

**High Performance**

**IPCore**

**RSA 512 bit**

**Data-sheet v.1.0**

**Emilio Castillo Villar  
Javier Castillo Villar**



## Content Index

1. Introduction:.....	3
2. Core Interface:.....	3
3. On constants and input format:.....	5
3.1 Data format:.....	5
3.2 Calculating constants:.....	5
4. Required Memory Cores:.....	6
4.1 Mem_b:.....	6
4.2 res_out_fifo:.....	7
4.3 Fifo_512_bram:.....	8
4.4 Fifo_256_feedback:.....	9

## 1. Introduction:

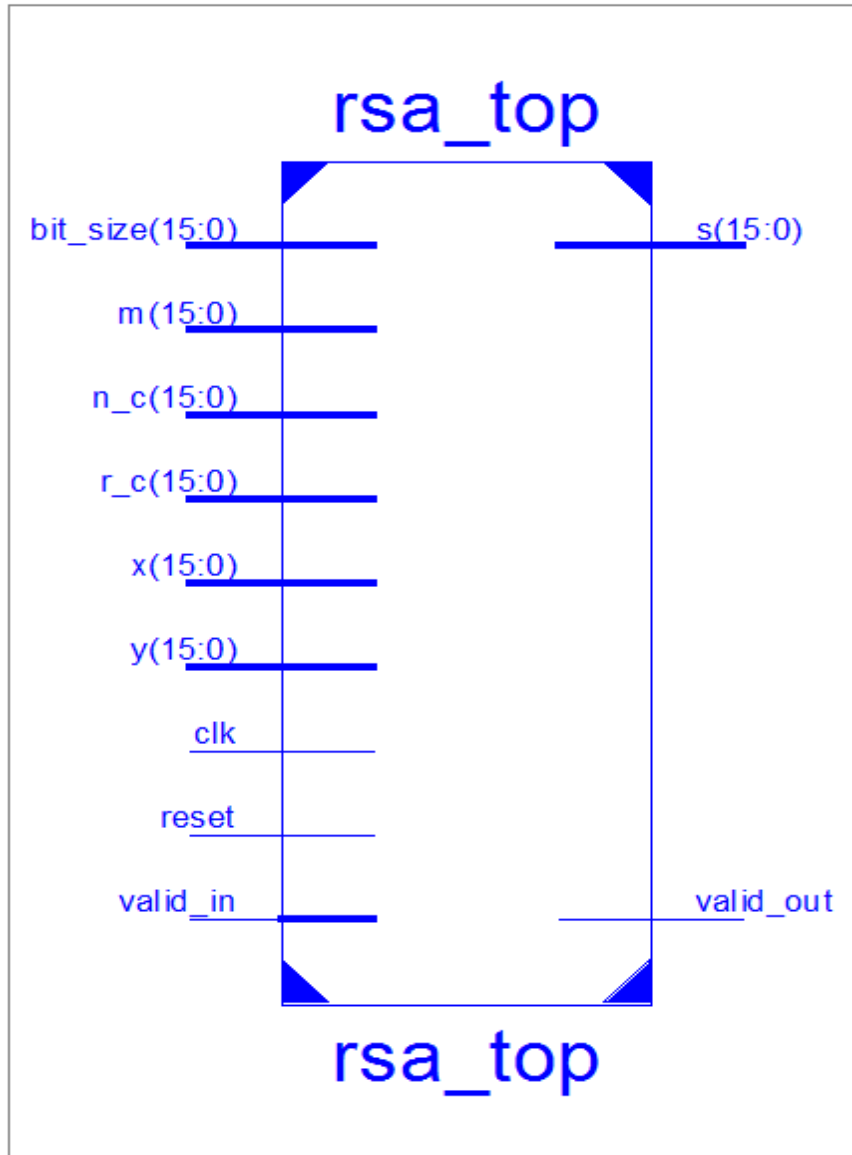
Here, we present the first available open-source 512 bit RSA core. This is a reduced version of a full FIPS Certified 512-4096 capable RSA Crypto-core.

