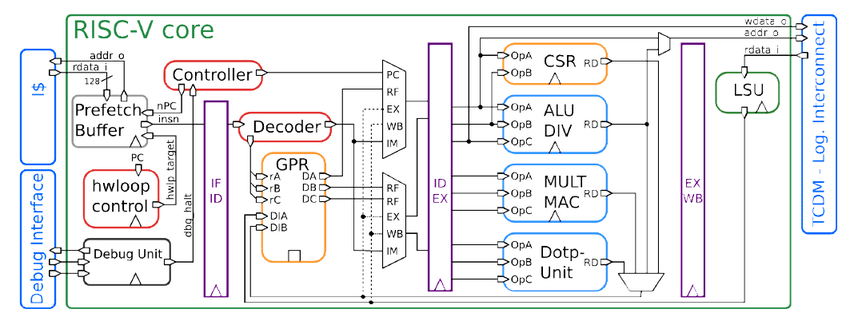
* AES Engine (“riscv\_aes\_cipher”):

Contains the AES algorithm.

* AES WB (“riscv\_aes\_wb”):

Functions as a write-back module, it contains a state machine, which halts the RISC-V pipe for 4 cycles each time it receives a new ciphered text.

In each one of the 4 cycles, it writes one register (32-bit) of ciphered data to the memory (starting at the address stored in the write-back register).



AES engine

AES reg file

AES WB

128-bit data

128-bit key

32-bit

128-bit data

128-bit ciphered

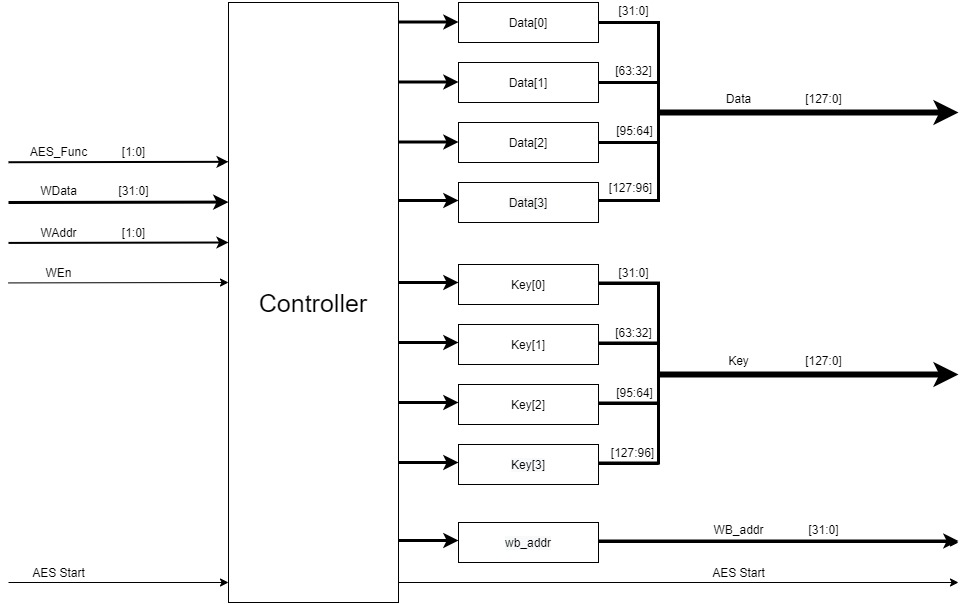
32-bit

### AES register file

We designed the AES register file to hold 4 data registers, 4 key registers, and a write-back register.

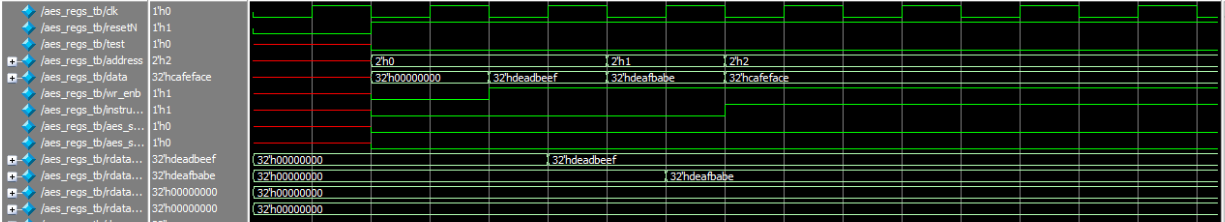
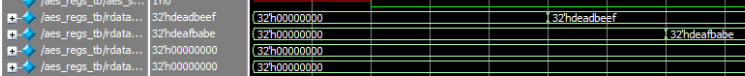
Also, the AES register file receives a “start” signal.

