www.opencores.org

Modular Simultaneous Exponentiation IP Core Specification

Acknowledgments

This project is maintained by the DraMCo research group[1] of KAHO Sint-Lieven[2], part of the KU Leuven association[3]. The base design for this IP core is written by Geoffrey Ottoy, member of the DraMCo research group. Further adjustments have been made by Jonas De Craene

[1]`http://www.dramco.be/`
[2]`http://www.kahosl.be/`
[3]`http://associatie.kuleuven.be/`

Document Revision History

History

Revision	Date	By	Description
0	November 2012	JDC	First draft of this specification
1.0	November 2012	JDC	Added sections "Acknowledgement" and "Performance and resource usage" as well as different fonts for *variables* and `signal_names`
1.1	November 2012	GO	Added this "Document Revision History". Made several small changes in layout and formulation.

Author info

GO: Geoffrey Ottoy
DraMCo research group
`geoffrey.ottoy@kahosl.be`

JDC: Jonas De Craene
KAHO Sint-Lieven
`JonasDC@opencores.org`

Contents

Acknowledgments **iii**

Document Revision History **iv**

1 Introduction **1**

2 Architecture **2**
2.1 Block diagram 2
2.2 Exponentiation core 3
2.2.1 Multiplier 3
2.2.2 Operand RAM and exponent FIFO 8
2.2.3 Control unit 8
2.2.4 IO ports and memory map 9
2.3 Bus interface 10

3 Operation **11**
3.1 Pipeline operation 11
3.2 Modular Simultaneous exponentiation operations 12
3.3 Core operation steps 12
3.3.1 Single Montgomery multiplication 12
3.3.2 Modular simultaneous exponentiation 13

4 PLB interface **14**
4.1 Structure 14
4.2 Parameters 15
4.3 IO ports 17
4.4 Registers 18
4.4.1 Control register (offset = 0x0000) 19
4.4.2 Software reset register (offset = 0x0100) 20
4.4.3 Global interrupt enable register (offset = 0x021C) 20
4.4.4 Interrupt status register (offset = 0x0220) 20
4.4.5 interrupt enable register (offset = 0x0228) 20
4.5 Interfacing the core's RAM 21
4.6 Handling interrupts 21

5 Performance and resource usage **22**

Chapter 1

Introduction

The Modular Simultaneous Exponentiation core is a flexible hardware design to support modular simultaneous exponentiations in embedded systems. It is able to compute a double exponentiation as given by (1.1)

$$g_0^{e_0} \cdot g_1^{e_1} \bmod m \tag{1.1}$$

where:

$$\begin{aligned}
g_0 &= \left(g0_{n-1}, \cdots, g0_1, g0_0\right)_2 && \text{with } n \text{ being the number of bits of the base operands}\\
g_1 &= \left(g1_{n-1}, \cdots, g1_1, g1_0\right)_2 &&\\
m &= \left(m_{n-1}, \cdots, m_1, m_0\right)_2 &&\\
e_0 &= \left(e0_{t-1}, \cdots, e0_1, e0_0\right)_2 && \text{with } t \text{ being the number of bits of the exponents}\\
e_1 &= \left(e1_{t-1}, \cdots, e1_1, e1_0\right)_2 &&
\end{aligned}$$

This operation is commonly used in anonymous credential and authentication cryptosystems like DSA [1], Idemix [2], etc.. For this reason the core is designed with the use of large base operands in mind (n=512, 1024, 1536 bit and more..). The hardware is optimized for these simultaneous exponentiations, but also supports single base exponentiations and single Montgomery multiplications. Flexibility is offered to the user by providing the possibility to split the multiplier pipeline into 2 smaller parts, so that in total 3 different base operand lengths can be supported. The length of the exponents can be chosen freely[3]

[1]FIPS-186-3, the third and current revision to the official DSA specification:
`http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf`

[2]IBM Idemix project website: `https://www.zurich.ibm.com/security/idemix/`

[3]The controlling software is responsible for loading in the desired number of exponent bits into the core's exponent FIFO

Chapter 2

Architecture

2.1 Block diagram

The architecture for the full IP core is shown in the Figure 2.1. It consists of 2 major parts, the actual exponentiation core (`mod_sim_exp_core` entity) with a bus interface wrapped around it. In the following sections these different blocks are described in detail.

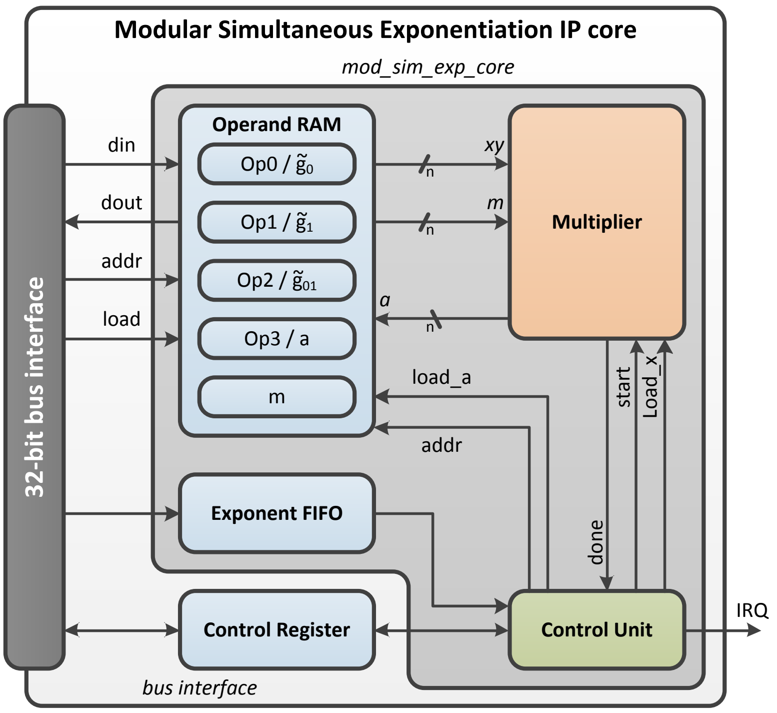

Figure 2.1: Block diagram of the Modular Simultaneous Exponentiation IP core

2.2 Exponentiation core

The exponentiation core (`mod_sim_exp_core` entity) is the top level of the modular simultaneous exponentiation core. It is made up by 4 main blocks (Figure 2.2):

- a pipelined Montgomery multiplier as the main processing unit
- RAM to store the operands and the modulus
- a FIFO to store the exponents
- a control unit which controls the multiplier for the exponentiation and multiplication operations