The full version supports all key sizes (512, 1024, 2048, 4096) and includes a complete testbench. It can reach more than 150 operations per second with a 1024 key size in a Spartan 6 FPGA and more than 200 in a Virtex 6.

The core fits in a XC6SLX25T, which makes it a nice solution for mobile devices needing RSA acceleration.

For more information about this core contact [jcastillo@opencores.org](mailto:jcastillo@opencores.org)

## 2. Core Interface:



The core performs a classical modular exponentiation  $x^y \bmod m$  the data needed is the following:

1. **bit\_size** : this is a constant value which specifies the bit length of value  $y$ , it is necessary in order to perform private-key exponentiation (The usual value of this field will be “512”) or public-key exponentiation (It can vary between a few bits). It can be calculated as  $\log_2(y)$  being  $y$  the key used to cipher.

2. **X**: This is the plain text input which will be ciphered, in section 3 we will detail the data format.

## High Performance RSA 512 bit IPCore

3.**Y**: This is the key input, which will be used to cipher X, in section 3 we will detail the data format.

4.**M**: This is the module m input , in section 3 we will detail the data format.

5.**n\_c** : this input is a 32 bit constant needed by the ciphering algorithm in order to achieve a high performance, it can be obtained as we detail in section 3.

6.**r\_c**: this is a 512 bit length constant needed by the ciphering algorithm in order to achieve a high performance, it can be obtained as we detail in section 3.

7.**valid\_in**: should be active high (logical value of 1 as long as the data is being introduced).

8.**S**: This port is the data output of the exponentiation.

9.**valid\_out**: as it's name says, it indicates when the values on S are valid.

Also don't forget to read section 4 where we explain how to generate the needed memory cores.

### 3. On constants and input format:

#### 3.1 Data format:

The values X, Y, M and r\_c needed to be coded as it follows.

Given a 512 bit number  $X = a_{31} a_{30} a_{29} \dots a_2 a_1 a_0$  with  $a_i$  being a 16 bit length word

It shall be introduced in the core starting by the least significant 16 bit word.

This means, in the first clock cycle we will input  $a_0$  in the second  $a_1$  and continue until  $a_{31}$  is reached

This example:

```
8393638f8410333522e0a9d9ff0746878c3b209d55274c7c97d11b815e4ed8305363b4c27f
20525c99fe3605485cc4c595ab0f3dc416f16b94cce4662025490
```

Will follow as, 5490 6202 ce46 ....

**The output S will follow the same format**

#### 3.2 Calculating constants:

The constants n\_c and r\_c are used to accelerate the exponentiation and depends only of the module m, this mean that if you intend to use the core with a few already known set of keys you can

## High Performance RSA 512 bit IPCore

pre-calculate this constants with the "constant\_gen.c" code included in the project.

Given a modulus  $m$  with 32 16-bit length words (this is 512 bit). We can calculate the Montgomery constant  $r$  as  $2^{(16*(32+1))}$

$-n\_c$  can be calculated as follows,  $n_c = (-m)^{-1} \bmod r$ , this is the modular inverse of  $-m$  module  $r$ , please note that you only need the least significant 16-bit!!

$-r\_c$  is  $r^2 \bmod m$  which will result in a maximum of 512 bit number.

Should you want to use our code to generate this constants, you have to edit the .c file and replace the  
`mpz_init_set_str(m, "8de7066f67be16fcacd05d319b6729cd85fe698c07cec504776146  
eb7a041d9e3cacbf0fcd86441981c0083eed1f8f1b18393f0b186e47ce1b7b4981417b491"  
, 16);`

With your own  $m$  value and compile it with "`gcc constant_gen.c -lgmp`" maybe you will have to install the gnu multiprecision library available at <http://gmplib.org/>