The register file sends the above registers and the “start” signal to the AES engine once the “start” signal raises.



#### Simulate the AES register file

In the simulation below, we loaded the 128’h deadbeefdeafbabe0000000000000000 into the data registers inside the AES register file:



### AES engine

The first phase of the project was to implement 10 rounds, 128-bit, AES algorithm in SV.

We found a basic implementation of the above algorithm in Git-Hub ([AES-128-Bit-Verilog](https://github.com/pnvamshi/Hardware-Implementation-of-AES-Verilog/tree/master/AES-128-Bit-Verilog)), we used it as a basic starting point to the RISC-V AES implementation.

The implementation is combinatorial, which means that after we updated the inputs to the algorithm, we can’t know when the output is ready.

To solve this problem, we added a new signal that runs through every step of the AES flow, and we set that signal to logic 1 when we want the AES engine to encrypt our data, thus when the new signal turns to 1 in the output, we know that the AES engine finished encrypting our data, and we can take the encrypted data and store it to the memory.

The top level of the implementation is “riscv\_aes\_cipher”, which receives 128-bit data, 128-bit key, 32-bit WB address, and start signal.

The top-level is running the 10 rounds of AES cipher one after another, by calling “rounds” module in the first nine rounds, and “roundlast” in the tenth round.

The “rounds” module calls the following modules in each round (in that order):

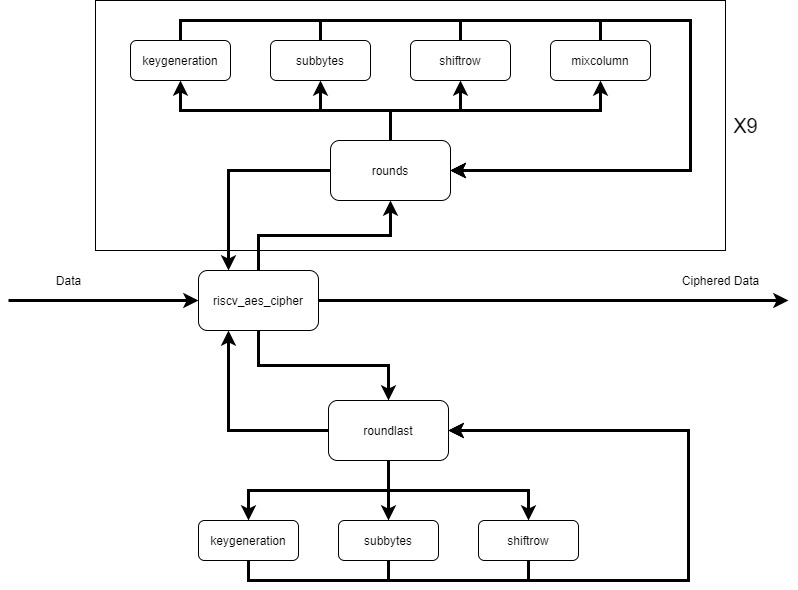
keygeneration

subbytes

shiftrow

mixcolumn

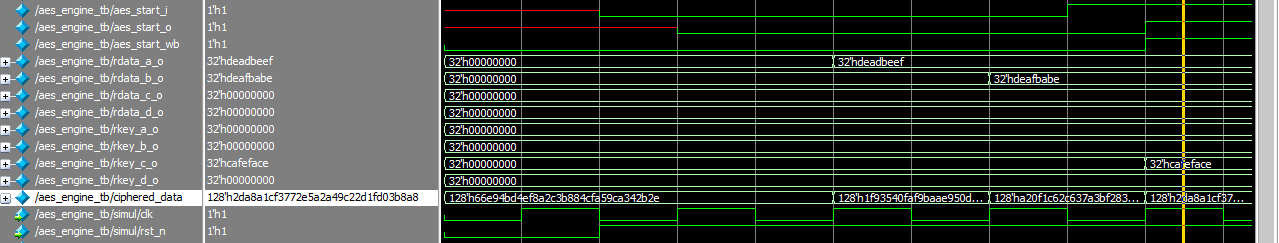
Each of the modules represents each one of the AES processing steps that occur in every round (except the last round, which runs only the first three steps).