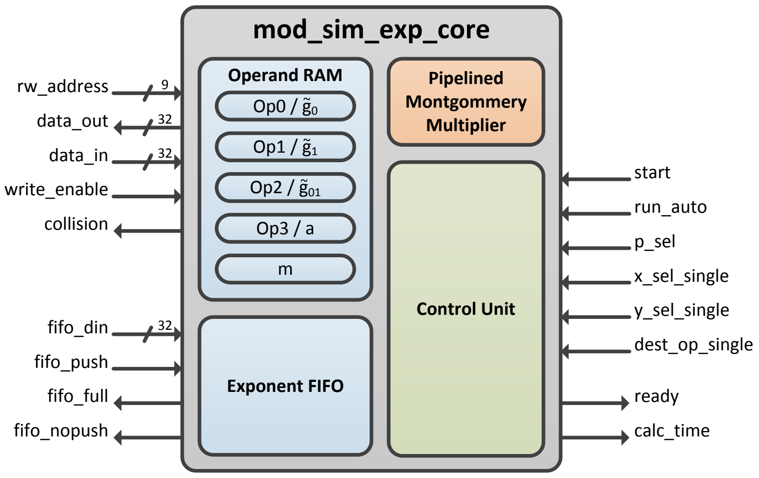

Figure 2.2: `mod_sim_exp_core` structure

2.2.1 Multiplier

The kernel of this design is a pipelined Montgomery multiplier. A Montgomery multiplication[1] allows efficient implementation of a modular multiplication without explicitly carrying out the classical modular reduction step. Right-shift operations ensure that the length of the (intermediate) results does not exceed $n+1$ bits. The result of a Montgomery multiplication is given by (2.1):

$$r = x \cdot y \cdot R^{-1} \bmod m \qquad \text{with } R = 2^n \tag{2.1}$$

For the structure of the multiplier, the work of *Nedjah and Mourelle*[2] is used as a basis. They show that for large operands (>512 bits) the *time* × *area* product is minimal when a systolic implementation is used. This construction is composed of cells that each compute a bit of the (intermediate) result.
Because a fully unrolled two-dimensional systolic implementation would require too many resources, a systolic array (one-dimensional) implementation is chosen. This implies that the intermediate results are fed back to the same same array of cells through a register. A shift register will shift-in a bit of the x operand for every step in the calculation (figure 2.3). When multiplication is completed, a final check is made to ensure the result is smaller than the modulus. If not, a final reduction with m is necessary.
Note: For this implementation the modulus m has to be uneven to obtain a correct result. However, we can assume that for cryptographic applications, this is the case.

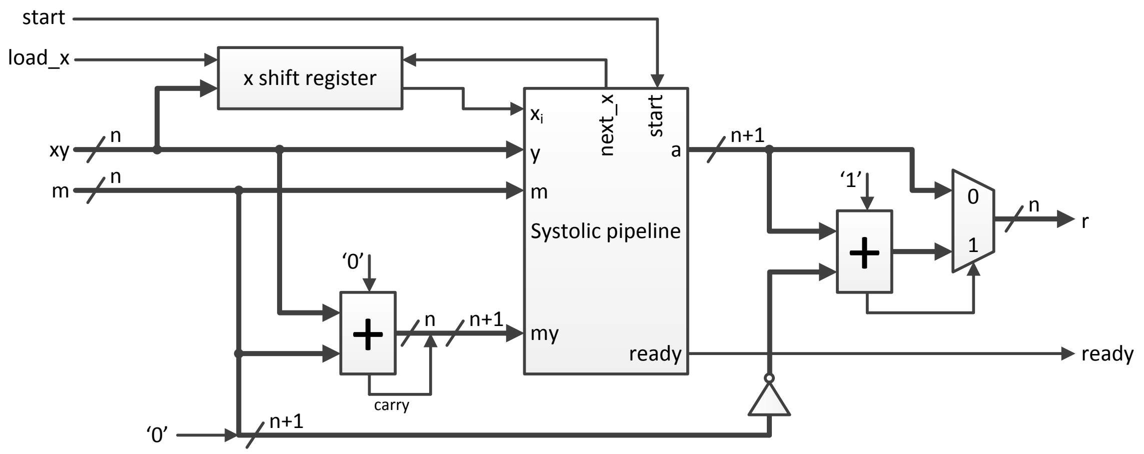

Figure 2.3: Multiplier structure. For clarification the *my* adder and reduction logic are depicted separately, whereas in practice they are internal parts of the stages. (See Figure 2.4)

Stage and pipeline structure

The Montgomery algorithm uses a series of additions and right shifts to obtain the desired result. The main disadvantage is the carry propagation in the adder, and therefore a pipelined version is used. The length of the operands (n) and the number of pipeline stages can be chosen before synthesis. The user has the option to split the pipeline into 2 smaller parts so there are 3 operand lengths available during runtime[1].
The stages and first and last cell logic design are presented in Figure 2.4. Each stage takes in a part of the modulus m and y operand and for each step of the multiplication, a bit of the x operand is fed to the pipeline (together with the generated q signal), starting with the Least Significant Bit. The systolic array cells need the modulus m, the operand y and the sum $m+y$ as an input. The result from the cells is latched into a register, and then passed back to the systolic cells for the next bit of x. During this pass the right shift operation is implemented. Each stage thus needs the least significant bit from the next stage to calculate the next step. Final reduction logic is also present in the stages for when the multiplication is complete.
An example of the standard pipeline structure is presented in Figure 2.5. It is constructed using stages with a predefined width. The first cell logic processes the first bit of the m and y operand and generates the q signal. The last cell logic finishes the reduction and selects the correct result. For operation of this pipeline, it is clear that each stage can only compute a step every 2 clock cycles. This is because the stages rely on the result of the next stage.
In Figure 2.6 an example pipeline design is drawn for a split pipeline. All multiplexers on this figure are controlled by the pipeline select signal (`p_sel`). During runtime the user can choose which part of the pipeline is used, the lower or higher part or the full pipeline.

[1] e.g. a total pipeline length of 1536 bit split into a part of 512 bit and a part of 1024 bit