## 4. Required Memory Cores:

### 4.1 Mem\_b:

A Single port Ram Core must be generated with name Mem\_b

```
component Mem_b
  port (
    clka: IN std_logic;
    wea: IN std_logic_VECTOR(0 downto 0);
    addra: IN std_logic_VECTOR(5 downto 0);
    dina: IN std_logic_VECTOR(15 downto 0);
    douta: OUT std_logic_VECTOR(15 downto 0));
end component;
```

With length parameters as follows:

## High Performance RSA 512 bit IPCore

Port A Options

Memory Size

Write Width: 16 Range: 1..1152      Read Width: 16

Write Depth: 40 Range: 2..9011200      Read Depth: 40

Operating Mode

Write First

Read First

No Change

Enable

Always Enabled

Use ENA Pin

### 4.2 res\_out\_fifo:

```
component res_out_fifo
  port (
    clk: IN std_logic;
    rst: IN std_logic;
    din: IN std_logic_VECTOR(31 downto 0);
    wr_en: IN std_logic;
    rd_en: IN std_logic;
    dout: OUT std_logic_VECTOR(31 downto 0);
    full: OUT std_logic;
    empty: OUT std_logic);
end component;
```

## High Performance RSA 512 bit IPCore

Standard FIFO  
 First-Word Fall-Through

**Built-in FIFO Options**

The frequency relationship of WR\_CLK and RD\_CLK MUST be specified to generate the correct implementation.

Read Clock Frequency (MHz)  Range: 1..1000  
Write Clock Frequency (MHz)  Range: 1..1000

**Data Port Parameters**

Write Width  Range: 1,2,3..1024  
Write Depth  Actual Write Depth: 64  
Read Width   
Read Depth  Actual Read Depth: 64

**Implementation Options**

Enable ECC  
 Use Embedded Registers in BRAM or FIFO (when possible)

Read Latency (From Rising Edge of Read Clock): 1

### 4.3 Fifo\_512\_bram:

component fifo\_512\_bram

```
port (  
  clk: IN std_logic;  
  rst: IN std_logic;  
  din: IN std_logic_VECTOR(15 downto 0);  
  wr_en: IN std_logic;  
  rd_en: IN std_logic;  
  dout: OUT std_logic_VECTOR(15 downto 0);  
  full: OUT std_logic;  
  empty: OUT std_logic);
```



## High Performance RSA 512 bit IPCore

END component;

The screenshot shows the configuration interface for the High Performance RSA 512 bit IPCore. It is divided into three main sections:

- Read Mode:** Contains two radio buttons: "Standard FIFO" (selected) and "First-Word Fall-Through".
- Built-in FIFO Options:** Includes a warning: "The frequency relationship of WR\_CLK and RD\_CLK MUST be specified to generate the correct implementation." Below this are two input fields: "Read Clock Frequency (MHz)" and "Write Clock Frequency (MHz)", both set to "1" with a range of "1..1000".
- Data Port Parameters:** Contains four input fields: "Write Width" (16, Range: 1,2,3..1024), "Write Depth" (64, Actual Write Depth: 64), "Read Width" (16), and "Read Depth" (64, Actual Read Depth: 64).

### 4.4 Fifo\_256\_feedback:

component fifo\_256\_feedback

```
port (  
  clk: IN std_logic;  
  rst: IN std_logic;  
  din: IN std_logic_VECTOR(48 downto 0);  
  wr_en: IN std_logic;  
  rd_en: IN std_logic;  
  dout: OUT std_logic_VECTOR(48 downto 0);  
  full: OUT std_logic;  
  empty: OUT std_logic);
```

## High Performance RSA 512 bit IPCore

END component;

Standard FIFO  
 First-Word Fall-Through

---

**Built-in FIFO Options**

The frequency relationship of WR\_CLK and RD\_CLK MUST be specified to generate the correct implementation.

Read Clock Frequency (MHz)  Range: 1..1000  
Write Clock Frequency (MHz)  Range: 1..1000

---

**Data Port Parameters**

Write Width  Range: 1,2,3..1024  
Write Depth  Actual Write Depth: 32  
Read Width   
Read Depth  Actual Read Depth: 32