Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Optimization of Montgomery Multiplication via MATLAB HDL Coder for FPGA Implementation

Ghada Elsayed

Ghadakdr@gmail.com

BUE University

Research Article

Keywords: FPGA Design, MATLAB HDL Coder, Security Computation, Embedded Systems, Complex Computation, combinatorial delay, pipeline, Montgomery Multiplication Algorithm

Posted Date: November 6th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11278580/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Optimization of Montgomery Multiplication via MATLAB HDL Coder for FPGA Implementation

Ghada Elsayed[1*]

[1*]Computer Networks and Cyber Security, BUE University, Suez Desert Road, El Shorouk, 11837, Cairo, Egypt.

Corresponding author(s). E-mail(s): Ghadakdr@gmail.com; Ghada.elsayed@bue.edu.eg;

Abstract

Field-Programmable Gate Arrays (FPGAs) are essential for implementing and verifying complex computational algorithms. This paper is part of a series of research aimed at enabling developers to use FPGAs as an embedded system without prior knowledge of digital design engineering. This paper discusses minimizing combinatorial delay that exists in complex algorithm computation using a software pipeline. How to implement this pipeline using a MATLAB HDL Coder algorithm. A comparison of the implementation of the complex Montgomery Multiplication Algorithm, with and without a pipeline, is performed on the same Virtex-7 platform. The Experiment shows a 56.5% improvement in the maximum frequency, 128.810 MHz. Although the utilization of slice registers increased by 1%, the number of slice-LUTs increased by 13%, the number of fully used LUT-FF pairs increased by 7%, and memory usage remained constant, demonstrating that high-level optimizations can produce hardware performance that typically requires expert-level manual RTL design.

Keywords: FPGA Design, MATLAB HDL Coder, Security Computation, Embedded Systems, Complex Computation, combinatorial delay, pipeline, Montgomery Multiplication Algorithm.

1 Introduction

This paper focuses on improving the developer's writing techniques to produce an efficient algorithm that supports the creation of good VHDL code. HDL Coder is an application inside MATLAB. The MATLAB HDL Coder is a model-based design

tool that automates the generation of VHDL or Verilog from MATLAB algorithms. The standard design flow involves a Unit Under Test (UUT) file and a corresponding testbench. The UUT should be written to be synthesized so that the visualization/debugging functions are not permitted. In the testbench, the usage of these functions is allowed. The UUT is the top-level function that calls all internal functions; there is no need to add the files for these internal functions to the project structure. This workflow progresses through the selection of input types, fixed-point conversion, the selection of the target FPGA, and finally the operation of a third-party FPGA synthesizer [1]. In Figure 1 [2], the two ways to generate HDL code using MATLAB are introduced.

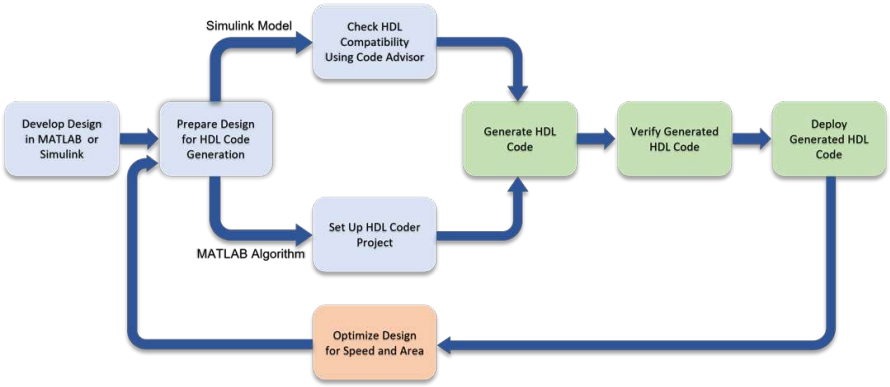


Fig. 1 HDL code generation workflow.

This paper selects the workflow of the MATLAB algorithm rather than the Simulink model. Many developers used it for various applications, but were not interested in optimization or comparison with hardware-description languages. Some have applied it to various applications, such as convolution filtering for image processing on the Xilinx Virtex-4 platform, where researchers used the Xilinx 13.1 toolkit to manage specialized hardware resources [3].

Other studies focused on specific applications in complex communication systems, such as the implementation of an OFDM (Orthogonal Frequency Division Multiplexing)-based DVB-T (a digital multi-carrier modulation method) system. The main challenge was to convert floating-point calculations of the MATLAB code to fixed-point VHDL logic to improve the ratio of DSP chips and the multipliers used [4].

Similarly, digital FIR filters were applied to evaluate the tool's performance. The results revealed that while HDL Coder offers high flexibility, it may consume approximately 15% more lookup tables (LUTs) compared to pre-implemented and optimized Soft-IP cores from vendors [5].

In more general applications, researchers have demonstrated the "source-code utility" of generated code by successfully converting MATLAB functions to VHDL that can be fabricated for implementation in standard FPGA assemblies [6].

Others focused on the tools and methodologies for compiling the core code. Early research introduced systems such as Accel FPGA and various MATLAB-to-VHDL

compilers designed to map high-level programs—ranging from 20 to 170 lines of code—to distributed hardware architectures such as the Wild Child board [7], [8], and [9].

These publications emphasized the importance of architecture files (ARCH) to ensure that the generated RTL is compatible with various FPGA vendors and supports optimal mapping for multiple hardware modules [10]. In Table 1, a summary of different FPGA implementations using MATLAB HDL Coder. The table also illustrates the target FPGA, the maximum clock frequency, and the percentage of utilization of area resources. This discussion intersects with our topic only in the method of designing the FPGA. The latter implementations do not have a relation to the target algorithm. This work demonstrates the successful design and implementation of an FPGA system using MATLAB HDL Coder. A key focus is identifying the techniques and design practices that enable the generated hardware to achieve optimization levels comparable to those of hardware description language (HDL) implementations developed by experienced digital design engineers. The comparison is done on the basis of the third-party FPGA synthesizer report. The synthesis process produces comprehensive reports on the utilization of the area and the maximum clock frequency estimate (f_{max}) [14]. FPGA synthesizers evaluate the target design by producing estimated reporting metrics. The synthesis report aggregates physical hardware usage, tracking essential components such as LUTs, flip-flops, and specialized DSP blocks [15]. Beyond hardware footprints, evaluating the achievable maximum frequency (f_{max}) remains a key constraint for timing closure. Researchers frequently utilize these output summaries to benchmark novel architectural designs against baseline hardware implementations.

When algorithms are implemented within a MATLAB workflow, optimization requires careful attention to several key factors—most notably the data types assigned to inputs, outputs, and internal signals. Developers with general MATLAB experience often overlook the critical role of data typing when their end goal is HDL code generation, particularly regarding how these variables translate into actual hardware implementation. How many bits would a certain variable require if it were declared as double? From many trials, the recommended types are either logic or fixed-point numeric, and of course any array of them. For the logic type, it represents a bit or std_logic in VHDL. So, its value in MATLAB is true or false and in VHDL is 1 or 0. The word length of the logic type is one. The fixed-point numeric, on the other hand, is the most general and wide type that represents the variable properties: Value (v), whether it is scalar, vector, matrix, or multidimensional array; Sign (s), 1 is signed and 0 is unsigned; The word length in bits (w), the default is 16 bits; the fraction length in bits (f), the default is 15 bits. The typical command is *a = fi (v, s, w, f)*, respectively. Ensure that if the code equates two variables, they have exactly the same type; otherwise, an error will appear in the code generation. The second point is the optimum technique for writing loops. Loops are not preferable if the objective is to find an optimum design for maximum clock frequency [16]. Optimizing the design for maximum clock frequency requires replacing traditional for loops with conditional structures. This technique was previously discussed [17] and allows the target hardware to process repetitions in successive clock cycles. The end of the loop is controlled

Table 1 Different FPGA Designs and Implementations for MATLAB/HDL Coder

Application	**FPGA Type**	**Area Resource Utilization**	**Citation**
General MATLAB-to-VHDL Conversion	Generic FPGA Kit	Analysis of source utility and VHDL code generation flow for bitstream creation.	[6]
General FPGA Design Environment	Mentor Graphics Precision-C	Utilization of Precision-C and HDL-Link for implementation in research environments.	[11]
Design Flow Integration	Generic FPGA	Efficient flow integrating MATLAB with HDL Designer for rapid prototyping.	[12]
OFDM-based DVB-T System	Generic FPGA	Optimization of fixed-point design to ensure efficient VHDL generation for wireless standards.	[4]
AccelFPGA Compiler Overview	Multi-vendor (Xilinx/Altera)	Automated RTL generation for device-specific synthesis across major FPGA vendors.	[7]
MATLAB Mapping Benchmarks	Multi-vendor FPGA	Mapping of benchmarks ranging from 20 to 170 lines of code into synthesizable VHDL.	[8]
Convolution Filtering	Xilinx Virtex-4	Implementation using Xilinx 13.1; results accessed through MATLAB interface.	[3]
Hardware Synthesis (Wildchild)	Wildchild Board	Architecture-independent synthesis using specific hardware dealgorithmion (ARCH) files.	[9]
Digital FIR Filter Comparison	Generic FPGA	Comparative study between HDL Coder-generated logic and vendor-provided Soft-IP Cores.	[5]
Optimized RTL Synthesis	Multi-FPGA systems	Systematic mapping of MATLAB applications to distributed multi-FPGA hardware architectures.	[10]
Adaptive Digital Chaotic Generator	Xilinx Spartan-6 (xc6slx9)	Design 1: 15.711 (MHz) Design 2: 11.635(MHz)	[13]

by implementing a counter that counts the required number of clock cycles to represent the loop condition. The counter needs to store the previous value in a register to be incremented next clock cycle. This is achieved by defining the variable using the persistent statement. If the for loop is kept as is, the synthesizer will understand that all rounds should be implemented in a clock cycle. This represents a bulk combinatorial delay where the input is feedback from the output without a pipeline register. By replacing this loop with an if condition, only one round is implemented, not all of them; this not only reduces the delay of the large combinatorial logic but also reduces area. Hardware naturally reuses the same logic at each edge of the clock cycle unless a condition prevents it. Using this technique, the MATLAB HDL coder can compete with the VHDL designer[17]. In another previous scope, the Montgomery Multiplication algorithm was studied and designed using the MATLAB HDL coder. This marked the initial implementation of the algorithm using this tool. The design was scalable such that it can fit any small FPGA on account of latency. The design was also scalable such that one can configure the IOs data width, the modulus length, and whether there is serialization with which factor? [18]. The maximum achieved frequency was 80.81 MHz for a 4096-bit key size with an 8-bit data width (base 256) and a serialization factor of 2 on a Xilinx Virtex-7 (xc7vx330t-2ffg1157). Although these results show lower performance compared to the prior literature, direct point-to-point comparison is constrained by differences in FPGA target technologies, which inherently alter the maximum attainable clock frequency ($f_{\max}$). To avoid duplicate of previous research on Montgomery multiplication designs based on FPGA, a comparative summary of relevant previous work is presented in Table 2. Table 2 summarizes this existing work,