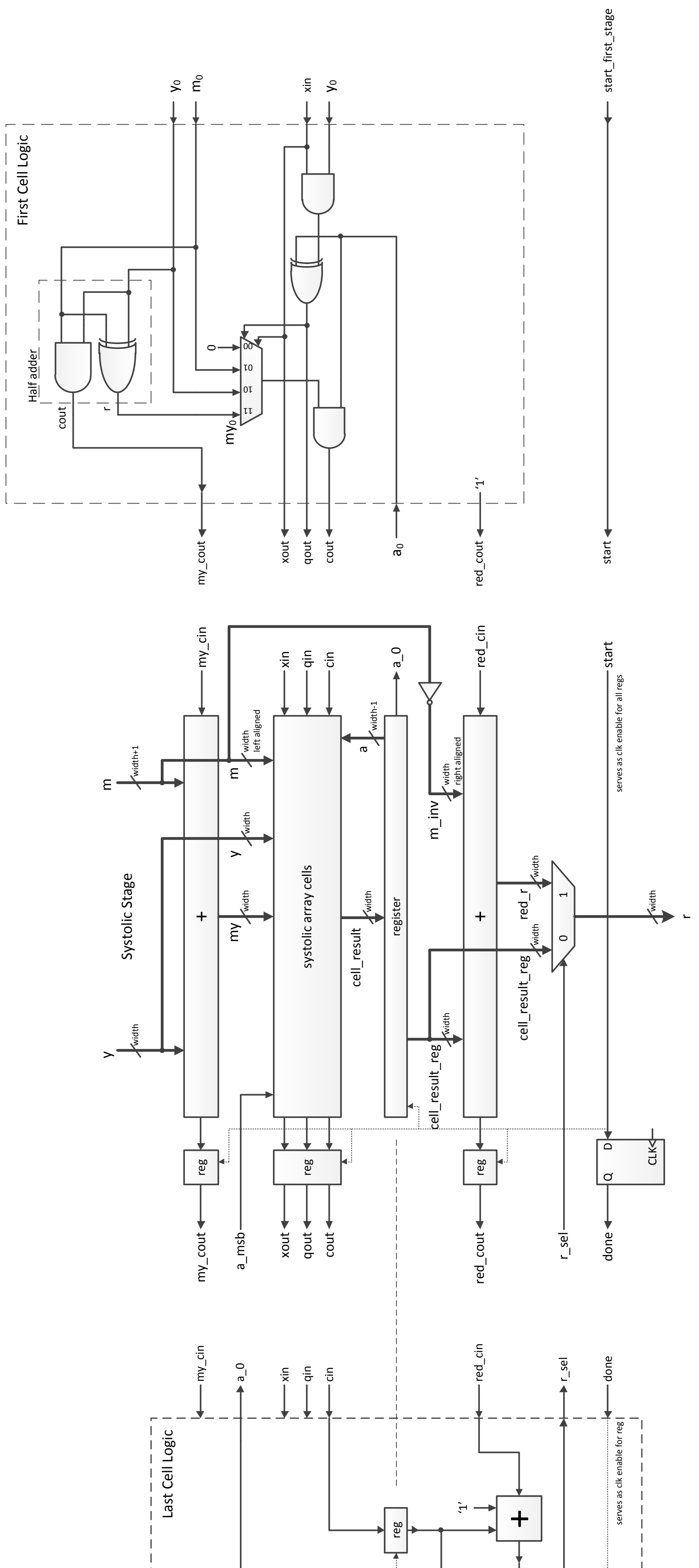

Figure 2.4: Pipeline stage and first and last cell logic

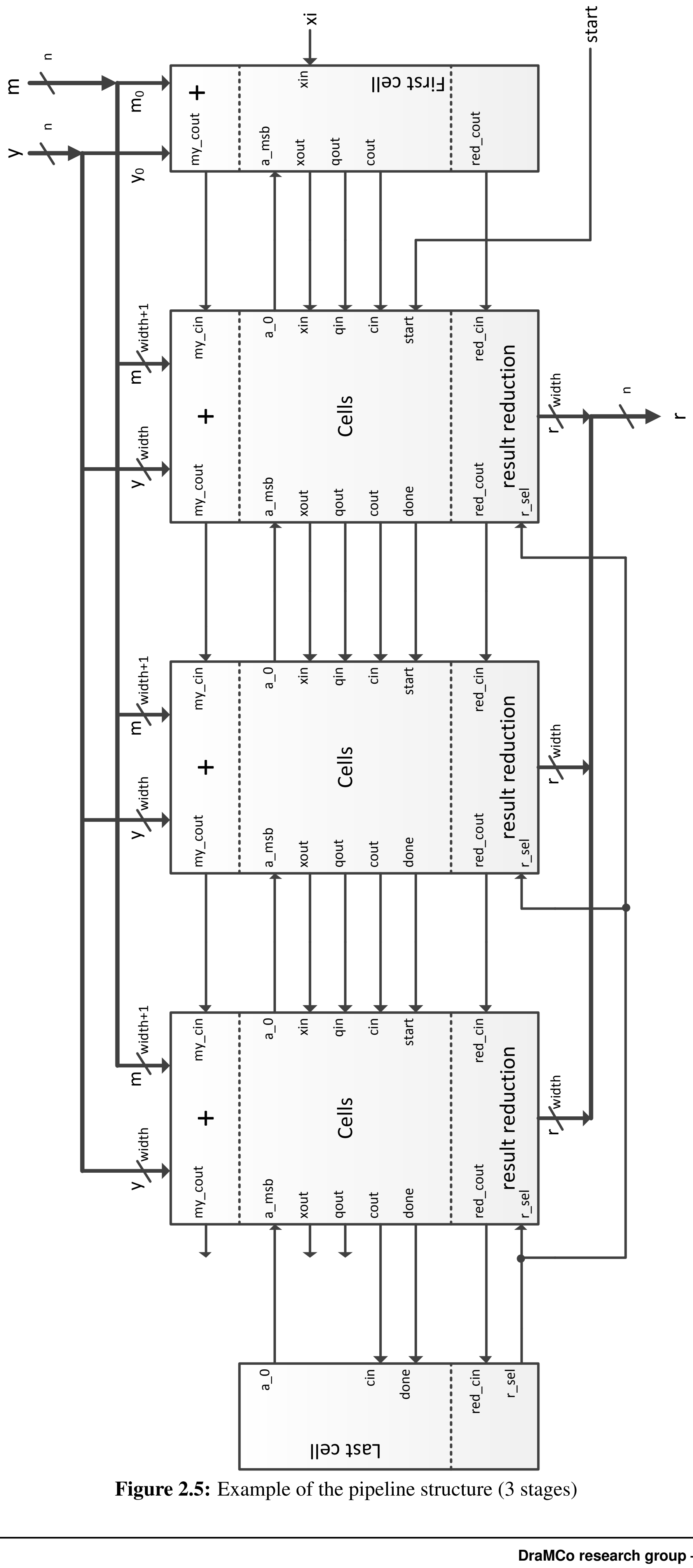

Figure 2.5: Example of the pipeline structure (3 stages)