#### Simulate the AES engine

In the simulation below (in Mentor Graphics’s ModelSim PE), we loaded the 128’h deadbeefdeafbabe0000000000000000 into the data registers, and 128’h 0000000000000000cafeface00000000 into the key registers, and set the start signal to 1.

We can see in the waveform, that in the cycle after setting the start signal to 1, the start signal in the output raised, and the cyphered data bits updated to the correct cyphered data: 128’h 2da8a1cf3772e5a2a49c22d1fd03b8a8 (verified by AES algorithm in OnlineDomainTools site: http://aes.online-domain-tools.com/):



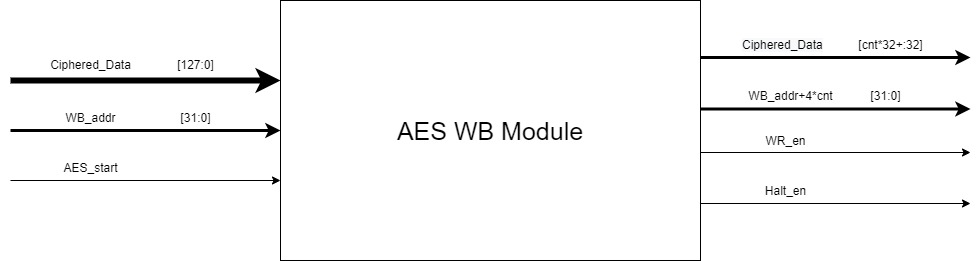
### AES write-back

We designed the write-back as a state-machine.

The state-machine starts in the “IDLE” state and waits in this state till the “start” signal from the AES engine raises.

Once the “start” signal raises, it switches to the “WRITE” state, and in this state the “halt\_en” signal raises (which halts the pipeline), parsing the data to 32-bits, and start to write the data to the memory (7 clock cycles every write).

After finishing with the memory writing, the state-machine switches to the “FINISH” state, which resets the “halt\_en” signal, and return the machine to the “IDLE” state.

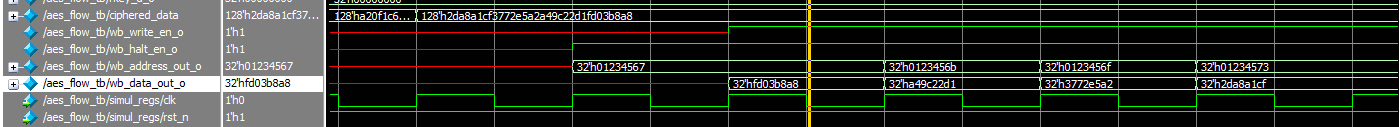


### AES engine with register file and write-back module

After implementing the AES register file and write-back module, we integrate it with the AES engine and simulate it.

In the simulation below, we loaded the same data and key as the last simulation, but this time to the register file (instead of directly to the AES engine) and received the same ciphered data.

The ciphered data was parsed by the write-back module into 4 32-bit registers (in RED), and each one of the registers was sent out with an address, while the address is raised by 4 (in Orange) with each register (the registers are stored in adjacent locations in the memory):



### Integrate the AES flow with RISC-V core

After implementing and simulating the AES flow, we integrated the AES flow with the RISC-V core, and from this step, we switched from working with Mentor Graphics’s “ModelSim PE”, to working with Cadence and Synopsys tools, that provided to us by VLSI lab.

During the integration, we decided to put the AES modules (register file, engine, and write-back) in the CORE module of the RISC-V, which gives us the ability to take signals from the original RISC-V modules relatively easy, and add the AES flow as parallel as we can to the RISC-V flow (keep the RISC-V performance).

