High Performance

IPCore

RSA 512 bit

Data-sheet v.1.1

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 an early prototype version of a full FIPS Certified 512-4096 capable RSA Crypto-core which will be on sale soon. The version provided, has not the same performance than the final product since it was a proof of concept that we decided to release to the community in order to help small projects which need RSA ciphering.

2. Core Interface:

The core performs a classical modular exponentiation \( x^y \mod m \) 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.
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. \( r_c \): 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.
6. start_in: active it when load the first 16 bits of \( m \). After 6 cycles you can activate valid_in to insert the rest of the data. See testbench.
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.
High Performance RSA 512 bit IPCore

Given a 512 bit number \( X = a_31a_{30}a_{29} \ldots a_2a_1a_0 \) 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:

\[
8393638f8410333522e0a9d9ff0746878c3b209d55274c7c97d11b815e4ed8305363b4c27f20525c99fe3605485cc4c595ab0f3dc416f16b94ccee4662025490
\]

Will follow as, 5490 6202 ce46 ….

The output \( S \) will follow the same format.

3.2 Calculating constants:

The constant \( r_c \) is 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 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

\[
2^{(16 \times (32 + 1))}
\]

\(-r_c \) is \( r^2 \mod m \) 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

```c
mpz_init_set_str(m,"8de7066f67be16fcac05d319b6729cd85fe698c07cec504776146eb7a04d9e3cacbf0fcd86441981c0083ed1f8f1b18393f0b186e47ce1b7b4981417b491",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

```vhdl
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:

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;
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);
END component;
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);
END component;