Table 2 Summary of Montgomery Multiplication Implementations on FPGAs

FPGA Family/Device	**Key Lengths(k) (Bits)**	**Cryptosystem Context**	**Max f_{max} (MHz)**	**Citations**
Xilinx Virtex-7 Series	256, 384, 521, 1024, 2048, 4096	RSA, ECC, PQC	∼300 to 475+	[19], [18], [20], [21], [28]
Xilinx Virtex-6 Series	192, 224, 256, 384, 521, 1024	ECC, RSA	∼250 to 350	[29], [20], [21], [22], [23]
Xilinx Artix-7 / Zynq-7000 Series	112, 163, 256	ECC (IoT/Embedded)	∼150 to 295	[24], [25], [30], [31]
Xilinx Spartan-6	112, 163, 256	ECC, RSA (Smaller)	∼150 to 280	[25], [26], [27]
Xilinx Virtex-5 Series	163, 256, 571	ECC, RSA	∼200 to 320	[19], [28], [32]

highlighting the critical variations in the maximum clock frequency, device utilization, and operand bit-widths, thereby providing a clear performance benchmark for

the current work. Consequently, this research shifts focus toward a modular scripting paradigm. By structurally decomposing the algorithm into individually verifiable functional blocks, this study evaluates how micro-architectural isolation influences the overall timing closure and resource consumption of the synthesized hardware. After the part that reduces the clock frequency, a proposal will be introduced to enhance the results. The technique of this enhancement will be discussed in detail. A Necessary background on the Montgomery multiplication Algorithm is presented in the following paragraph.
Modular multiplication is a complex computational process that requires both hardware resources and processing time. The Montgomery multiplication algorithm is widely used for this type of calculation [33]. Figure 2 illustrates the working mechanism of the algorithm in a flowing manner, where n is the modulus, x and y are the multiplicand and the multiplcand. This flowchart for the hardware-realization version is presented in [18].

There are some basic changes between this hardware-realized algorithm and that published by Montgomery [33]. The inner step of the for loop is converted to a nested for loop to introduce the scalability feature. This also introduced control over how many resources can be used and allowed implementation of the design over a small FPGA with a latency trade-off. The flowchart shows that the inputs should be loaded, indicating the presence of a memory module here. Next, we can find the multiply-add step and slice the least significant word to achieve mod b, where b is the base number. In the inner for loop, there are also multiply, add, and shift. In the last step, there is a conditional subtraction. This indicates that we should have here a computational module that has multiplying, adding, shifting, slicing, and subtracting operations. The responsibility of synchronization and timing indicates the existence of the control module. For more details on the algorithm, see [33] and [18]. The flow chart also illustrates that two main counters control the inner and outer loops. It also illustrates that there are two feedback paths in which the calculated output is used as input in the next round. They are the carry *cw* and the temporary output *aj*. To store these values to be used next clock cycle, use the persistent statement [17]. The subtraction part is also done in the same way by feeding back the borrow. The following section will illustrate the architecture of the system designed as three modules. It will illustrate the design of each module and the connection between them.

2 Preliminary Design for the Montgomery Multiplication Algorithm

This section introduces the hardware architecture of the modular Montgomery multiplication algorithm. First, the primary I/O signals and their corresponding bit-widths are defined, followed by a structural diagram of the interconnections between components. A functional breakdown of each sub-module is then presented. The architecture is parameterized by three main configuration factors: Data Width (Data_Width): The bit-length of the modulus word, which establishes the radix of the system $b = 2^{\text{Data_Width}}$. Word Count ($W$): The ratio of the total key length to the data width ($W = \frac{\text{KeyLength}}{\text{Data_Width}}$).Serialization Factor ($S$): The number of Data_Width units processed

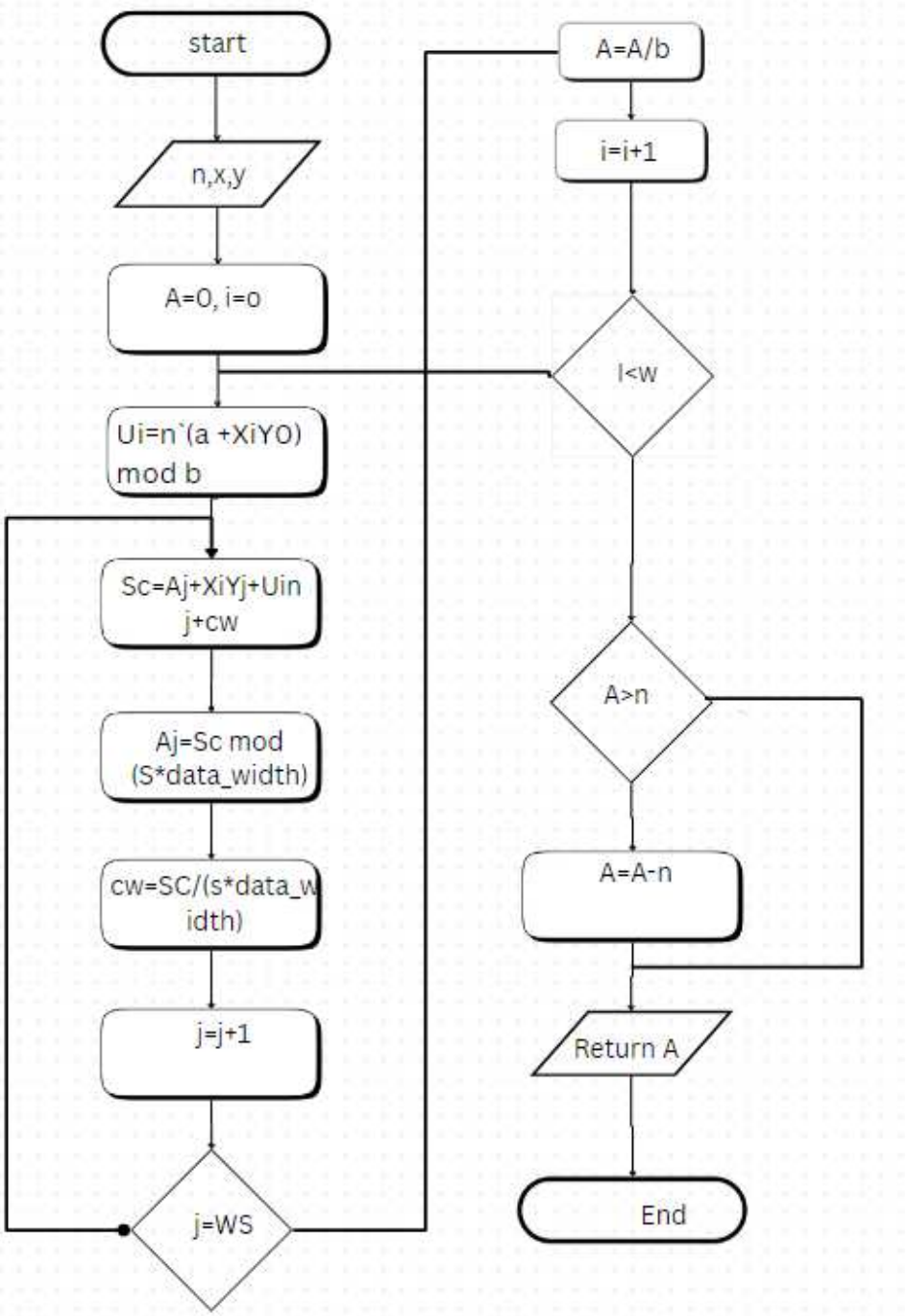


Fig. 2 A Flow Chart of the Montgomery Multiplication Algorithm.

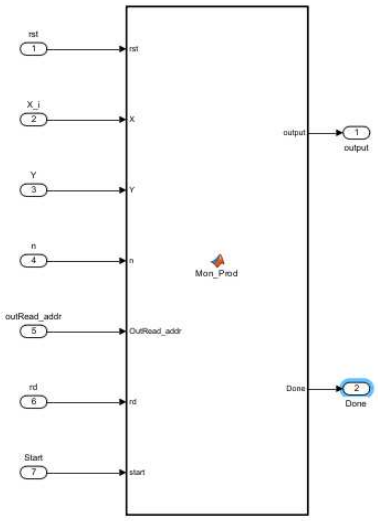


Fig. 3 Montgomery Multiplication Top-Level.

per clock cycle. From these primary parameters, several operational metrics and signals are derived: a Digit represents a Data_Width-bit word; a big_digit represents a word of $S \times$ Data_Width bits; WS denotes the total number of *big_digit* in the modulus ($WS = W/S$); ed specifies the bit-width required to address W words ($ed = \lceil \log_2 W \rceil$); and eds specifies the bit-width required to address S ($eds = \lceil \log_2 S \rceil$).

2.1 Top-Level Block Diagram

Figure 3 illustrates the top-level block diagram of the proposed design, which features a total of seven inputs and two outputs. The data path includes multi-bit ports of width *data_width* for the X, the multiplier Y, the modulus (n), and the primary output data. Control and status operations are governed by three single-bit inputs—*rst*, *start*, and *rd*—and one single-bit active-high output flag, *Done*. Specifically, *rst* initializes all internal registers and memory blocks; *start* triggers algorithm execution (reserved for integration into modular exponentiation sequences); and *rd* enables reading the computed result from memory once the active-high *Done* signal indicates completion. The memory read operation is indexed by the final address input, *outRead_addr*. The subsequent subsections first present the complete top-level structure and interconnection, followed by a detailed functional breakdown of each sub-module.