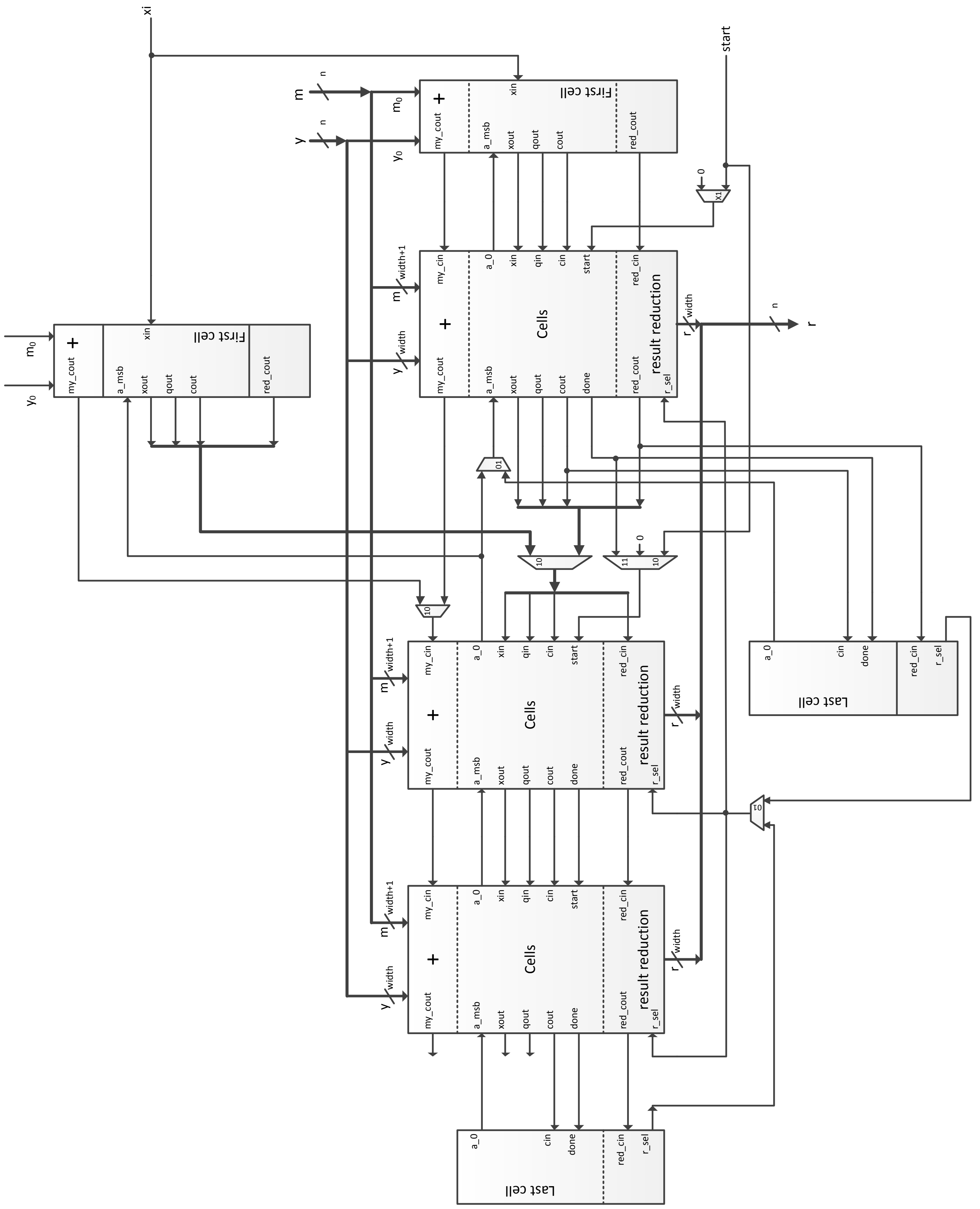

Figure 2.6: Example of a split pipeline (1+2 stages)

2.2.2 Operand RAM and exponent FIFO

In the core's RAM there is space for 4 operands and 1 modulus. Currently this is instantiated using Xilinx BRAM primitives, so there is a fixed RAM of 4x1536 bit for the operands + 1536 bit for the modulus available. If using a split pipeline, it is important that operands for the higher part of the pipeline are loaded into the RAM with preceding zero's for the lower bits of the pipeline. To store the exponents there is a FIFO of 32 bit wide and 512 deep (also a Xilinx primitive, FIFO18E1), so it is able to store 2 exponents of each 8192 bit long. Every 32 bit entry has to be pushed in as 16 bit of e_0 for the lower part [15:0] and 16 bit of e_1 for the higher part [31:16]. Starting with the least significant word and ending with the most significant word of the exponents.

2.2.3 Control unit

The control unit loads in the operands and has full control over the multiplier. For single multiplications, it latches in the x operand, then places the y operand on the bus and starts the multiplier. In case of an exponentiation, the FIFO is emptied while the necessary single multiplications are performed. When the computation is done, the ready signal is asserted to notify the system.

2.2.4 IO ports and memory map

The `mod_sim_exp_core` IO ports

Port	Width	Direction	Description
`clk`	1	in	core clock input
`reset`	1	in	reset signal (active high) resets the pipeline, fifo and control logic
operand memory interface			
`rw_address`	9	in	operand memory read/write address (structure descibed below)
`data_out`	32	out	operand data out (0 is lsb)
`data_in`	32	in	operand data in (0 is lsb)
`write_enable`	1	in	write enable signal, latches `data_in` to operand RAM
`collision`	1	out	collision output, asserts on a write error
exponent FIFO interface			
`fifo_din`	32	in	FIFO data in, bits [31:16] for e_1 operand and bits [15:0] for e_0 operand
`fifo_push`	1	in	push `fifo_din` into the FIFO
`fifo_nopush`	1	out	flag to indicate if there was an error pushing the word to the FIFO
`fifo_full`	1	out	flag to indicate the FIFO is full
control signals			
`x_sel_single`	2	in	selection for x operand source during single multiplication
`y_sel_single`	2	in	selection for y operand source during single multiplication
`dest_op_single`	2	in	selection for the result destination operand for single multiplication
`p_sel`	2	in	specifies which pipeline part to use for exponentiation / multiplication. “01” : use lower pipeline part “10” : use higher pipeline part “11” : use full pipeline
`exp_m`	1	in	core operation mode. “0” for single multiplications and “1” for exponentiations
`start`	1	in	start the calculation for current mode
`ready`	1	out	indicates the multiplication/exponentiation is done
`calc_time`	1	out	is high during a multiplication, indicator for used calculation time

The following figure describes the structure of the Operand RAM memory, for every operand there is a space of 2048 bits reserved.

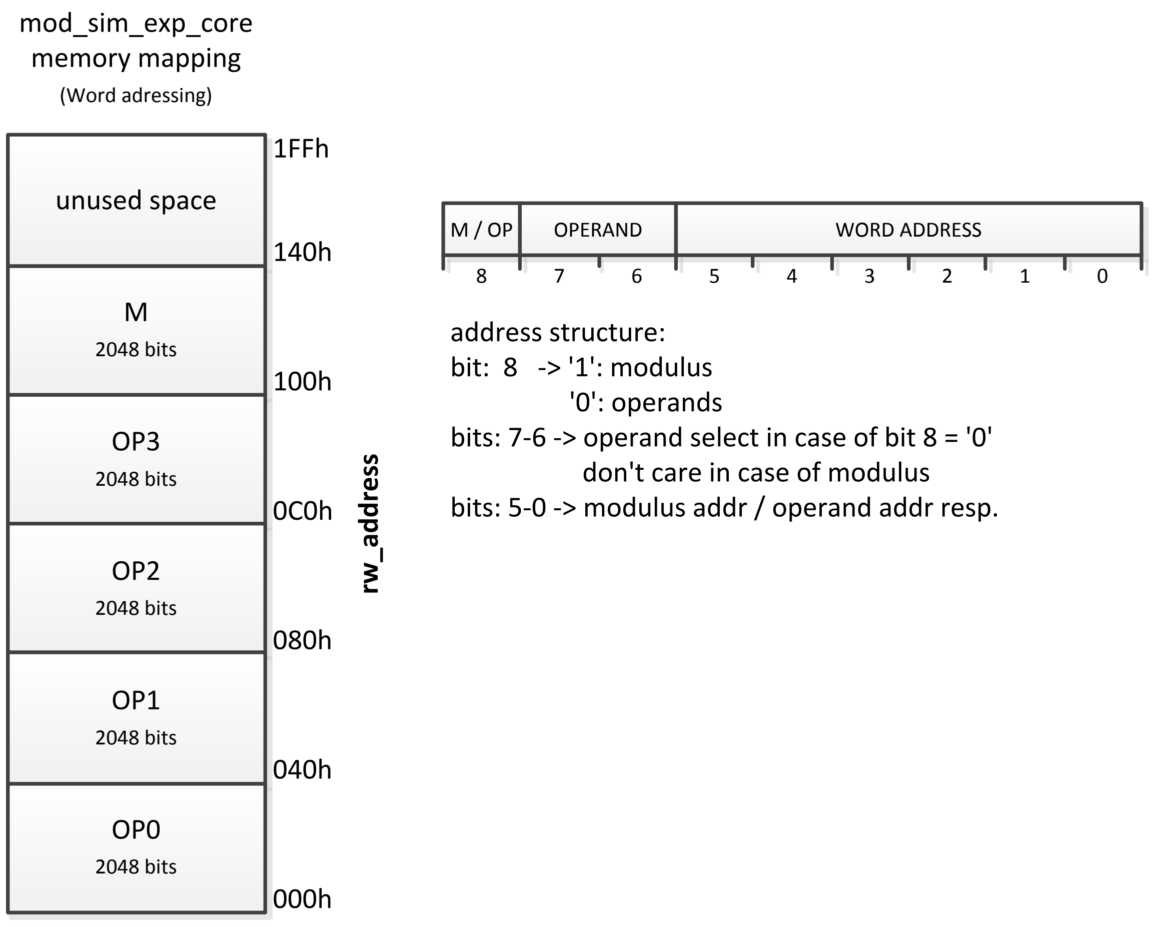

Figure 2.7: RAM structure of the exponentiation core

2.3 Bus interface

The bus interface implements the register necessary for the control unit inputs to the `mod_sim_exp_core` entity. It also maps the memory to the required bus and connects the interrupt signals. The embedded processor then has full control over the core.

