Table of Contents

[RISCV Pulpenix: Adding AES engine](#_Toc23005013)

[Acknowledgements](#_Toc23005014)

[Table of Contents](#_Toc23005015)

[List of Figures](#_Toc23005016)

Executive [Summary](#_Toc23005017)

[List of Abbreviations](#_Toc23005018)

[1. Introduction](#_Toc23005019)

[2. Background](#_Toc23005019)

[2.1. RISC-V](#_Toc23005025)

[2.2. PULPino](#_Toc23005026)

[2.3. Pulpenix](#_Toc23005027)

[2.4. AES](#_Toc23005028)

[3. AES Architecture](#_Toc23005035)

[3.1. Top-level](#_Toc23005036)

[3.2. Controller](#_Toc23005037)

[3.3. Interface](#_Toc23005038)

[4. Alternative solutions](#_Toc23005040)

[5. Synthesis](#_Toc23005041)

[6. Simulation](#_Toc23005046)

[7. Comparison with C implementation](#_Toc23005046)

[7.1. Advantages](#_Toc23005025)

[7.2. Disadvantages](#_Toc23005025)

[8. Conclusion](#_Toc23005057)

[Appendix A : List of Design Changes](#_Toc23005058)

[Appendix B : How To Guide](#_Toc23005059)

[References](#_Toc23005061)

**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 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 support 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.
* RISC-V handles 32-bit constants and addresses with instructions that set the upper 20 bits of a 32-bit register (LUI – load upper immediate), Then a second instruction can set the bottom 12 bits.
* RISC-V segregates math into a minimal set of [integer](https://en.wikipedia.org/wiki/Integer) instructions with [add, subtract, shift, bit-wise logic](https://en.wikipedia.org/wiki/Arithmetic_logic_unit) and comparing-branches.

**PULPINO**

PULPino is a single-core System-on-a-Chip (“SOC”) built for the RISC-V RI5CY and zero-riscy 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 which 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 [[[4]](#footnote-4)].

**AES**

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

The AES algorithm is a subset of 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 on November 2001. 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.

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 encrypted text.

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

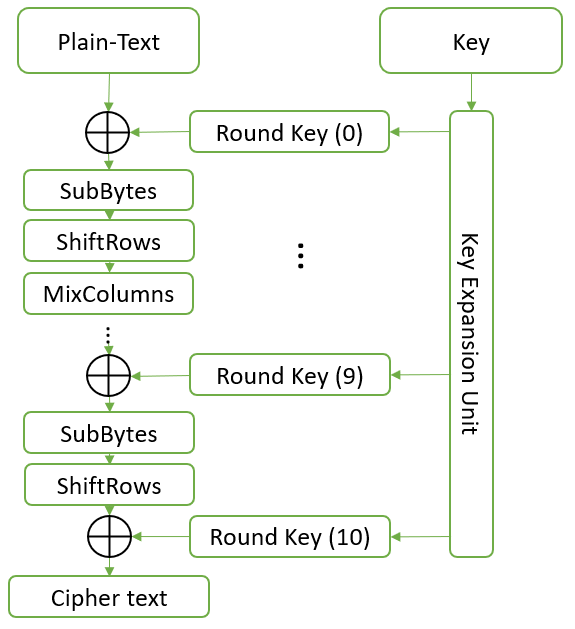
AES ciphering process:

The AES consisting of 10, 12 or 14 rounds.

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

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

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



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

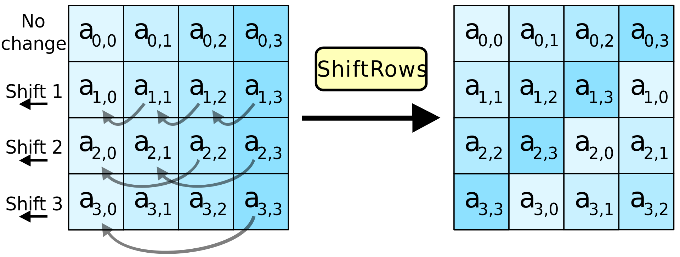
A non-linear substitution step where each byte is replaced with another according to a lookup table (AES S-box).