2.2 Montgomery Multiplication Modules Block Diagram

One of the aims of this paper is to write a script for each component separately to study the effect of modular design on performance. Figure 4 illustrates the three main modules. The *Mon_prod_mul* is the module that is responsible for the computations. *Mon_prod_mem* is the module that stores data. The *Mod_prod_cont* is the module that

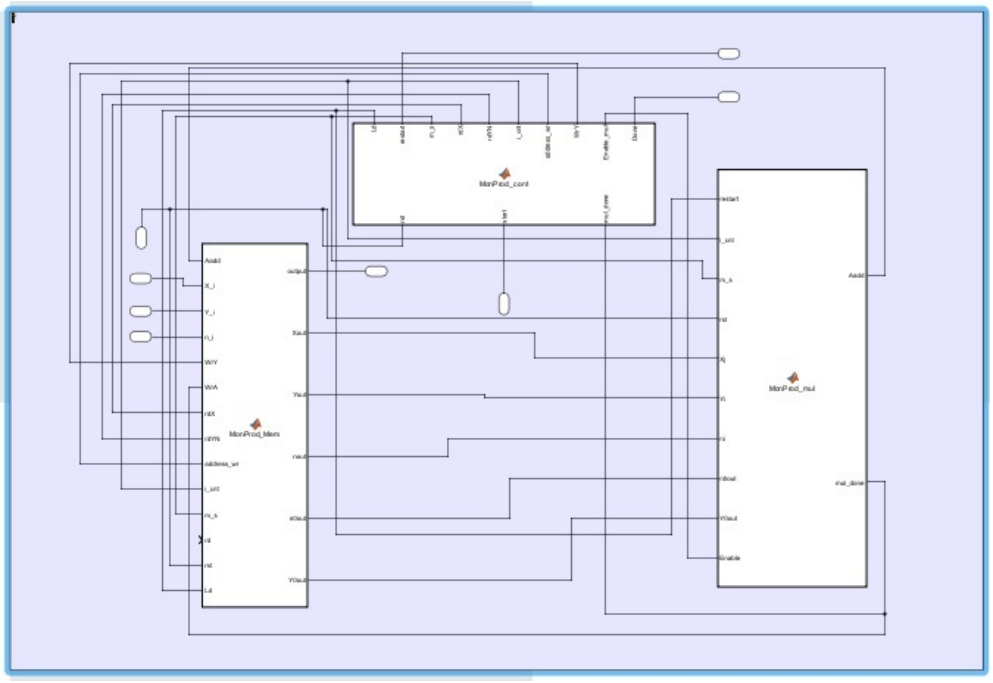


Fig. 4 Montgomery Multiplication Block Diagram.

supplies the other two modules with the required control signals. The figure also illustrates that the *rst* signal is an input to all three modules and that the data input ports are loaded into the *Mon_prod_mem*. During loading, the *Mod_prod_cont* supplies the *Mon_prod_mem* with the *Ld* and the *address_wr*, the load signal, and the required address for writing. The *addressth is W. After the loa_wr* widding state, the computation is started with multiplication and addition. This is done *Digit by big_digit* each clock cycle until the last *big_digit*. This loop is controlled by the counter *m_s*. This counter counts from zero to *WS-1*. So, for each digit in X, an inner loop is done for all *big_digits* on Y and n. The counter that controls reading X is called *i_cnt*. This counter counts from zero to *W-1*. At the end of this counter, the subtraction loop will start. The subtraction is done by subtracting the output of the previous stage from the modulus n. If there is no borrowing *brw*, then the output is taken from the subtraction; otherwise, it is taken from the result of the multiply and add process. From the previous discussion, we can conclude that the memory module outputs *big_digits* for Y and n. It outputs a *Digit* for X. These are supplied to the computation module. The subtraction is performed using *big_digits*. The final results are transferred from the computation module to the memory in parallel. This is to reduce an additional loop to write the output digit by digit. Figure 5 illustrates the top-level system architecture and I/O interconnects. Execution begins with the control module *MonProd_cont*, which generates and asserts the control signals required by the remaining hardware modules. Subsequently, the memory module *MonProd_Mem* is instantiated

```
function [output,Done] = Mon_Prod(rst,X,Y,n,OutRead_addr,start)
% This function calculates the Montgomery Product
% INPUT: integers n = (nm-1 … n1n0)b, x = (xm-1 ... x1x0)b, y = (ym-1 …y1y0)b with 0 ≤
data_width=8;% this controlled by the input output controls
W=8;
serialization=4;
WS=W/serialization;
persistent Aadd_i
if isempty (Aadd_i)
    Aadd_i = fi(zeros(serialization,WS),0,data_width,0);
end
Aadd=Aadd_i;
persistent mul_done
if isempty (mul_done)
mul_done=false;
end

[Ld,restart,m_s,rdX,rdYN,i_cnt,address_wr,Done,WrY,Enable_mul] = MonProd_cont(rst, ...
    start, mul_done);
[Xout, Yout, nout, n0out, Y0out,output] = MonProd_Mem(rst,Ld,WrY,mul_done,rdX,rdYN, ...
    address_wr, i_cnt,m_s,OutRead_addr,Aadd,X,Y,n);
[Aadd_i,mul_done] = MonProd_mul(restart, i_cnt, m_s, rst,Enable_mul,Xout,Yout,nout, ...
    n0out, Y0out);
```

Fig. 5 Montgomery Multiplication Top-Level Script

to load the input data—specifically the parameters n, X, and Y. Once memory initialization is complete, this module provides the computational core *MonProd_mul* with the required multiplicand, multiplier, modulus, and the modular inverse of the modulus's least significant digit. Finally, the computation module is invoked to execute the arithmetic operation.

In this HDL-targeted modeling approach, high-level software code serves as a structural description of hardware rather than a sequential CPU instruction stream. Although variables appear to update sequentially within the script, they represent physical wires and registers evaluated concurrently. The total output of the block is resolved in each clock cycle, during which the generated FPGA hardware executes the complete control flow simultaneously—leveraging the script's conditional logic to route the data through specific hardware paths at each active clock edge. But still, the sequence of calling the internal function matters, especially in dependency cases.

2.3 Modules of the Montgomery Multiplication Design

This section presents the detailed design of each sub-module. A primary constraint of MATLAB HDL Coder is that variable dimensions and array bounds must be defined as compile-time constants for valid HDL synthesis. To achieve a fully parameterized and configurable design despite this limitation, a top-level wrapper architecture is implemented. In this approach, configuration factors are declared as static constants in a parent wrapper file, which then passes them as input to the underlying module functions. This parameterization pattern is applied to the controller module, the memory

```
function [Ld,restart,m_s,rdX,rdYN,i_cnt,address_wr,Done,WrY,A_cnt,Enable_mul] = MonProd_cont( ...
    rst, start,mul_done)

W=8;% number of digits in the modulus n:%W=key_length/data_width
ad=3;% ad =ceil(log2(W))
serialization=4;
ads=1;% ads= ceil(log2(serialization))
WS=W/serialization;% integer ceil(W/serialization)
if serialization==1
    ad_ads=ad;
else
    ad_ads=ad-ads;
end


[Ld,restart,m_s,rdX,rdYN,i_cnt,address_wr,Done,WrY,A_cnt,Enable_mul] = control(rst, ...
    start,mul_done,ad,ad_ads,ads,W,WS);

end
```

Fig. 6 Controller Module Configuration

```
function[Xout, Yout, nout, n0out, Y0out,output] = MonProd_Mem(rst, ...
    Ld,WrY,WrA,rdX,rdYN,address_wr, i_cnt,m_s,rd,Aadd,X_i,Y_i,n_i)

data_width=8;% this controlled by the input output controls
serialization=4;
W=8;
ad=3;%2^ad=W
WS=W/serialization;

X= fi(X_i,0,data_width,0);
n= fi(n_i,0,data_width,0);
Y= fi(Y_i,0,data_width,0);
rd=fi(rd,0,ad,0);
[Xout, Yout, nout, n0out, Y0out,output] = memories(ad, WS,serialization, ...
    data_width,rst,Ld,WrY,WrA,rdX,rdYN,address_wr, i_cnt,m_s,rd,Aadd,X,Y,n);
end
```

Fig. 7 Memory Module Configuration

module, and the computation module, as illustrated in Figures 6, 7, and 8, respectively. These figures illustrate that the underlying module functions for the parent modules are *control*, *memories*, and *MonProd_int*, respectively. The whole script of the Montgomery multiplication algorithm is named *Mon_Prod*. Calls the three modules inside the required sequence as illustrated in Figure 9. As discussed above, configuration parameters are defined in the parent files, and one should be very careful when changing them to be the same in all files. All modules should be the same. In the following subsections, the block diagram of each module is discussed.

```
function [Aadd,add_mul] = MonProd_mul(restart, i_cnt, m_s, rst, ...
    Enable,Xj,Yi,ni, n0out, Y0out)
data_width=8;% this controlled by the input output controls
serialization=4;
W=8;
WS=W/serialization;
Aadd = fi(zeros(serialization,WS),0,data_width,0);

persistent add_mul_i
if isempty (add_mul_i)
add_mul_i=false;
end
persistent Aadd_i;
 if isempty (Aadd_i)
 Aadd_i = fi(zeros(serialization,WS),0,data_width,0);
 end
if Enable ==false
[Aadd_i,add_mul_i] = MonProd_int(restart,i_cnt, m_s, rst,Xj,Yi,ni, n0out, ...
    Y0out,serialization,data_width,WS,W);
end
add_mul=add_mul_i;
Aadd=Aadd_i;
end
```

Fig. 8 Computation Module Configuration

2.3.1 Controller Module

Figure 10 shows the top-level architecture of the controller module. Based on the preceding specification, the dimensions of the I/O signals are defined as follows: single-bit logical control signals comprise *rst, start, mul_done, Ld, restart, Enable_mul, Done, rd, rdYN, WrY*, and *rdX*. Multi-bit control signals include counters *address_wr* and the *i_cnt* , sized to *ed* bits, and counter m_s, with a *bit_width* of $\lceil \log_2(WS) \rceil$. The signal *Enable_mul* is a one-clock-cycle delayed version of *Ld*. The *Ld* signal remains active (high) until *address_wr* triggers, after which its state becomes fixed. Unlike the cyclical counter m_s, *address_wr* counts non-repetitively from 0 to W; once it reaches W, it remains W and stops. In contrast, m_s is initialized to 0 and continuously counts from 0 to W_S. When *LD*, *Start*, or *Reset* is asserted, the m_s counter remains held at zero. The m_s counter generates four key control signals: *Enab_m_s*, *Enab_i_cnt*, *rdX*, and *rdYN*. The *Enab_m_s* signal is asserted one clock cycle after m_s triggers to control *i_cnt*, while *Enab_i_cnt* enables *i_cnt* increment. The signals *rdX* and *rdY_N* serve as read-enable signals for memory operations; since reading operations for memories Y and N occur simultaneously, they share the single *rdYN* read signal. The *i_cnt* counter counts non-repetitively from 0 to W, incrementing by one each time m_s fires. Upon reaching completion (firing), *i_cnt* activates the done signal.

```
function [output,Done] = Mon_Prod(rst,X,Y,n,OutRead_addr,rd,start)
W=512;
ad=9;% ad =ceil(log2(W))
serialization=2;
ads=1;% ads= ceil(log2(serialization))
WS=W/serialization;% integer ceil(W/serialization)
if serialization==1
  ad_ads=ad;
else
ad_ads=ad-ads;
end
persistent Aadd_i
if isempty (Aadd_i)
    Aadd_i = fi(zeros(serialization,WS),0,data_width,0);
end
Aadd=Aadd_i;
persistent mul_done
if isempty (mul_done)
mul_done=false;
end

[Ld,restart,j_cnt,rdX,rdYN,i_cnt,address_wr,Done,WrY,~,Enable_mul] = MonProd_cont( ...
    rst, start, mul_done);
 m_s=bitsliceget(j_cnt,ad_ads+1,2);

[Xout, Yout, nout, n0out, Y0out,output] = MonProd_Mem(rst,Ld,WrY,mul_done, ...
    rdX,rdYN,address_wr, i_cnt,m_s,OutRead_addr,Aadd,X,Y,n);
[Aadd_i,mul_done] = MonProd_mul(restart, i_cnt, j_cnt, rst,Enable_mul,Xout, ...
    Yout,nout, n0out, Y0out);
end
```

Fig. 9 Mon_Prod Script

2.3.2 Memory Module

Figure 11 illustrates the top-level view of the memory module. The number of bits of the memory module's IOs is as follows: first, the logical IOs, which are *rst*, *Ld*, *rdYN*, *rdX*, and *WrY*. Second, the number of bits of the counters *m_s*, which is $log_2(WS)$, the number of bits of the counter *i_cnt*, *address_wr*, and *rd* are *ad*. Third, the number of bits of the input data *Xi*, *Yi*, and *ni* is *Digit*, and *Aadd* is *key_length*. Fourth, the number of bits of the output port *Xout*, *Yout*, and *nout* is as *big_digit*, *Y0out*, *n0out*, and *output* is as *Digit*.