Chapter 3

Operation

3.1 Pipeline operation

The operation of the pipeline is shown in Figure 3.1. One can see that the stages are started every 2 clock cycles (τ_c is the core clock period). This is needed because the least significant bit of the next stage result is needed. Every stage has to run n (the width of the operands) times for the multiplication to be complete.

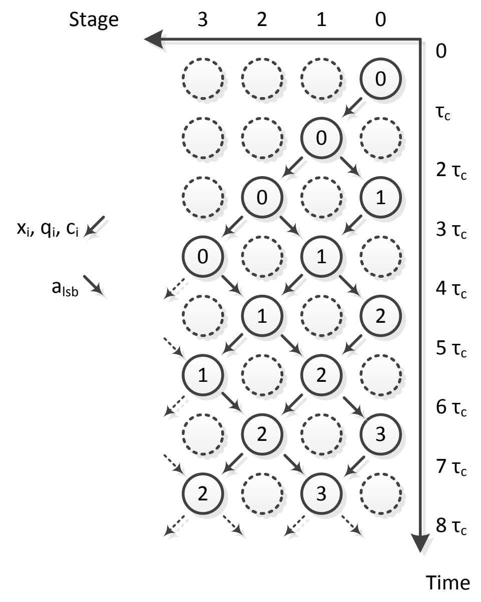

Figure 3.1: Pipeline operation: Each circle represents an active stage. The number indicates how much times that stage has run. Dotted line contours indicate the stage is inactive.

For performing one Montgomery multiplication using this core, the total computation time T_m for an n-bit operand with a k-stage pipeline is given by (3.1).

$$T_m = [k + 2(n-1)]\,\tau_c \qquad (3.1)$$

3.2 Modular Simultaneous exponentiation operations

Exponentiations are calculated with Algorithm 1 which uses the Montgomery multiplier as the main computation step. It uses the principle of a square-and-multiply algorithm to calculate an exponentiation with 2 bases.

Algorithm 1 Montgomery simultaneous exponentiation

```
Input: $g_0, g_1, e_0 = (e_{0_{t-1}} \cdots e_{0_0})_2, e_1 = (e_{0_{t-1}} \cdots e_{0_0})_2, R^2 \bmod m, m$
Output: $g_0^{e_0} \cdot g_1^{e_1} \bmod m$
 1: $\tilde{g}_0 := \mathrm{Mont}(g_0, R^2), \tilde{g}_1 := \mathrm{Mont}(g_1, R^2), \tilde{g}_{01} := \mathrm{Mont}(\tilde{g}_0, \tilde{g}_1)$
 2: $a := \mathrm{Mont}(R^2, 1)$        ▷ This is the same as $a := R \bmod m$.
 3: for $i \leftarrow (t-1)$ downto 0 do
 4:     $a := \mathrm{Mont}(a, a)$
 5:     switch $e_{1_i}, e_{0_i}$
 6:         $0, 1 : a := \mathrm{Mont}(a, \tilde{g}_0)$
 7:         $1, 0 : a := \mathrm{Mont}(a, \tilde{g}_1)$
 8:         $1, 1 : a := \mathrm{Mont}(a, \tilde{g}_{01})$
 9: $a := \mathrm{Mont}(a, 1)$
10: return $a$
```

It can be seen that the algorithm requires $R^2 \bmod m$ which is $2^{2n} \bmod m$. We assume $R^2 \bmod m$ can be provided or pre-computed. The for loop in the algorithm is executed by the control logic of the core. Apart from this, a few pre- and one post-calculations have to be performed.
The computation time for an exponentiation depends on the number of zero's in the exponents, from Algorithm 1 one can see that if both exponent bits are zero at a time, no multiplication has to be performed. Thus reducing the total time. The average computation time for a modular simultaneous exponentiation, with n-bit base operands and t-bit exponents is given by (3.2).

$$T_{se} = \frac{7}{4} t \cdot T_m = \frac{7}{4} t \cdot [k + 2(n-1)]\tau_c \tag{3.2}$$

For single base exponentiations, i.e. 1 exponent is equal to zero, the average exponentiation time is given by (3.3).

$$T_e = \frac{3}{2} t \cdot T_m = \frac{3}{2} t \cdot [k + 2(n-1)]\tau_c \tag{3.3}$$

The formulas (3.2) and (3.3) given here are only the theoretical average time for an exponentiation, excluding the pre- and post-computations.

3.3 Core operation steps

The core can operate in 2 modes, multiplication or exponentiation mode. The steps required to do one of these actions are described here.

3.3.1 Single Montgomery multiplication

The following steps are needed for a single Montgomery multiplication:

1. load the modulus to the RAM using the 32 bit bus
2. load the desired x and y operands into any 2 locations of the operand RAM using the 32 bit bus.
3. select the correct input operands for the multiplier using `x_sel_single` and `y_sel_single`
4. select the result destination operand using `result_dest_op`
5. set `exp/m` = ‘0’ to select multiplication mode
6. set `p_sel` to choose which pipeline part you will use

7. generate a start pulse for the core
8. wait until interrupt is received and read out result in selected operand

Note: this computation gives a result $r = x \cdot y \cdot R^{-1} \bmod m$. If the actual product of x and y is desired, a final Montgomery multiplication of the result with R^2 is needed.

3.3.2 Modular simultaneous exponentiation

The core requires $\tilde{g}_0$, $\tilde{g}_0$, $\tilde{g}_{01}$ and a to be loaded into the correct operand spaces before starting the exponentiation. These parameters are calculated using single Montgomery multiplications as follows:

$$\begin{aligned}
\tilde{g}_0 &= Mont(g_0, R^2) & &= g_0 \cdot R \bmod m & &\text{in operand 0}\\
\tilde{g}_1 &= Mont(g_1, R^2) & &= g_1 \cdot R \bmod m & &\text{in operand 1}\\
\tilde{g}_{01} &= Mont(\tilde{g}_0, \tilde{g}_1) & &= g_0 \cdot g_1 \cdot R \bmod m & &\text{in operand 2}\\
a &= Mont(R^2, 1) & &= R \bmod m & &\text{in operand 3}
\end{aligned}$$

When the exponentiation is done, a final multiplication has to be started by the software to multiply a with 1. The steps needed for a full simultaneous exponentiation are:

1. load the modulus to the RAM using the 32 bit bus
2. load the desired g_0, g_1 operands and $R^2 \bmod m$ into the operand RAM using the 32 bit bus.
3. set `p_sel` to choose which pipeline part you will use
4. compute $\tilde{g}_0$ by using a single Montgomery multiplication of g_0 with R^2 and place the result $\tilde{g}_0$ in operand 0.
5. compute $\tilde{g}_1$ by using a single Montgomery multiplication of g_1 with R^2 and place the result $\tilde{g}_1$ in operand 1.
6. compute $\tilde{g}_{01}$ by using a single Montgomery multiplication of $\tilde{g}_0$ with $\tilde{g}_1$ and place the result $\tilde{g}_{01}$ in operand 2.
7. compute a by using a single Montgomery multiplication of R^2 with 1 and place the result a in operand 3.
8. set the core in exponentiation mode (exp/m='1')
9. generate a start pulse for the core
10. wait until interrupt is received
11. perform the post-computation using a single Montgomery multiplication of a(in operand 3) with 1 and read out result

Chapter 4

PLB interface

4.1 Structure

The Processor Local Bus interface for this core is structured as in Figure 4.1. The core acts as a slave to the PLB bus. The PLB v4.6 Slave[3] logic translates the interface to a lower level IP Interconnect Interface (IPIC). This is then used to connect the core internal components to. The user logic contains the exponentiation core and the control register for the core its control inputs and outputs. An internal interrupt controller[4] handles the outgoing interrupt requests and a software reset module is provided to be able to reset the IP core at runtime. This bus interface is created using the “Create or Import Peripheral” wizard from Xilinx Platform Studio.

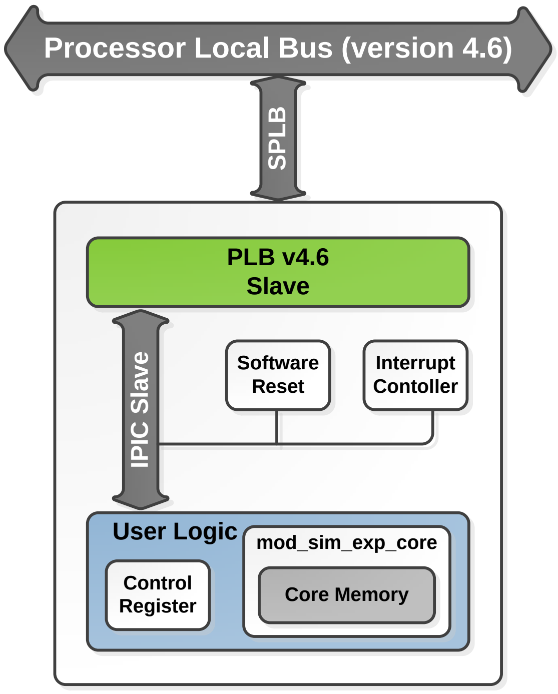

Figure 4.1: PLB IP core structure

4.2 Parameters

This section describes the parameters used to configure the core, only the relevant parameters are discussed. PLB specific parameters are left to the user to configure. The IP core specific parameters and their respective use are listed in the table below.

Name	Description	VHDL Type	Default Value
Memory configuration			
`C_BASEADDR`	base address for the IP core's memory space	std_logic_vector	X"FFFFFFFF"
`C_HIGHADDR`	high address for the IP core's memory space	std_logic_vector	X"00000000"
`C_M_BASEADDR`	base address for the modulus memory space	std_logic_vector	X"FFFFFFFF"
`C_M_HIGHADDR`	high address for the modulus memory space	std_logic_vector	X"00000000"
`C_OP0_BASEADDR`	base address for the operand 0 memory space	std_logic_vector	X"FFFFFFFF"
`C_OP0_HIGHADDR`	high address for the operand 0 memory space	std_logic_vector	X"00000000"
`C_OP1_BASEADDR`	base address for the operand 1 memory space	std_logic_vector	X"FFFFFFFF"
`C_OP1_HIGHADDR`	high address for the operand 1 memory space	std_logic_vector	X"00000000"
`C_OP2_BASEADDR`	base address for the operand 2 memory space	std_logic_vector	X"FFFFFFFF"
`C_OP2_HIGHADDR`	high address for the operand 2 memory space	std_logic_vector	X"00000000"
`C_OP3_BASEADDR`	base address for the operand 3 memory space	std_logic_vector	X"FFFFFFFF"
`C_OP3_HIGHADDR`	high address for the operand 3 memory space	std_logic_vector	X"00000000"
`C_FIFO_BASEADDR`	base address for the FIFO memory space	std_logic_vector	X"FFFFFFFF"
`C_FIFO_HIGHADDR`	high address for the FIFO memory space	std_logic_vector	X"00000000"
Multiplier configuration			
`C_NR_BITS_TOTAL`	total width of the multiplier in bits	integer	1536
`C_NR_STAGES_TOTAL`	total number of stages in the pipeline	integer	96
`C_NR_STAGES_LOW`	number of lower stages in the pipeline, defines the bit-width of the lower pipeline part	integer	32
`C_SPLIT_PIPELINE`	option to split the pipeline in 2 parts	boolean	true

The complete IP core's memory space can be controlled. As can be seen, the operand, modulus and FIFO memory space can be chosen separately from the IP core's memory space which hold the registers for control, software reset and interrupt control. The core's memory space must have a minimum width of 1K byte for all registers to be accessible. For the FIFO memory space, a minimum width of 4 byte is needed, since the FIFO is only 32 bit wide. The memory space width for the operands and the modulus need a minimum width equal to the total multiplier width.

There are 4 parameters to configure the multiplier. These values define the width of the multiplier operands and the number of pipeline stages. If `C_SPLIT_PIPELINE` is false, only operands with a width of

`C_NR_BITS_TOTAL` are valid. Else if `C_SPLIT_PIPELINE` is true, 3 operand widths can be supported:

- the length of the full pipeline ($C\_NR\_BITS\_TOTAL$)
- the length of the lower pipeline ($\frac{C\_NR\_BITS\_TOTAL}{C\_NR\_STAGES\_TOTAL} \cdot C\_NR\_STAGES\_LOW$)
- the length of the higher pipeline ($\frac{C\_NR\_BITS\_TOTAL}{C\_NR\_STAGES\_TOTAL} \cdot (C\_NR\_STAGES\_TOTAL - C\_NR\_STAGES\_LOW$)

4.3 IO ports

Port	Width	Direction	Description
PLB bus connections			
`SPLB_Clk`	1	in	see note 1
`SPLB_Rst`	1	in	see note 1
`PLB_ABus`	32	in	see note 1
`PLB_PAValid`	1	in	see note 1
`PLB_masterID`	3	in	see note 1
`PLB_RNW`	1	in	see note 1
`PLB_BE`	4	in	see note 1
`PLB_size`	4	in	see note 1
`PLB_type`	3	in	see note 1
`PLB_wrDBus`	32	in	see note 1
`Sl_addrAck`	1	out	see note 1
`Sl_SSize`	2	out	see note 1
`Sl_wait`	1	out	see note 1
`Sl_rearbitrate`	1	out	see note 1
`Sl_wrDack`	1	out	see note 1
`Sl_wrComp`	1	out	see note 1
`Sl_rdBus`	32	out	see note 1
`Sl_MBusy`	8	out	see note 1
`Sl_MWrErr`	8	out	see note 1
`Sl_MRdErr`	8	out	see note 1
unused PLB signals			
`PLB_UABus`	32	in	see note 1
`PLB_SAValid`	1	in	see note 1
`PLB_rdPrim`	1	in	see note 1
`PLB_wrPrim`	1	in	see note 1
`PLB_abort`	1	in	see note 1
`PLB_busLock`	1	in	see note 1
`PLB_MSize`	2	in	see note 1
`PLB_TAttribute`	16	in	see note 1
`PLB_lockerr`	1	in	see note 1
`PLB_wrBurst`	1	in	see note 1
`PLB_rdBurst`	1	in	see note 1
`PLB_wrPendReq`	1	in	see note 1
`PLB_rdPendReq`	1	in	see note 1
`PLB_rdPendPri`	2	in	see note 1
`PLB_wrPendPri`	2	in	see note 1
`PLB_reqPri`	2	in	see note 1
`Sl_wrBTerm`	1	out	see note 1
`Sl_rdWdAddr`	4	out	see note 1
`Sl_rdBTerm`	1	out	see note 1
`Sl_MIRQ`	8	out	see note 1
Core signals			
`IP2INTC_Irpt`	1	out	core interrupt signal
`calc_time`	1	out	is high when core is performing a multiplication, for monitoring

Note 1: The function and timing of this signal is defined in the IBM® 128-Bit Processor Local Bus Architecture Specification Version 4.6.