A picture containing crossword, clock, black, hanging

Description automatically generated

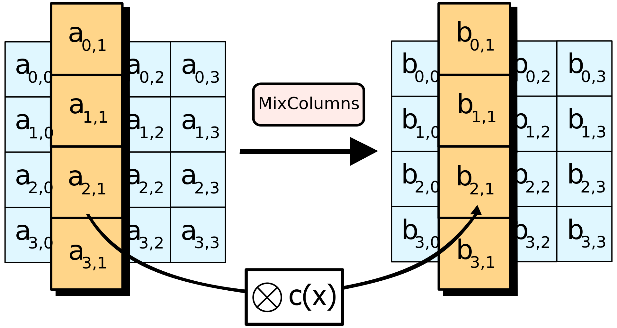
ShiftRows

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



MixColumns

A linear mixing operation which 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

**AES architecture**

We used a simple implementation of AES algorithm in SystemVerilog.

The implementation is combinatorial, which means that every change in the inputs cause a change in the outputs.

The top level of the implementation is “riscv\_aes\_cipher”, it 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):

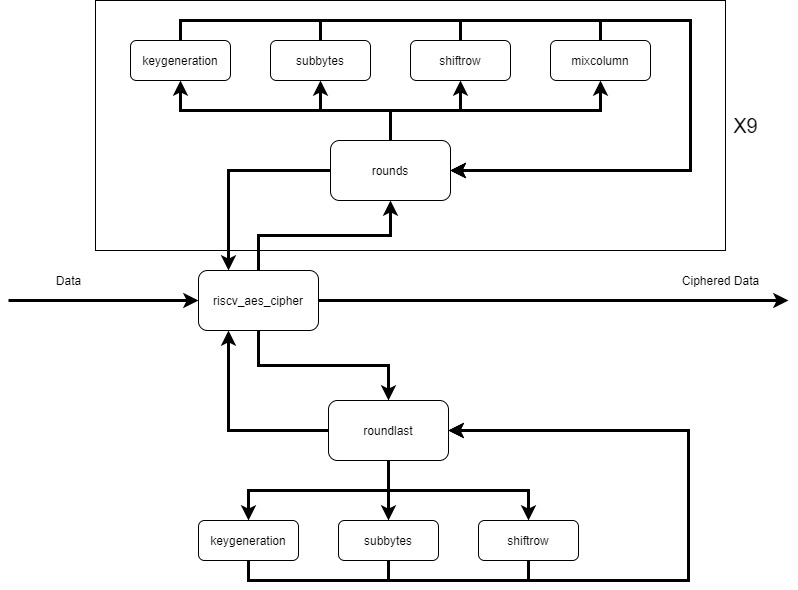
keygeneration

subbytes

shiftrow

mixcolumn

Each of the modules represent each one of the AES processing steps that occurs in every round (except the last round, which running only the first three steps).



**Top level**

The top level of the AES engine inside the RISC-V core is divided into two modules:

* “riscv\_aes\_registers”:

Functions as AES instruction decoder and AES register file. It contains 4 data registers, 4 key registers and 1 WB 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.

* “riscv\_aes\_wb”:

Functions as write-back module, it contains state machine, which halts the RISC-V pipe for 4 cycles each time it receives 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 that stored in the write-back register).

**Interface**

In order to run AES encryption command, 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 “normal” register:

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 “normal” register:

AES KEY k1, t3

* AES MEM:

Writing an address to the AES WB register, which holds the write-back address. It receives as parameter RISC-V “normal” register:

AES MEM t5

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

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, starting at the address stored in the write-back address).

**Alternative solutions**

There was 3 optional solution on the table:

1. Encryption command:

In this solution, the mechanism of the encryption will be as follows:

1. Storing the data address inside register.
2. Storing the key address inside register.
3. Calling the AES command, which will load the 4 data and 4 key registers from the memory, run the AES algorithm, and store the ciphered data back to the non-ciphered data address.

Advantages:

* + Less special commands (only one new command).
  + Easy to use.
  + Can store one register (instead of 4 at a time).

Disadvantages:

* + Not secure – non-ciphered data must be written to memory before running the command.
  + Wasteful in terms of time and power - requires 8 loads from the memory and 8 stores to the memory.