2.3.3 Computation Module

Figure 12 illustrates the top-level view of the computation module. The IOs of the computation module are as follows: first, the logical IOs, which are *rst*, *restart*, *Enable*, and *mul_done*, where Enable is connected to the control module *Enable_mul* port and *Enable_mul* is connected to the *mul_done* port of the control module. Second, the counters *m_s* and *i_cnt* are connected from the control module ports. Third, the number of bits of the data inputs *Xj*, *Yi*, and *ni* are connected from the memory outputs' ports, and the *output* number of bits is *data_width*. The Block diagram of the multiply-and-add part is illustrated in Figure 13. The figure illustrates three main

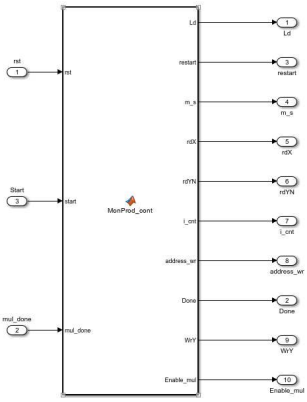


Fig. 10 Controller Module Top-Level

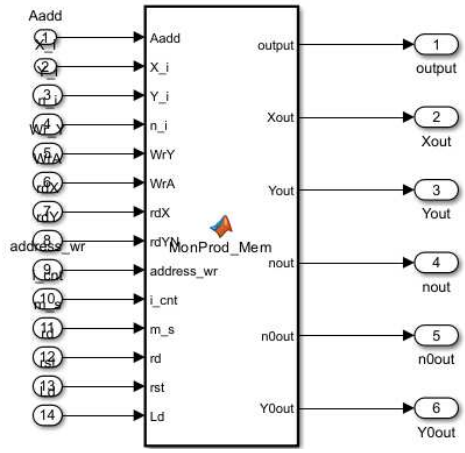


Fig. 11 Memory Module Top-Level

blocks *Ui_fun* that calculate the first line in the flowchart. It depends on the initial digit of the multiple Y, the multiplicand Xi, the negative of the multiplicative inverse of the first digit of the modulus, and the first digit of the feedback from the calculation output.

$$Ui = n' \times ((A0) + (Xi \times Y0))modb \tag{1}$$

This equation is written in two lines, one for the calculation, taking into consideration that the width of the Ui before the mod b operation is 3 × data_width +1, and after it is only data_width; the mod b operation is only one data_width. The mod b operation is performed using the *bitseliceget* embedded MATLAB function for the *fi* data type. This function slices the required bits from the input. The second line is for the *bitseliceget*. The second block is the *Add_mult*, which is responsible for calculating the next step of the flowchart 2.

$$SC = Aj + (Xi \times Yj) + (Ui \times nj) \tag{2}$$

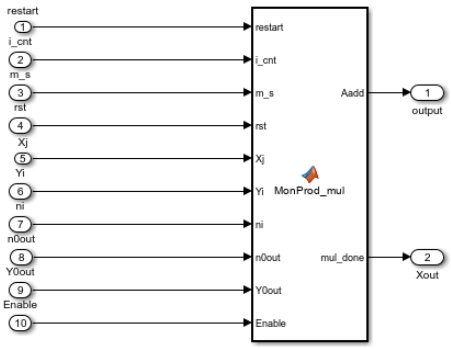


Fig. 12 Computation Module Top-Level

This equation illustrates that all the calculations are done in one line of code, taking into account that the size of the output will be 3 × data_width +2. The following two lines are for slicing the carry word from the output using *bitsliceget*. Both the carry word and the output are persistent variables. This is to be stored in a register for the next use in the next round/clock cycle. The block diagram is illustrated in Figure 15. The A-Register is initialized to zeros. Its size is *big_digits* by *WS*. The calculation of SC needs reading a word from the A register, and the output is sliced to the lowest *big_digits* bits to be stored in the A register in the same location. The reset bits are the carry CW. CW is a persistent variable to store the carry for the next round. The last block is for dividing the output by the radix b. This is done by shifting the words so that the first calculated output word is not needed. The control signal En specifies the shifting time. All used registers are for storing purposes. The subtraction part block diagram is illustrated in Figure 14. After writing the algorithm in a modular way, it should be tested using a testbench and continue in the workflow until the synthesis report is produced. This will be discussed in the next section.

3 Testers Design

In this section, the method of testing the hypotenuse is depicted. Four HDL Coder projects are used. The first is to test the entire Montgomery algorithm, where the UUT here is the Mon_Prod script as illustrated in Figure 9 and the test bench is illustrated in Figure 16. This figure illustrates the script used for testing the Mon_Prod script. In this script, the Mon_Prod function is called for different states. The first state is to reset everything to the initial values. This is done by enabling the active-high asynchronous *rst* signal. Then, the loading starts immediately after the *rst* goes low. The loading takes W clock cycles. The computation then starts for $2 \times WS + (WS \times W)$ clock cycles. After the computation is completed, the results are stored in memory and accessed via the read address, which is *outRead_addr*. The last three projects have the same testbench file, which is illustrated in Figure 17, but different UUTs. The second project tests the controller, so the UUT is the control script; the third project UUT is the memory script; and the fourth project UUT is the computation script.

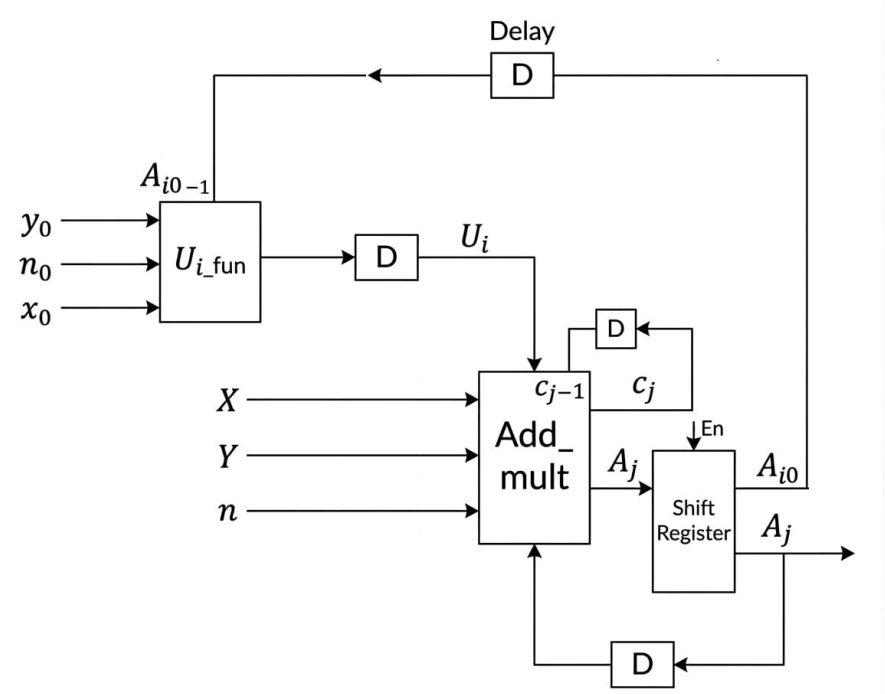


Fig. 13 Multiply and add Block diagram

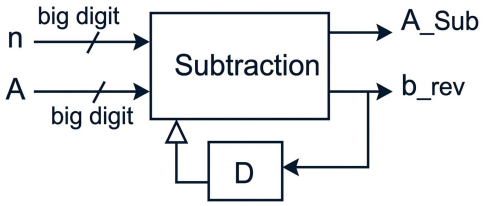


Fig. 14 Subtraction Block diagram

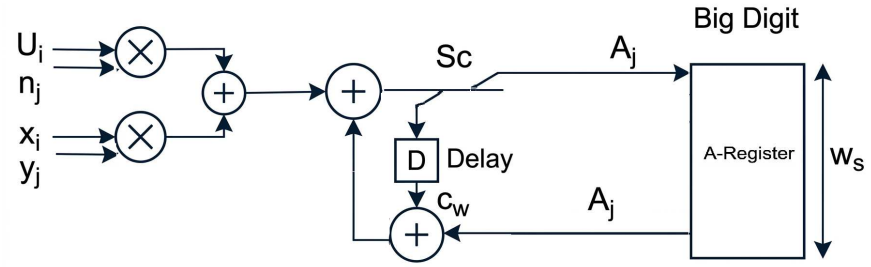


Fig. 15 Add_mult Block diagram.

```
for clk =0 :2*WS+(WS*W)+W% 2*(1+W+(WS*W) +W +2*W )+W
 if clk == 0
 rst= true;
 [output] = Mon_Prod(rst,X(1),Y(1),n(1),fi(0,0,ad,0),start);
 elseif clk<=W
 rst=false;
 [output] = Mon_Prod(rst,X(clk),Y(clk),n(clk),fi(0,0,ad,0),start);
 else
  [output] = Mon_Prod(rst,X(W),Y(W),n(W),fi(0,0,ad,0),start);
 end
end
A_out_arr= zeros(1,W);

for RD=0:W -1
  [output] = Mon_Prod(rst,X(1),Y(1),n(1),fi(RD,0,ad,0),start);
  A_out_arr(1,RD+1)=output;
end
dec2hex(A_out_arr)
```

Fig. 16 The test bench for the Montgomery Multiplication Algorithm

Figure 17 illustrates the script used to test the Montgomery multiplication modules. The difference here from the last testbench is that this testbench calls the internal functions in sequence so that the total computation of the algorithm is done inside the tester, but the HDL coder is just implementing one of these internal functions, i.e., the controller, memories, or computation modules.

3.1 Preliminary Design Results

There are three configuration factors. This opens the door to a variety of different specifications. First, we will study the most famous key length = 1024 bits.The architecture processes 8-bit word sizes ($data_width = 8$), yielding $W = 128$ words for 1024-bit modulus length. The top-level Montgomery multiplier (Mon) utilizes approximately 18,600 LUTs. The arithmetic core (Mul) is the primary driver of logic complexity, accounting for roughly 59% of total logic due to internal wide-word addition and alignment hardware. The memory block (Mem) accounts for 31% to handle multiplexing, word alignment, and register management. The control module (Cont) consumes negligible logic ($< 1\%$) as shown in Figure 18.