In addition to adding the AES modules, we had to update modules in the ID, EX, and WB stages.

There were a few different changes to be done:

* Decoding of AES commands:

Since we added new commands to the RISC-V, we had to add those commands to the Decoder module, to parse the new commands properly.

* Signals in the ID stage:

To receive data from the main register file (according to the AES commands), we had to update signals in the ID stage.

* Signals in the EX stage:

Halting the pipe during the write-back of ciphered data to the memory was done by sending signals to the EX stage.

* Write-back signals and data:

To write the ciphered data in the memory, we had to halt the pipe, supply the data and addresses to be written, and raise a writing signal to the LSU.

# **Simulate and Debug**

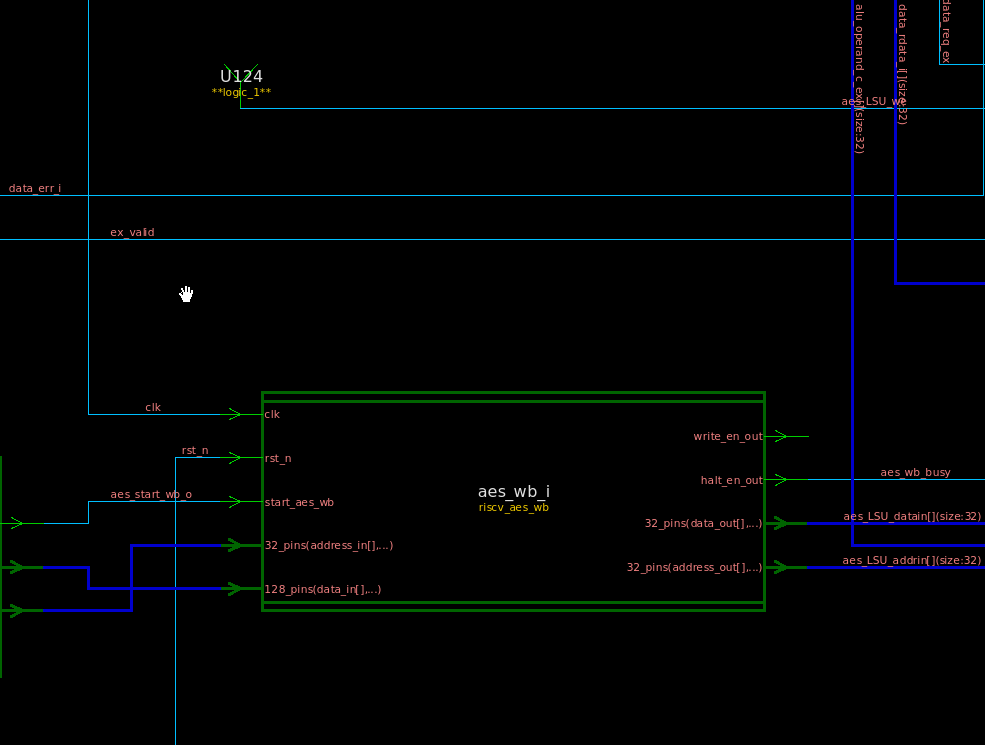
When we first started to run the full AES flow on the RISC-V, we had a lot of problems:

* Syntax errors:

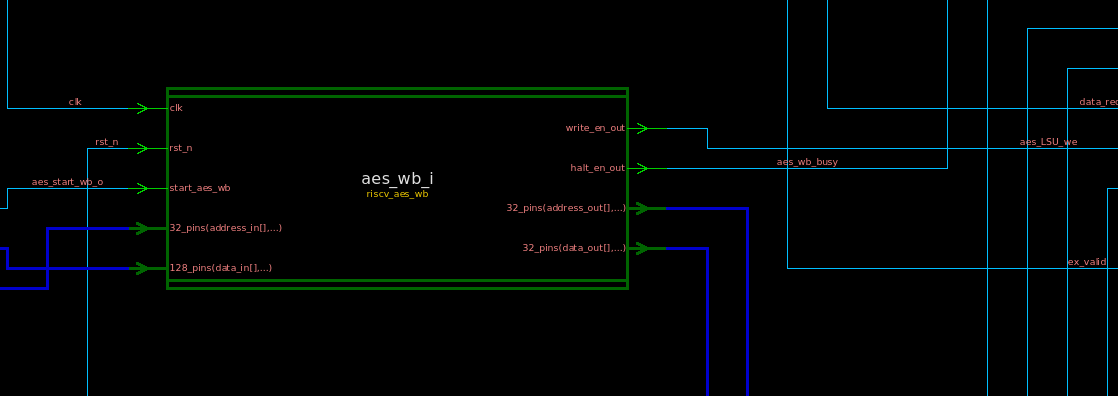
Mismatched signal names - fixed easily by renaming the incorrect signal.