1. Encryption on the fly:

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).
  2. Calling the AES command (equivalent to “Store” command)
  3. 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.

Advantages:

* + Secure – the data will be ciphered before storing it in the memory.
  + Can store one register (instead of 4 at a time).

Disadvantages:

* + Hard to implement.
  + Wasteful in terms of time and power - requires 4 loads from the memory and 4 stores to the memory.

**Pulpenix setup**

There are some shortcut scripts and environment settings that necessary in order to work efficiently with Pulpenix.

Running Pulpenix setup:

> source <absolute\_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**

In order to run synthesis and create “fresh” copy of Pulpenix, you can use the “shortcut” script:

> our\_pulp\_synthesis

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’

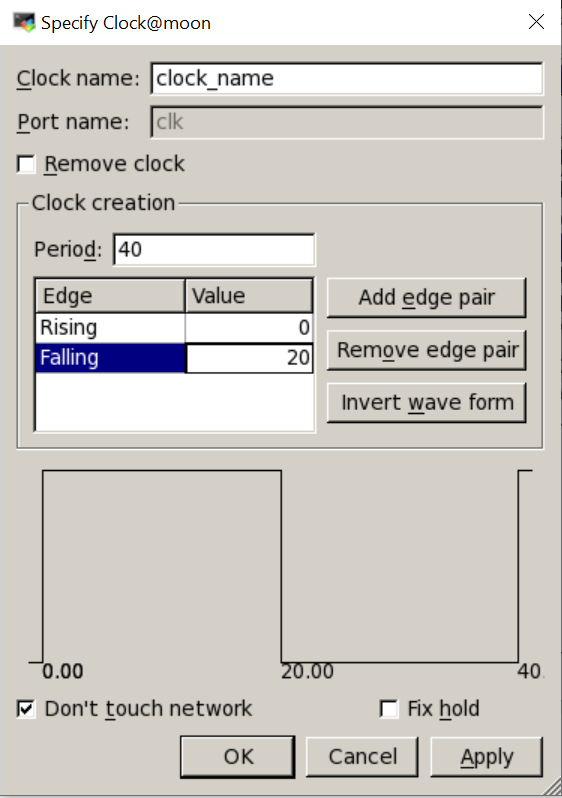
1. Specify a clock:

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 in 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>

pulp\_comp\_app\_noopt <progname>

This step 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:

> our\_wave\_trace

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

> our\_pulp\_run <progname>

$MY\_PULP\_APPS/waves/<name>.shm and the trace file is saved as

$MY\_PULP\_APPS/traces/<name>.log .

**Simvision**

In order to see the wave form 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 wave form, 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 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

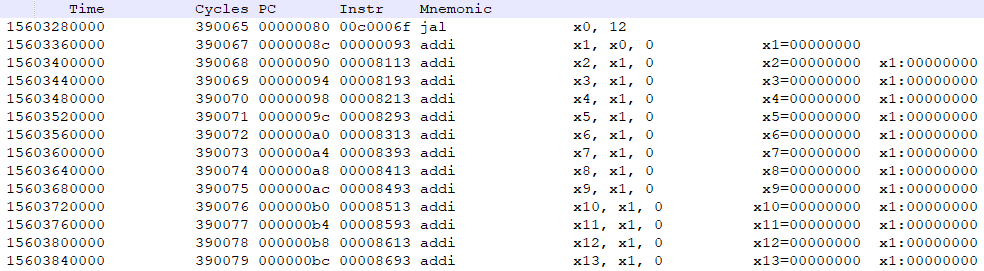
Open

**Trace file**

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

* Cycle - cycle number in the simulation (not necessarily accurate).
* Time - time in the simulation (accurate).
* 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 registers values after the current instruction.

The time column can help finding the required commands in the wave form.

Example for trace file:

1. Waterman Andrew, [Asanović Krste](https://en.wikipedia.org/wiki/Krste_Asanovi%C4%87" \o "Krste Asanović): ["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" \o "Krste Asanović). ["The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA version 2.2"](https://riscv.org/technical/specifications/) [↑](#footnote-ref-3)
4. Pulpino datasheet [↑](#footnote-ref-4)