The hardware synthesis results exactly consume 4 DSP48E1 slices, which are instantiated inside the (Mul) module to handle the 8 bits $\times$ 16 bits partial product operations. Because a Virtex-7 DSP48E1 slice natively supports up to a 25×18-bit two's complement multiplication, a single 8×16-bit operation fits cleanly within one DSP block without requiring multislice cascades or logic-based expansion. These 4 DSP blocks directly propagate to the top-level Mon module as shown in Figure 19.

```
for clk =0 : 2*WS+(WS*W)+W
 if clk == 0
 rst= true;
 start=false;
 mul_done=false;
 OutRead_addr= fi(0,0,ad,0);
 Aadd=fi(zeros(serialization,WS),0,data_width,0);
  [Ld,restart,m_s,rdX,rdYN,i_cnt,address_wr,Done,WrY,Enable_mul] = MonProd_cont(rst, start, mul_done);
  [Xout, Yout, nout, n0out, Y0out,output] = MonProd_Mem(rst,Ld,WrY,mul_done,rdX,rdYN,address_wr, ...
      i_cnt,m_s,OutRead_addr,Aadd,X(i_cnt+1),Y(address_wr+1),n(address_wr+1));
 elseif clk<=W

rst=false;
[Ld,restart,m_s,rdX,rdYN,i_cnt,address_wr,Done,WrY,Enable_mul] = MonProd_cont(rst, start, mul_done);
[Xout, Yout, nout, n0out, Y0out,output] = MonProd_Mem(rst,Ld,WrY,mul_done,rdX,rdYN,address_wr, ...
    i_cnt,m_s,OutRead_addr,Aadd,X(clk),Y(clk),n(clk));
[Aadd,mul_done] = MonProd_mul(restart, i_cnt, m_s, rst,Enable_mul,Xout,Yout,nout, ...
    n0out, Y0out);
 else
 rst=false;
[Ld,restart,m_s,rdX,rdYN,i_cnt,address_wr,Done,WrY,Enable_mul] = MonProd_cont(rst, ...
    start, mul_done);
[Xout, Yout, nout, n0out, Y0out,output] = MonProd_Mem(rst,Ld,WrY, ...
    mul_done,rdX,rdYN,address_wr, ...
    i_cnt,m_s,OutRead_addr,Aadd,X(W),Y(W),n(W));
[Aadd,mul_done] = MonProd_mul(restart, i_cnt, m_s, rst,Enable_mul,Xout, ...
    Yout,nout, n0out, Y0out);
 end
end
A_out_arr= zeros(1,W);
for RD=0:W -1
  [~, ~, ~, ~, ~,output] = MonProd_Mem(rst,Ld,WrY,mul_done,rdX,rdYN,address_wr, ...
      i_cnt,m_s,fi(RD,0,ad,0),Aadd,X(W),Y(W),n(W));
  A_out_arr(1,RD+1)=output;
end
dec2hex(A_out_arr)
```

Fig. 17 The test bench for each module

The design utilizes 2 BRAM primitives, localized entirely within the (Mem) module to store intermediate word state vectors during processing. The Input/Output Pin Expansion (Bonded IOBs), when synthesized individually out-of-context, both (Mem) (1,150) and (Mul) (1,100) report high IOB consumption. This artifact occurs because standalone module synthesis exposes internal 1024-bit/word-wide buses directly to physical FPGA pins. In the integrated top-level Mon, internal buses are fully encapsulated. The IOB requirement drops to 50 pins for external handshaking, clocking, reset, and serial/word-level input/output transfers. The control module (Cont) achieves a maximum frequency of 550 MHz, demonstrating that finite-state machine transitions do not limit performance. The memory subsystem (Mem) operates up to 330 MHz. The primary timing bottleneck lies in the multiplier module (Mul), which limits the maximum operating frequency to 92 MHz. Long multi-operand addition chains and carry-propagation paths during word multiplication dominate the critical path delay. Integrating the modules into the top-level (Mon) introduces slight routing delays, dropping the overall maximum operating frequency to 78 MHz as shown in Figure 20.

To further increase the target operational frequency, inserting pipeline registers within the carry-propagation path of the (Mul) module and at inter-module boundary crossings in Mon is recommended.

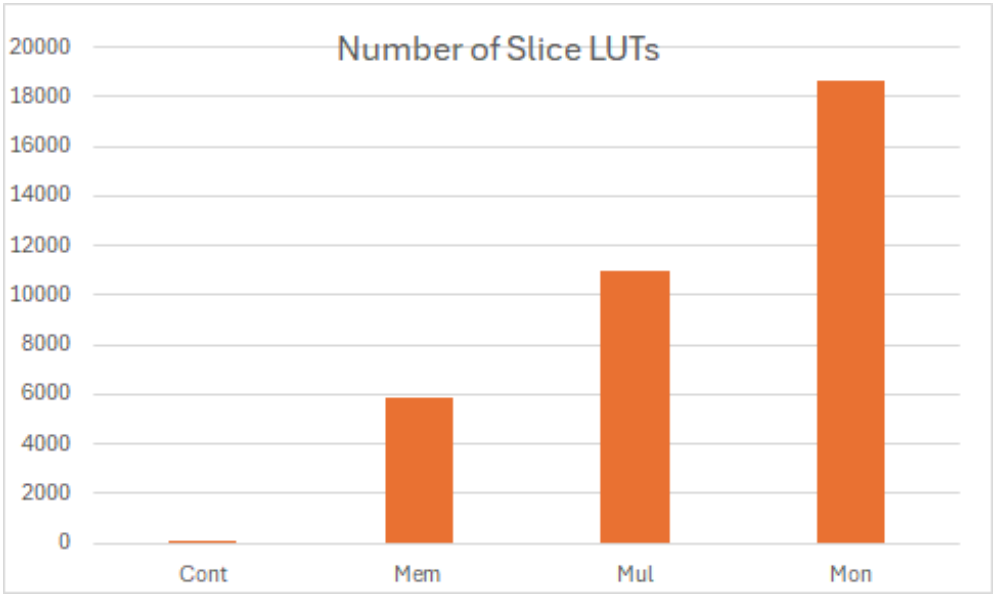


Fig. 18 Number of LUTs

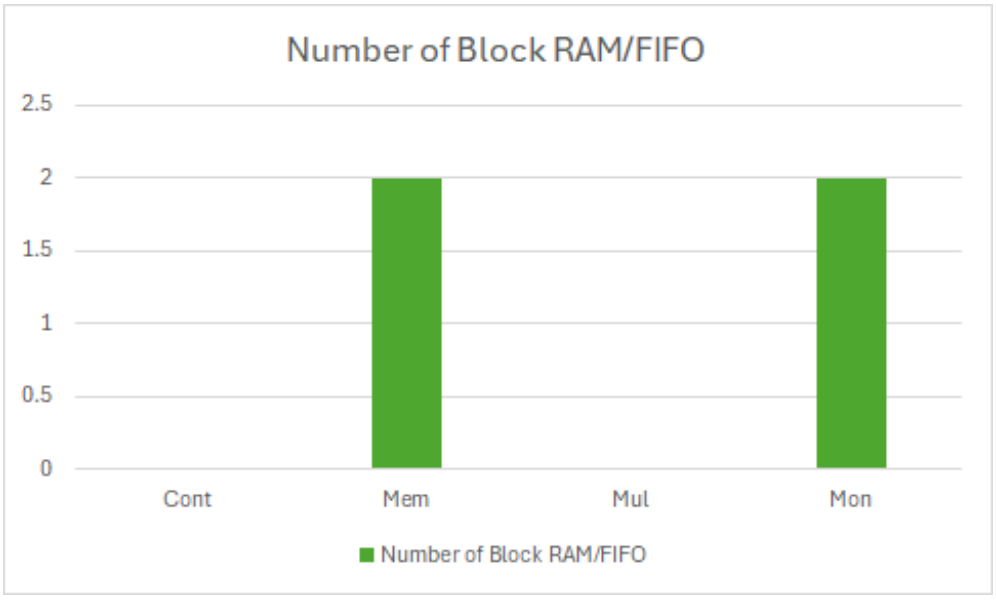


Fig. 19 Number of Block RAMs/FIFOs

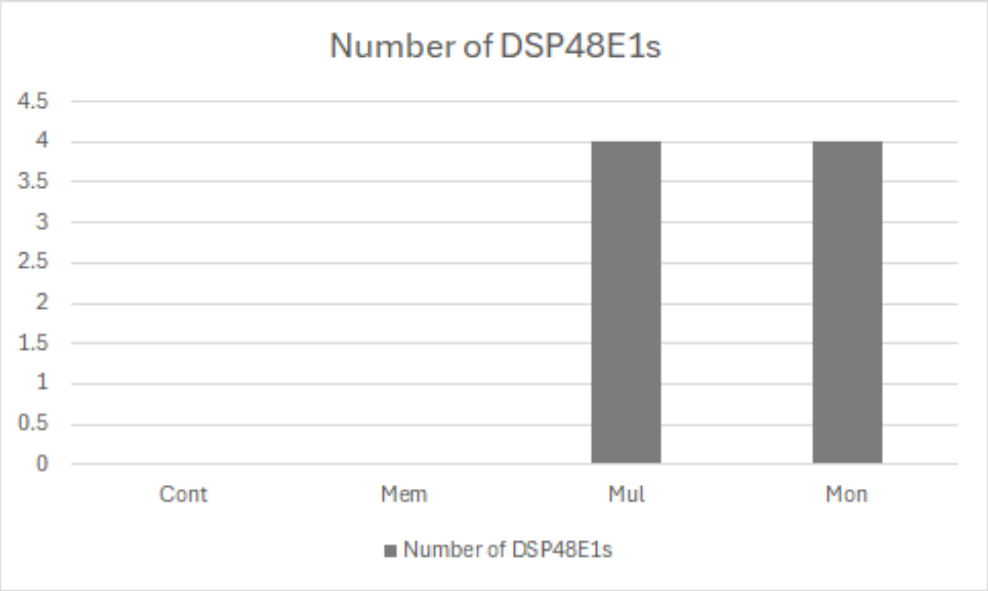


Fig. 20 Number of DSBs

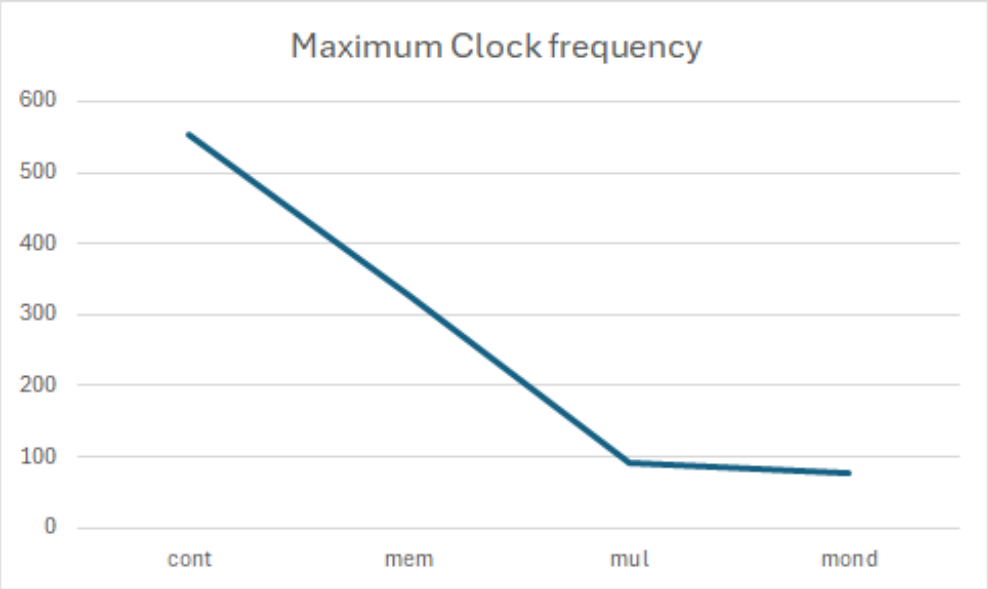


Fig. 21 Maximum Frequency (MHz)

To compare the effect of modularity on the design, we studied the 4069-bit modulus length with 8-bit data_width and 2 serialization. The comparison is done by comparing the documented clock frequency and area utilization [18] for the same platform and the same design tool, i.e., MATLAB HDL Coder. The same figures are repeated for these configurations as illustrated in Figures 21, 22, 23, 24, and 25.