* Connections problems:

After synthesis, we saw in the RTL schematic view that one of the outputs of the write-back module is disconnected:

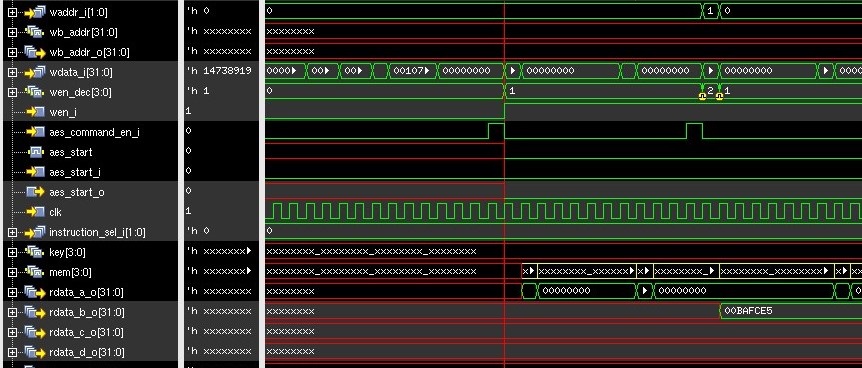


After looking for the problem in the AES write-back module and the LSU, the cause of this issue was defining the write\_en\_out signal only once (to logic 1) in the AES write-back module, which caused the dc\_shell to replace the connection with logic 1 signal during synthesis. After adding resets to this signal, the problem was solved:



* Flow issues

When we started testing the AES flow, we had an issue with writing to the first AES data register, which caused the value we written to the register being overwritten with 0:

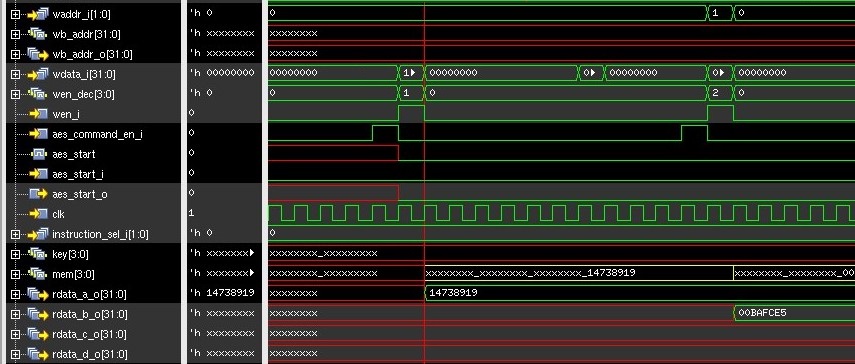


We can see in the waveform that data is being stored in the first AES register, and in the next cycle, the data is replaced with zeros (in RED).

The cause of this bug was the wen\_i signal, which changes to logic 1 (in BLUE) once we started writing to the AES register file and didn’t go back to logic 0.

We found the source of this bug in the ID stage main module, due to updating the above signal only when we have the AES command, thus resulting in wen\_i stuck on logic 1 after AES writing command.

We fixed the issue by updating the wen\_i signal on every clock cycle:



# **Performance**

## CoreMark

|  |  |
| --- | --- |
| GenPro PULPenix | GenPro PULPenix AES |
|  |  |

CoreMark is a benchmark program, a program that runs several chosen algorithms, that test different aspects of hardware performance.

The benchmark program gives, as a result, the number of clock cycles took the CPU to run the program, to compare to other CPUs.