4.4 Registers

This section specifies the IP core internal registers as seen from the software. These registers allow to control and configure the modular exponentiation core and to read out its state. All addresses given in this table are relative to the IP core's base address.

Name	Width	Address	Access	Description
control register	32	0x0000	RW	multiplier core control signals and interrupt flags register
software reset	32	0x0100	W	soft reset for the IP core
Interrupt controller registers				
global interrupt enable register	32	0x021C	RW	global interrupt enable for the IP core
interrupt status register	32	0x0220	R	register for interrupt status flags
interrupt enable register	32	0x0228	RW	register to enable individual IP core interrupts

4.4.1 Control register (offset = 0x0000)

This registers holds the control inputs to the multiplier core and the interrupt flags.

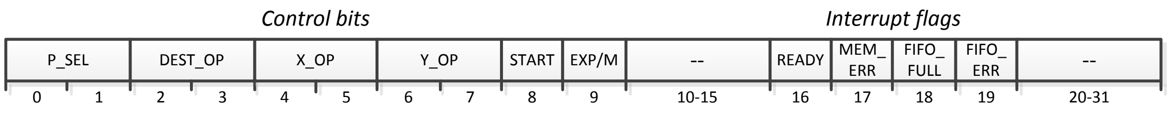

Figure 4.2: control register

bits 0-1	P_SEL : selects which pipeline part to be active • ”01” lower pipeline part • ”10” higher pipeline part • ”11” full pipeline • ”00” invalid selection
bits 2-3	DEST_OP : selects the operand (0-3) to store the result in for a single Montgomery multiplication[1]
bits 4-5	X_OP : selects the x operand (0-3) for a single Montgomery multiplication[1]
bits 6-7	Y_OP : selects the y operand (0-3) for a single Montgomery multiplication[1]
bit 8	START : starts the multiplication/exponentiation
bit 9	EXP/M : selects the operating mode • ”0” single Montgomery multiplications • ”1” simultaneous exponentiations
bits 10-15	unimplemented
bit 16	READY : ready flag, ”1” when multiplication is done must be cleared in software
bit 17	MEM_ERR : memory collision error flag, ”1” when write error occurred must be cleared in software
bit 18	FIFO_FULL : FIFO full error flag, ”1” when FIFO is full must be cleared in software
bit 19	FIFO_ERR : FIFO write/push error flag, ”1” when push error occurred must be cleared in software
bits 20-31	unimplemented

[1] when the core is running in exponentiation mode, the parameters DEST_OP, X_OP and Y_OP have no effect.

4.4.2 Software reset register (offset = 0x0100)

This is a register with write only access, and provides the possibility to reset the IP core from software by writing 0x0000000A to this address. The reset affects the full IP core, thus resetting the control register, interrupt controller, the multiplier pipeline, FIFO and control logic of the core.

4.4.3 Global interrupt enable register (offset = 0x021C)

This register contains a single defined bit in the high-order position. The GIE bit enables or disables all interrupts form the IP core.

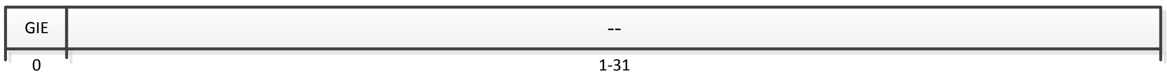

Figure 4.3: Global interrupt enable register

bit 0	GIE : Global interrupt enable • ”0” disables all core interrupts • ”1” enables all core interrupts
bits 1-31	unimplemented

4.4.4 Interrupt status register (offset = 0x0220)

Read-only register that contains the status of the core interrupts. Currently there is only one common interrupt from the core that is asserted when a multiplication/exponentiation is done, FIFO is full, on FIFO push error or memory write collision.

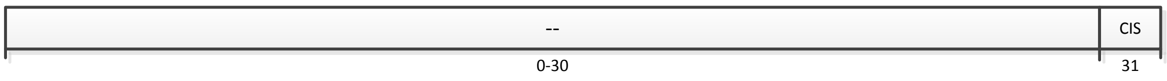

Figure 4.4: Interrupt status register

bits 0-30	unimplemented
bit 31	CIS : Core interrupt status is high when interrupt is requested from core

4.4.5 interrupt enable register (offset = 0x0228)

This register contains the interrupt enable bits for the respective interrupt bits of the interrupt status register.

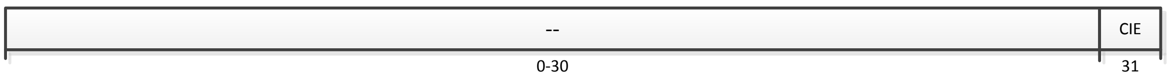

Figure 4.5: Interrupt enable register

bits 0-30	unimplemented
bit 31	CIE : Core interrupt enable • ”0” disable core interrupt • ”1” enable core interrupt

4.5 Interfacing the core's RAM

Special attention must be taken when writing data to the operands and modulus. The least significant bit of the data has be on the lowest address and the most significant bit on the highest address. A write to the RAM has to happen 1 word at a time, byte writes are not supported due to the structure of the RAM.

4.6 Handling interrupts

When the embedded processor receives an interrupt signal from this core, it is up to the controlling software to determine the source of the interrupt by reading out the interrupt flag of the control register.

Chapter 5

Performance and resource usage

This Modular Simultaneous Exponentiation IP core is designed to speed up modular simultaneous exponentiations on embedded systems. On embedded processors, software implementations (even with specialized libraries like GMP[1]), demand much CPU time when large operands are used. Practical tests of this core have shown a significant speed-up compared to software computations. For $n = 1536$ and $t = 1024$, hardware is about 70 times faster than a GMP-based implementation (with embedded linux) an a 100 MHz MicroBlaze processor (32-bit).

For the multiplier, execution time is given by (3.1), where τ_c is defined by the core operating frequency. Since the maximum frequency is highly influenced by the latency in the critical path, we can expect to achieve higher frequencies for shorter stage lengths. This trend is seen in Figure 5.1 for different operand lengths, which are results used from the static timing analysis during synthesis. A minimum execution time in this graph is found when the maximum operating frequency of the core first reaches the maximum frequency of the FGPA in use. Beyond that point, using a smaller stage width has no positive effect anymore because the frequency can not rise anymore and the number of clock cycles to complete a multiplication increases. Another remark that can be made is that splitting the pipeline, has no considerable effect on the performance of the core.