Scaling the modulus length to 4096 bits with a word size of 8 bits ($W = 512$) significantly increases the demand for registers and LUTs. The top-level module (Mon) consumes approximately 60,500 LUTs and 28,800 Slice Registers. The multiplier (Mul) remains the largest logic consumer (37,500 LUTs, 12,300 Registers) due to multi-operand accumulators, shift registers, and wide carry-propagate adders required for 4096-bit Montgomery reduction. The memory subsystem (Mem) consumes 26,500 LUTs and 16,800 Registers to support 512-word address decoding, word alignment, and serialization control logic. The control module (Cont) consumes negligible resources ($< 0.5\%$ total logic). The design utilizes exactly 4 DSP48E1 primitives located within the computational module (Mul). These blocks process partial products for 8 bits $\times$ 16 bits multiplication operations and map directly to the top level (Mon) without additional overhead. In standalone synthesis, (Mem) consumes 1 BRAM block to hold intermediate 4096-bit vector states. At the top-level integration (Mon), total consumption increases to 2 BRAM blocks to support dual-port ping-pong buffering and simultaneous read/write handshaking between (Mem) and (Mul) across execution phases. The control logic (Cont) operates at 550 MHz, confirming that control flow and state transitions contribute zero-latency overhead to the operational critical path. The (Mem) module achieves a maximum operating frequency of 320 MHz, demonstrating that word fetch and store operations do not choke memory throughput. The system bottleneck lies entirely inside the computational module, which limits the local operating frequency to 82 MHz. The critical path is dominated by multi-operand carry propagation and modular reduction accumulation paths. Integrating the modular blocks into the top-level design (Mon) reduces maximum clock frequency from 82 MHz to 70 MHz due to global interconnect and boundary routing delays between the (Mem) and (Mul) blocks. Inserting pipeline registers along the inter-module boundary path can mitigate this frequency drop.

From these results, an interesting note appears. The results illustrate that the computational module controls performance. This raises an important question: Does the computational module's performance depend on the change of the key length? So, the design is configured for two extra key lengths: 1024-bit and 64-bit. The results are illustrated in Figure 26. It is obvious that as the key length increased, the speed decreased. Based on this finding, the critical path is inside the computational module, especially in the feedback from the computed output to the input. In the following section, a proposal for the pipelining technique using the MATLAB HDL Coder is illustrated in detail.