As seen above, both the original RISCV and our RISCV AES ran for 369594 cycles.

|  |  |
| --- | --- |
| GenPro PULPenix | GenPro PULPenix AES |
|  |  |
| **Total power :** | **Total power :** |

## Power

As seen above, our solution had a very small effect on the power consumption of the RISCV processor.

The difference in percents:

## Area

|  |  |
| --- | --- |
| GenPro PULPenix | GenPro PULPenix AES |
|  |  |

We can see that the AES flow is very wasteful in terms of area – the RISCV with AES is 4 times bigger than RISCV without AES.

When we implemented the AES flow, we used combinatorial implementation of AES.

By using synchronized implementation, we can improve the area of the AES to be ¼ than the combinatorial implementation, but such implementation will make the AES engine much slower (10 times slower than combinatorial implementation, assuming we use the same clock).

## Timing

|  |  |
| --- | --- |
| GenPro PULPenix | GenPro PULPenix AES |
|  |  |
| **Slack : 7.57 [nSec]** | **Slack : 9.89 [nSec]** |

As seen above, our solution had a very small effect on the critical path delay in the RISCV processor, and the processor can work with a clock without a problem.

# **Appendix A: User guide**

There are some shortcut scripts and environment settings that necessary to work efficiently with PULPenix.

Running PULPenix setup:

> source <path to pulp>/pulp/pulpenix/misc/genpro\_pulpenix\_setup.sh

Consider adding the above line to your .cshrc (in order it to run on startup).

Then you will have the PULPenix commands and scripts available, you can see those at:

<path to pulp>/pulp/pulpenix/misc/scripts

<path to pulp>/pulp/pulpenix/apps/sw\_utils

## Synthesis

To run the synthesis and create a “fresh” copy of PULPenix, you can use the “shortcut” script:

> our\_pulp\_synthesis

The technology in use is Tower , with clock () with duty cycle.

If an error message occurs, you’ll need to do a full manual synthesis via dc\_shell:

1. Type in your terminal the following commands:

>cd $RISCV\_DESIGN\_SYN

>dc\_shell

>start\_gui

1. In the GUI that opened, go to the upper left corner and click:

File -> Analyze -> Add

and add:

riscv\_defines

riscv\_config

apu\_macros

apu\_core\_package

If the last step gives you errors, it’s probably because of the files themselves including each other, so try adding them one by one in that order.

1. Click:

File -> Analyse -> Add

And add everything under the “riscv” folder, except the files mentioned above, the ‘riscv\_tracer’ files, and ‘riscv\_register\_file\_latch’. In general, you should either work with riscv\_register\_file\_latch, for FPGA, or with riscv\_register\_file, for ASIC, but not with both, as explained in the PULP spec (under documentation in the PULPenix\_ceu Github repository).

1. Click:

File -> Analyse -> Add

and add everything under the “aes” folder.

1. Click:

File - > Elaborate

Choose the library to be WORK and the top-level module to be “riscv\_core”

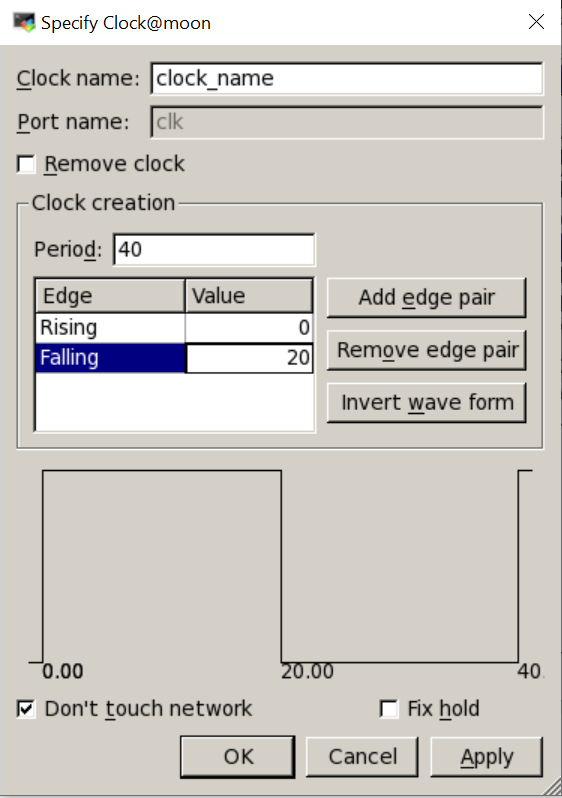
1. Specify a clock:

In the hierarchy pane, click on the hierarchy drop-down menu and filter by “pins/ports”.