[1]GNU Multiple Precision Arithmetic Library – Project website: `http://gmplib.org/`

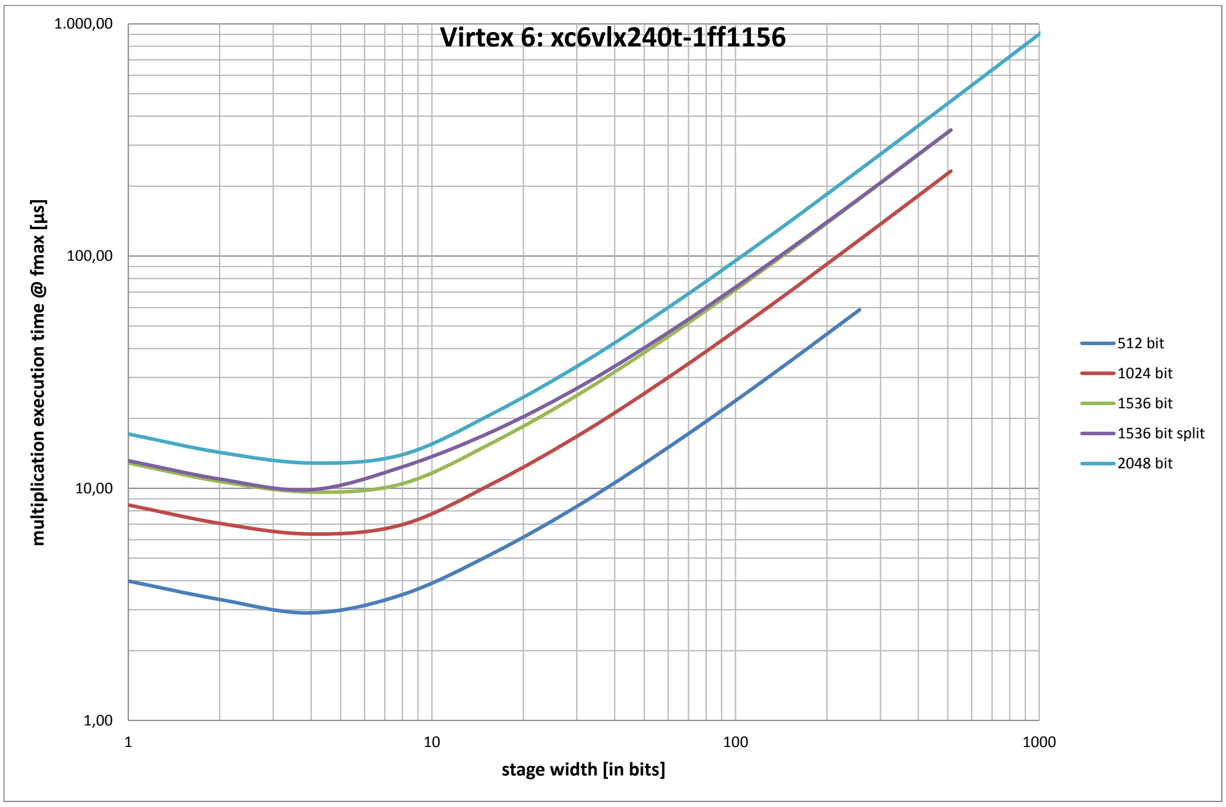

Figure 5.1: Example of multiplication execution time in function of the stage width for a Virtex6 FPGA.

In general, shorter stage lengths result in smaller execution times. However, using more stages implies that more flip-flops will be needed, thus more resources are used. A balance must be found between a execution time and resources. Currently, the core's operating frequency is the same as the bus frequency of the embedded processor. For optimal operation of the core, the stage width must be chosen so that the maximum frequency given in synthesis is just above or equal to the bus frequency.
In the tables below resource usage and timing results are shown for different operand lengths and FPGA's. As a rule of thumb, the number of flip-flops is given by (5.1).

$$5+2\cdot n+6\cdot\frac{n}{s}+\lceil\log_2(n)\rceil+\lceil\log_2(\frac{n}{s})\rceil \tag{5.1}$$

where s is the stage width.
The number of LUTs is almost completely determined by n and the number of LUT-inputs. A pre-synthesis estimate can be made with (5.2) and (5.3).

$$8\cdot n \quad \text{for 4-input LUTs} \tag{5.2}$$

$$6\cdot n \quad \text{for 6-input LUTs} \tag{5.3}$$

Results for a Virtex 6 device xc6vlx240t-1ff1156, speedgrade -1
Synthesis settings: Optimization: area, Effort: high

n		512			1024			2048		[bit]
$stagewidth$	64	16	4	64	16	4	64	16	4	[bit]
f_{max}	64,91	199,96	395,57	64,91	199,66	358,62	94,91	199,96	358,62	[MHz]
$T_m@f_{max}$	15,87	5,27	2,91	31,77	10,55	9,63	63,57	21,11	12,84	[μs]
$cycles$	1030	1054	1150	2062	2110	3454	4126	4222	4606	[cycles]
Resources										
Flipflops	1089	1235	1813	2163	2453	5401	4309	4887	7193	
LUT's	3094	3096	3102	6169	6171	9252	12315	12318	12324	

Results for a Spartan 3 device xc3s1000-5fg320, speedgrade -5
Synthesis settings: Optimization: area, Effort: high

n		256				512		[bit]
$stagewidth$	32	8	2	64	32	8	2	[bit]
$f_m ax$	21,49	69,30	127,32	11,36	21,49	69,30	127,32	[MHz]
$T_m@f_{max}$	24,1	7,82	5,01	90,7	48,29	15,67	10,04	[μs]
$cycles$	518	542	638	1030	1038	1086	1278	[cycles]
Resources								
Flipflops	576	722	1300	1089	1138	1428	2582	
LUT's	2072	2074	2079	4124	4126	4128	4135	

Results for a Virtex 4 device xc4vlx200-11ff1513, speedgrade -11
Synthesis settings: Optimization: area, Effort: high

n		512				1024			[bit]
$stagewidth$	64	32	8	2	128	32	8	2	[bit]
f_{max}	22,83	43,05	138,31	246,98	11,77	43,05	138,31	246,98	[MHz]
$T_m@f_{max}$	45,12	24,11	7,85	5,17	87,5	24,5	8,31	6,21	[μs]
$cycles$	1030	1038	1086	1278	1030	1054	1150	1534	[cycles]
Resources									
Flipflops	1089	1138	1428	2582	2114	2260	2838	5144	
LUT's	4124	4126	4128	4135	8225	8230	8234	8238	

Bibliography

[1] P. L. Montgomery, “Modular multiplication without trail division,” *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.

[2] N. N. de Macedo Mourelle L., “Three hardware architectures for the binary modular exponentiation: Sequential, parallel, and systolic,” *IEEE Transactions on Circuits and Systems - I: Regular Papers*, vol. 53, no. 3, pp. 627–633, 2006.

[3] Xilinx, “Plbv46 slave single (v1.01a) ds561.” `http://www.xilinx.com/support/documentation/ip_documentation/plbv46_slave_single.pdf`.

[4] Xilinx, “Interrupt control (v2.01a) ds516.” `http://www.xilinx.com/support/documentation/ip_documentation/interrupt_control.pdf`.

License