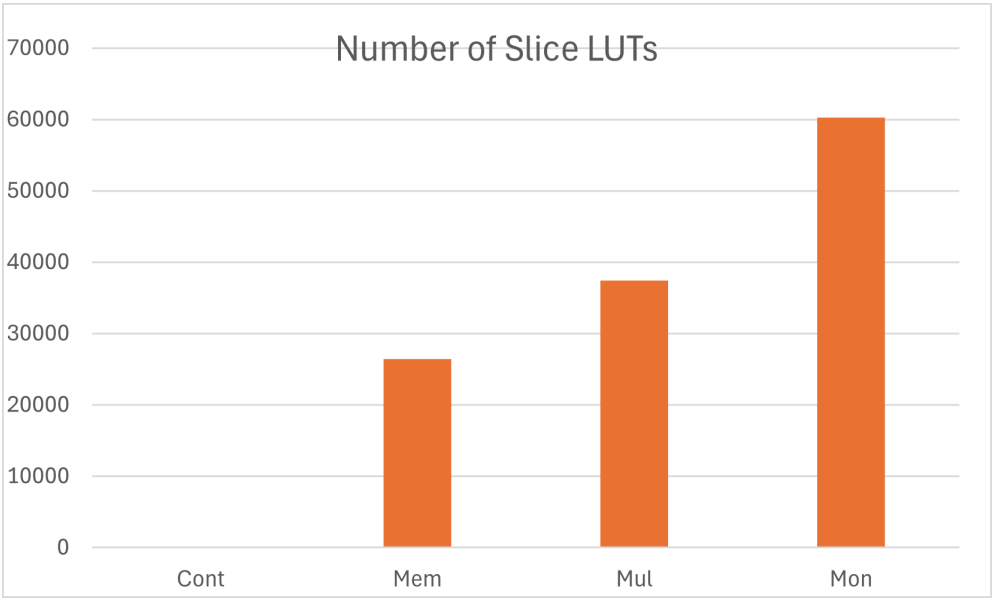


Fig. 22 Number of Slice LUTs 4096_8_2

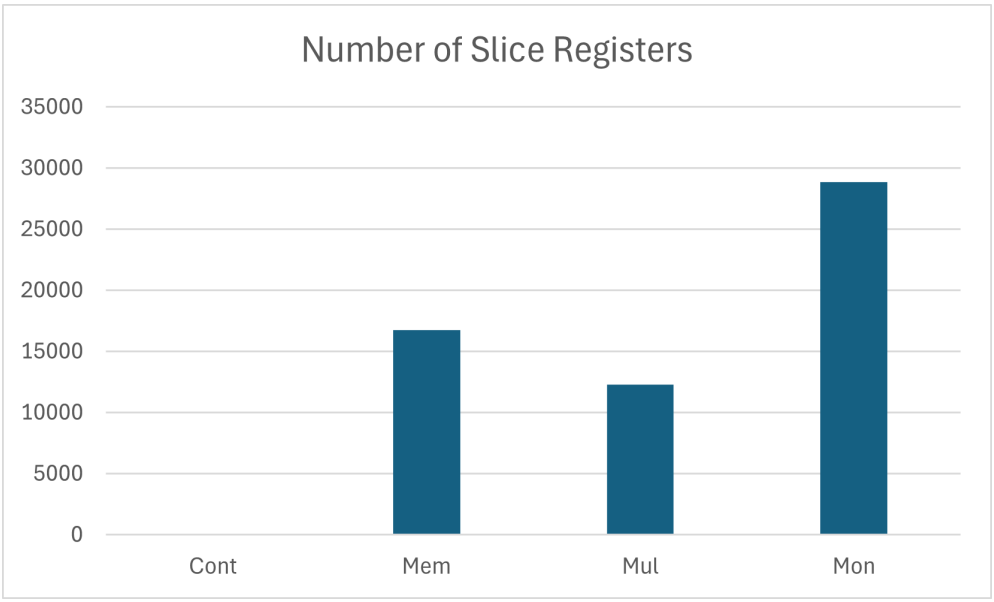


Fig. 23 Number of Slice Registers 4096_8_2

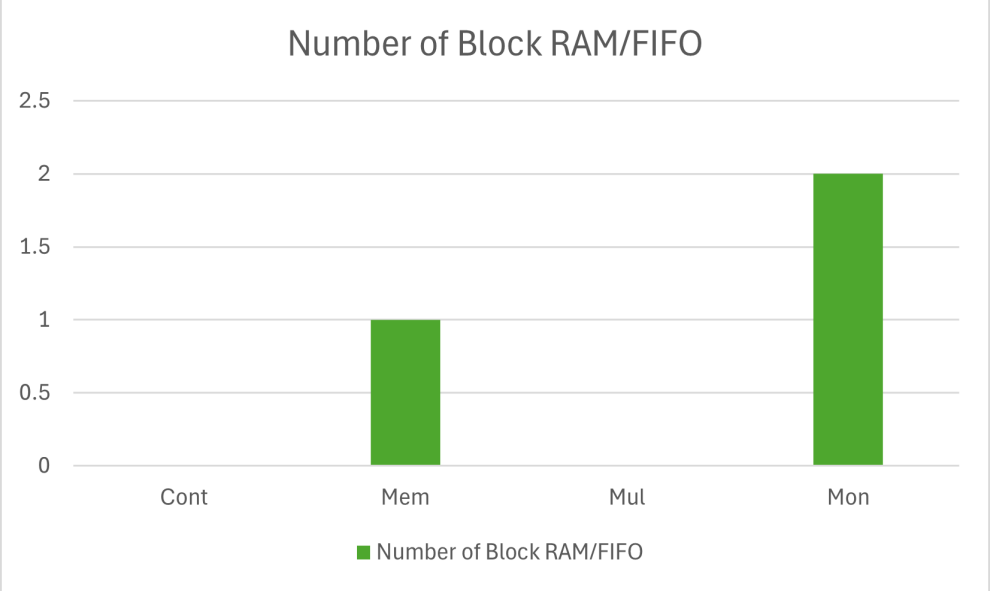


Fig. 24 Number of Block RAMs/FIFOs 4096_8_2

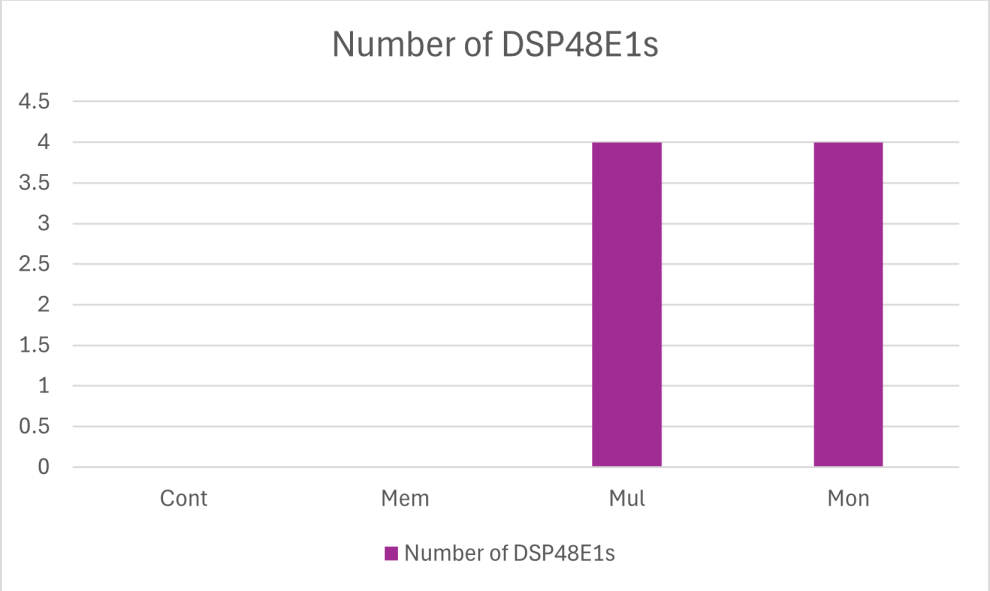


Fig. 25 Number of DSBs 4096_8_2

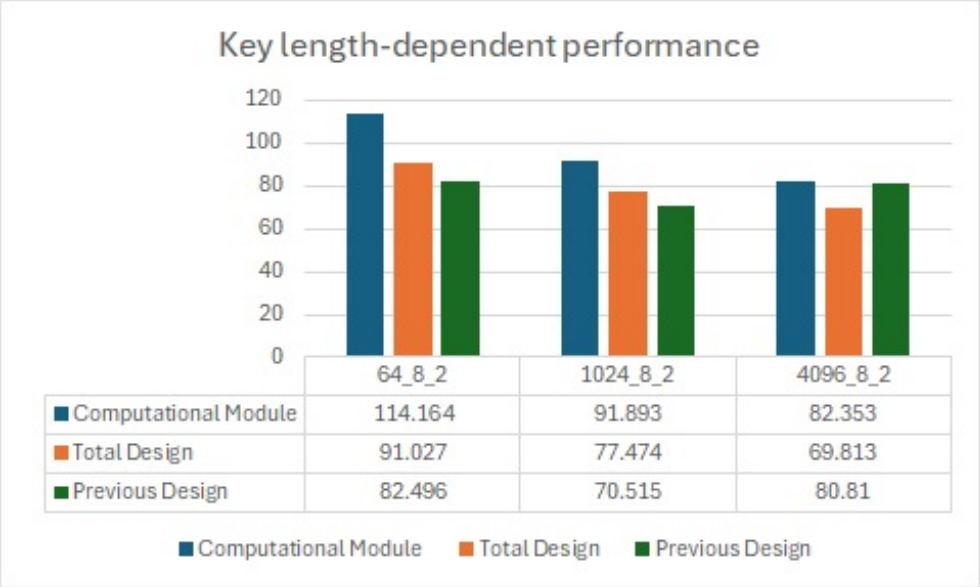


Fig. 26 Key length-dependent f_{max} in MHz

4 Optimized Architecture

As the most architecturally complex component of the design, the computational module introduces a dominant combinatorial delay along the critical path, necessitating explicit pipeline optimization. Preliminary experiments utilizing *hdl.delay* directives and separate register sub-functions proved unsuitable for automated HDL code generation. Instead, optimal synthesis is achieved by modeling pipeline registers directly using MATLAB persistent variables, which reliably infer physical hardware flip-flops during synthesis.

This section details the modeling and implementation of these pipeline stages within MATLAB for HDL Coder synthesis. A basic arithmetic example establishes the foundational concepts of data buffering and multi-cycle latency, demonstrating a two-clock-cycle delay pipeline to illustrate how signals are registered, processed, and output across active clock edges. Subsequently, this methodology is applied to the main computational module, progressing from its structural block diagram to a line-by-line code implementation. Finally, the section details the control logic adaptations required in the controller module to accommodate the pipeline's cycle latency.

4.1 Demonstartion of pipeline in MATLAB HDL Coder

To demonstrate this register inference mechanism, a basic test function was developed to increment an input signal while introducing a deterministic latency of two clock cycles. The corresponding MATLAB script, its companion verification testbench, and the resulting simulation are illustrated in Figure 27.

addone.m

```
function [outputArg1] = addone(inputArg1)
%UNTITLED Summary of this function goes here
%   Detailed explanation goes here

persistent tmp1;
if isempty (tmp1)
    tmp1=0;
end
persistent tmp2;
if isempty (tmp2)
    tmp2=0;
end
outputArg1 = tmp2;
tmp2=tmp1+1;
tmp1=inputArg1;
end
```

test_persistent.m

```
A=10:10:100;
 for i=1:10
B(i)=addone(A(i));
 end
[A', B']
```

Command Window

```
>> test_persistent

ans =

    10     20     30     40     50     60     70     80     90    100
     0      1     11     21     31     41     51     61     71     81
```

Fig. 27 Demo test for Persistent variable.png

As a demonstration, the *addone* function models a two-cycle latency pipeline that increments its input by one, where each function call corresponds to a single system clock cycle. To verify this behavior, a dedicated testbench (test_persistent) sweeps the input from 10 to 100 in steps of 10. Upon reset, the internal persistent variables (*temp*1 and *temp*2) initialize to 0. On subsequent calls, the primary input port (In) feeds directly into the first D flip-flop register (temp1), introducing a single clock cycle delay

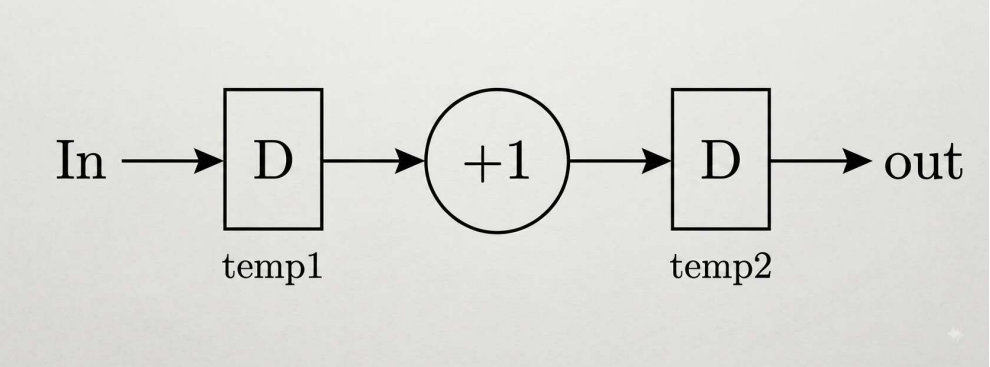


Fig. 28 Block diagram of the two-stage pipelined incrementer circuit

(D). The registered signal from temp1 then passes through a combinational adder stage that increments the value by one (+1). This intermediate result is subsequently captured by the second D flip-flop register (temp2) on the following active clock edge before driving the final output port (out). Overall. The most important point here is the sequence of these steps in the code. If you are thinking software-wise, you will arrange the sequence in reverse. The execution sequence over the initial clock cycles illustrates the pipeline's register-transfer behavior. In the first cycle, the output yields the initial value of *temp*2 (0). Subsequently, *temp*2 updates to 1 (*temp*1 + 1), while *temp*1 captures the first input value (10). In the second cycle, the output reads 1 (*temp*2's updated state). *temp*2 then advances to 11 (10 + 1), and *temp*1 captures the second input (20). By the third cycle, the output reads 11, *temp*2 updates to 21 (20 + 1), and *temp*1 registers the third input (30). This sequential register shift and increment operation continues predictably for all subsequent clock cycles. The architecture exhibits a total pipeline latency of two clock cycles, resulting in the transfer function $\text{out}[n] = \text{In}[n-2] + 1$. Figure 28 illustrates the hardware block diagram of the two-stage pipelined incremented circuit. The output remains 1 (0 + 1) for the first two clock cycles before yielding the expected sequence of 11, 21, 31, ..., 101.

4.2 Pipeline Computational module

To reduce the combinatorial delay, a register is added in all feedback paths. The optimized block diagram for the *Ui* block is illustrated in Figure 29; part a) of the figure illustrates how Ui is calculated before the pipeline, and part b) illustrates where the pipeline registers are added. Multiplication between Xi and Y0 results in double the data_width; only the least significant data_width is needed. This will be added to the first word of A, A0, and then multiplied by the modular inverse of n, n0. The needed value is also the first least significant data_width. To pipeline, a register is added after the two multiplications and the adder. Uit is the output of the first multiplier's register; Uit1 is the output of the adder's register, and Uit2 is the output of the final multiplier's register. The way this is written in the code requires focusing on

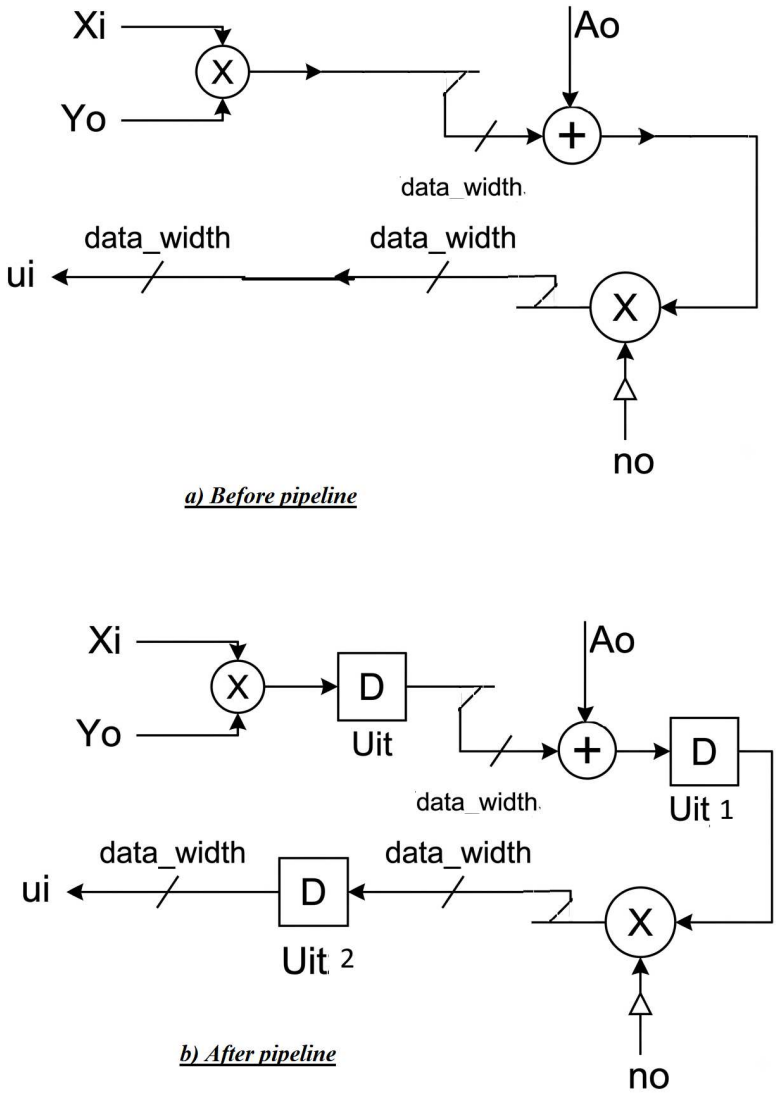


Fig. 29 Optimized Multiply and Add Block Diagram

the sequence of writing updates to the persistent statements. The sequence of writing the code is illustrated in Figure 30.