Right-click on your clock-signal port and choose `select`.

In the upper bar of the GUI, click:

attributes->specify clock.



We specified the clock to be – 10 [ns] (frequency of 100 [MHz]), and symmetrical.

1. In the upper bar click:

Design -> Compile Design

1. After compilation click:

File->Save As <name>, Name what you want to save then click: Open

To open what you saved click:

File -> Read <name>

1. In the logical hierarchy pane, right-click on the top-level module and select Schematic View. Then, you’ll be able to watch the synthesized file at the gate level. Notice that if you do the schematic view on the RTL files (elaboration) you’ll be able to see logic gates but the names on them would start with Gtech, which means ‘general technology’. After synthesis, however, the names would be the technology names.

## Simulation

In order to run a simulation, a few steps must be done:

1. Create a folder under pulp/pulpenix /apps with your program (source code, .c files)
2. Compile your program:

cd $MY\_PULP\_APPS/<progname>

our\_pulp\_compile <progname>

This step convert the AES commands to .word command (in hexadecimal base), create .s files (assembly files) , .elf files (Executable Linkable Format), and .slm files (memory initialization files) out of your source files.

If the compilation fails, check your code for errors and run again.

1. Get your app:

> cd $MY\_PULP\_IRUN

> pulp\_get\_app <progname>

this step copies your .slm files to the irun folder.

1. Run:

You have three main options for running your program:

* 1. running with wave form:

> pulp\_irun\_probe

The waves file will be created in $MY\_PULP\_IRUN/waves.shm folder.

* 1. running with trace:

> pulp\_irun\_trace

The trace file will be named “trace\_core\_00\_0.log” inside $MY\_PULP\_IRUN folder.

* 1. running with trace and probe:

> pulp\_irun\_probe\_trace

There is a shortcut script that runs steps 2,3 and 4.c:

> our\_pulp\_run <progname>

The waves file will be saved as:

$MY\_PULP\_APPS/waves/<name>.shm

The trace file will be saved as:

$MY\_PULP\_IRUN/trace\_core\_00\_0.log

AES verification test

You can also run the AES verification test. running the test:

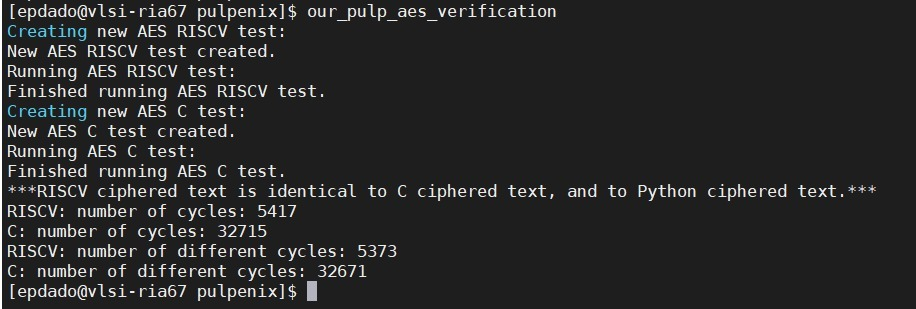
> our\_pulp\_aes\_verification

During the test a random 128-bit key and random 128-bit data are created, new source files with those key and data are created, and the script run the RISCV processor with two different source files:

1. AES implemented in C (C AES) – runs on the RISCV without using the AES logic.
2. AES implemented in assembly (RISCV AES) – runs on the AES logic that we implemented.

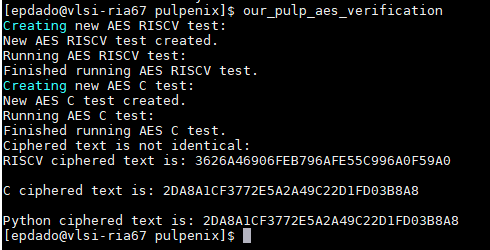
Also, the script encrypts the data using a Python script (Python AES).

At the end of the script a comparison is made between the Python AES encrypted data, RISCV AES encrypted data, and C AES encrypted data (in BLUE):



The number of cycles is also printed (in RED), specifying the number of different cycles between the RISCV AES test and the C AES test.

If one of the three is different from the other, an error message is printed specifying the results of the different implementations (in RED):



## Simvision

To see the waveform of the simulation, open Simvision:

> simvision &

In simvision:

File -> Open Database

in the browse tab, there will be a folder called waves.shm, select it, and then click `open & dismiss`.

A screenshot of a cell phone

Description automatically generated

You’ll find the RISC-V design under tb/top\_i:

A screenshot of a cell phone

Description automatically generated

Now you can add signals of your choice to the waveform, and travel across the simulation timeline.

You can save a command script, which will save all your actions on the Simvision session (signals, markers, position on the timeline, etc.) since you opened it:

File -> save command scrip

Choose a name and location

OK

Running a command script:

File -> source command script

Choose your script

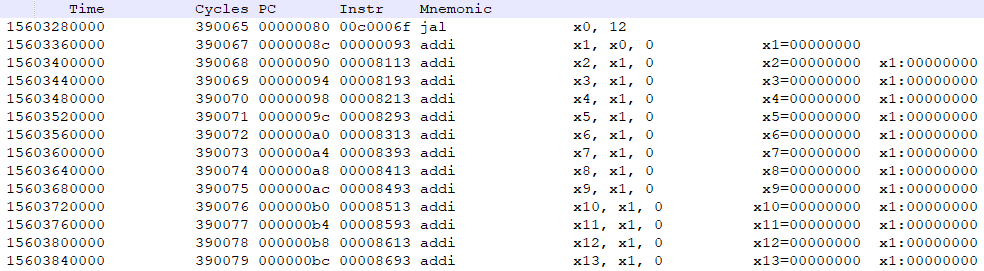
Open

## Trace file

Inside the trace file, you can see each assembly command that runs on the CPU and its:

* Cycle - cycle number in the simulation.
* Time - Time in the simulation.
* PC - Program Counter, the address in which the command is stored inside the instruction memory.
* Instr – hex decoding of the assembly instruction.
* Mnemonic – the assembly instruction, and the values of the registers after the current instruction.

The time column can help in finding the required commands in the waveform.

Example for trace file:

1. Waterman Andrew, [Asanović Krste](https://en.wikipedia.org/wiki/Krste_Asanovi%C4%87): ["The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA version 2.2"](https://riscv.org/technical/specifications/) [↑](#footnote-ref-1)
2. [Asanović, Krste](https://en.wikipedia.org/wiki/Krste_Asanovi%C4%87). ["Instruction Sets Should be Free"](https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.pdf) [↑](#footnote-ref-2)
3. Waterman, Andrew; [Asanović, Krste](https://en.wikipedia.org/wiki/Krste_Asanovi%C4%87). ["The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA version 2.2"](https://riscv.org/technical/specifications/) [↑](#footnote-ref-3)
4. Andreas Traber, Michael Gautschi, and Pasquale Davide Schiavone, RI5CY User-Manual, 6, Apr 2019 [↑](#footnote-ref-4)
5. Pulpino datasheet [↑](#footnote-ref-5)
6. Westlund, Harold B. (2002). "NIST reports measurable success of Advanced Encryption Standard". Journal of Research of the National Institute of Standards and Technology. Archived from the original on 2007-11-03. [↑](#footnote-ref-6)
7. "Announcing the ADVANCED ENCRYPTION STANDARD (AES)" (PDF). Federal Information Processing Standards Publication 197. United States National Institute of Standards and Technology (NIST). November 26, 2001. [↑](#footnote-ref-7)
8. 7 Advanced Encryption Standard, From Wikipedia, the free encyclopedia. [↑](#footnote-ref-8)