Taking into consideration that the software updates the variable line by line inside the function and all these updates are done within one clock cycle, the sequence of code is reversed. The last multiplication is updated first, then the adder, and finally the first multiplication, so that the correct answer appears after three clock cycles. This introduces three clock cycles of latency compared to the original design. This means that the output will be ready after 3 clock cycles, later than the other inputs

```
uit2=multmodb(uit1,n0out,data_width);
ui=uit2;
uit1=addmodb(uit,Aout0,data_width);
uit=multmodb(Xj,Y0out,data_width);
function [z] = multmodb(x,y,data_width)
%UNTITLED Summary of this function goes here
%   Detailed explanation goes here
xi= fi(x,0,data_width,0);
yi= fi(y,0,data_width,0);
zi = xi* yi;
z= bitsliceget(zi,data_width,1);
end
```

Fig. 30 Pipelining the Ui Block

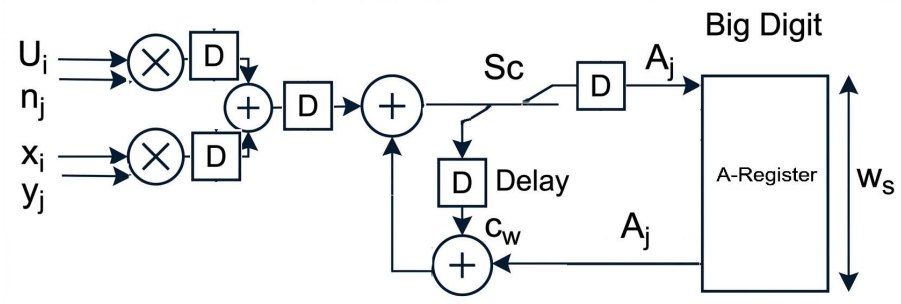


Fig. 31 Pipelining the Add_mult Block

to the next block. The controller module is modified to cope with the change. The next Block is the multiply-and-add block. From equation (2), two multipliers can be implemented in parallel in one clock cycle, followed by the addition step in another clock cycle. The block diagram is illustrated in Figure 31. So, this block needs 2 extra clock cycles. The suffices of the inputs i and j indicate the two counters in the control module: *i_cnt* and *m_s*. This means that all the inputs *Ui, nj, Xi* and *Yj* should be ready at the same time, which requires some synchronization on the control module.

4.3 Updated Control Module

Architectural modifications to the computational datapath require dedicated timing adjustments within the Finite State Machine (FSM) control logic to preserve multi-operand synchronization across iterative processing steps. To compensate for the two-clock-cycle latency ($L = 2$) introduced by the repeated execution of the multi-operand adder-multiplier block (*Add_mult*), the word-indexing counter (m_s) must increment once every two clock cycles rather than on every cycle. This is efficiently realized without introducing extra idle state logic by extending a unified master counter (j_cnt) with an extra Least Significant Bit (LSB), where the remaining higher-order bits form m_s ($m_s = j_cnt[\text{MSB} : 1]$) to naturally divide the increment frequency by two. Furthermore, to satisfy the latency of the three-clock cycle setup ($T_{\text{setup}} = 3$ cycles) required for a valid calculation U_i before reduction, an internal delay counter triggers when j_cnt is in its zero state ($j_cnt = 0$) to stall execution until U_i stabilizes, with the control module eventually exporting both j_cnt and m_s to drive the downstream (Mem) and (Mul) alignment.

5 Results and discussion of the Pipelined Modular Montgomery Multiplication Design

The optimization paradigms for MATLAB HDL Coder differ significantly from standard software engineering metrics such as code brevity or memory footprint. High-performance hardware design often requires expanded, out-of-order code structures to optimize register transfer levels (RTL) and critical paths. Ultimately, maximizing clock speed ($f_{\max}$) comes at the expense of increased pipeline latency. The tester file *for loop* condition is increased to complete the algorithm's computation. Architectural optimizations yielded substantial performance improvements, elevating the maximum operational frequency to 128.810 MHz (+56.5% compared to the previous iteration) for the configuration with 4k *key_length*, 8 bits for *data_width*, and a *serialization* factor of 2. The same platform is used. Table 3 shows the summary of the utilization of the FPGA. The table indicates an increase in the slice registers, the number of slice-LUT, and the number of fully used LUT-FF pairs by 1%, 13% and 7%, respectively. Before pipelining, the baseline latency is expressed as $L_b = W + (WS \times W) + 2 \times WS$ ($8,448$ clock cycles for the tested configuration). After introducing pipeline stages, the latency scales to $L_p = W + W \times (2 \times WS + 3) + 2 \times WS$ ($17,088$ clock cycles). Consequently, the total execution time for a single modular multiplication increases from 104.54 μs in the original design to 132.16μs in the pipelined variant. Although pipelining mitigates the critical combinational path delay and enhances the maximum clock frequency by 56.5%, the doubled cycle count overhead leads to higher total single-operation execution latency.

6 Discussion

Evaluating the modular Montgomery multiplier across 1024-bit ($W = 128$) and 4096-bit ($W = 512$) key lengths on the Xilinx Virtex-7 FPGA (xc7vx330t-2ffg1157) demonstrates how operand scaling impacts logic utilization while dedicated hardware

Logic Utilization	Used	Available	Utilization
Number of Slice Registers	28853	408000	7%
Number of Slice LUTs	65440	204000	32%
Number of fully used LUT-FF pairs	23903	70390	33%
Number of bonded IOBs	48	600	8%
Number of Block RAM/FIFO	1	750	0%
Number of BUFG/BUFGCTRL/BUFHCEs	2	200	1%
Number of DSP48E1s	4	1120	0%

Table 3 Device Utilization Summary

remains bounded. The top-level design (Mon) grows from 18,600 LUTs for 1024-bit operands to 60,500 LUTs (32% device utilization) and 28,853 Slice Registers for 4096-bit operands. This non-linear logic expansion is driven primarily by the arithmetic multiplier block (Mul) and the memory management unit (Mem). DSP consumption remains strictly constant at 4 DSP48E1 primitives across both key lengths. Because a single Virtex-7 DSP block natively supports up to 25×18-bit operations, the 8×16-bit partial multiplications fit entirely within 4 DSP blocks without requiring multi-slice cascading or logic expansion. Increasing the word count ($W = 128 \rightarrow 512$) forces internal vector registers, carry-propagate adders, and reduction accumulators inside (Mul) to scale from 1024 to 4096 bits, driving (Mul) logic usage from 11,000 LUTs to 37,500 LUTs. Addressing 512 words in (Mem) requires wider multiplexing trees and decoding logic compared to 128 words, expanding (Mem) logic consumption from 5,800 LUTs to 26,500 LUTs (16,800 Slice Registers). BRAM allocation highlights the interplay between physical primitive capacity (RAMB36E1) and port contention during modular integration: In the 1024-bit design, the 1 KB vector footprint splits across two 18 Kb primitives (RAMB18E1) to fulfill dual-port address constraints. In contrast, the 4096-bit design (4 KB total vector footprint for X, Y, N, and $Result$) fits entirely within the $36,864$-bit capacity of a single 36 Kb primitive (RAMB36E1), reporting only 1 BRAM in standalone synthesis. At top-level integration, both configurations consume 2 BRAM primitives. Because a single BRAM tile physically provides only 2 hardware ports, running concurrent read operations for multiplicand $Y[j]$ and modulus $N[j]$ alongside write-backs for intermediate accumulator words $Result[j]$ creates port contention. Top-level synthesis resolves this by instantiating a second BRAM block to guarantee dedicated memory ports across processing phases. Frequency analysis reveals that control logic (Cont) operates at 550 MHz and memory access (Mem) operates at 320–330 MHz, establishing that FSM state transitions and memory fetches do not bound execution speed. The critical path resides within the (Mul) core (82–90 MHz), where long carry-propagation paths during multi-operand

reduction limit operational throughput. Top-level interconnect routing (Mon) further degrades operational frequency to 78 MHz (1024-bit) and 72 MHz (4096-bit). While inserting pipeline registers inside the (Mul) datapath mitigates the combinational path and increases maximum clock frequency by 56.5%, it introduces an execution trade-off: Pipelining increases cycle latency from $L_o = W + (WS \times W) + 2 \times WS = 8,448$ cycles to $L_p = W + W \times (2 \times WS + 3) + 2 \times WS = 17,088$ cycles for tested configurations. Despite the frequency gain, total single-operation execution time increases from 104.54 μm (unpipelined) to 132.16 μs (pipelined). This demonstrates that for sequential, single-operation Montgomery implementations, depth pipelining without multi-stream throughput interleaving increases total execution latency due to cycle count To systematically evaluate this trade-off, a custom MATLAB script was developed to automatically generate pipelined register stages across arbitrary stage depths within the proposed design methodology.

7 Conclusion

This study evaluated a modular, word-based Montgomery multiplier on the Xilinx Virtex-7 FPGA across 1024-bit and 4096-bit key lengths ($data_width = 8$ bits). The findings demonstrate key trade-offs between hardware scalability, memory allocation, and pipelining efficiency. The modular architecture maintains a minimal dedicated resource footprint of exactly 4 DSP48E1 blocks and 2 Block RAMs regardless of key size expansion. Scaling from 1024 to 4096 bits impacts only general logic (increasing top-level usage to 60,500 LUTs and 28,800 Slice Registers) to accommodate wider intermediate accumulators and multiplexing networks. The Computational module (Mul) represents the primary hardware bottleneck, limiting the overall frequency of the system to 72 to 78 MHz due to long propagation chains during modular reduction, while memory and control logic run substantially faster ($> 320MHz$).The Pipelining Latency Penalty: Although the addition of pipeline stages mitigates the combinatorial path delay and increases the maximum clock frequency by 56.5% (128.810 MHz), it expands the execution latency from 8,448 to 17,088 clock cycles. As a result, total single-operation execution time increases from 104.54 μs to 132.16 μs.For single-operation cryptographic workloads, the unpipelined architecture yields superior real-time latency performance, whereas Pipeline technology is only useful when combined with multi-flow implementation to reduce cycle costs. To address the combinational delay while managing these execution trade-offs, this study proposes an automated method using a custom MATLAB script to generate and integrate pipelined register stages into synthesizable VHDL via MATLAB HDL Coder.

Supplementary information. Not applicable

Acknowledgements. The authors thank the British University in Egypt (BUE) for supporting us with the MATLAB HDL Coder.

Declarations

Ethics approval and consent to participate

Not Applicable. . .

Consent for publication

Not applicable. . .

Availability of data and materials

We provide the code and the results in this manuscript. Any required data or materials are available if needed.. . .

Competing interests

The author declares that they have no competing interests.

Funding

No funding was obtained for this study . . .

Authors' contributions

GE is the sole author of the manuscript. She has read and approved the manuscript.. . .

Acknowledgements

The authors would like to acknowledge the British University in Egypt (BUE), Egypt, for supporting us with the simulation tool (MATLAB). . . .

Authors' information

Ghada Farouk Naiem Elsayed ' Biography

GHADA ELSAYED has received her BSc, MSc, and PhD degrees in electronic and communication engineering, 2001, 2005, and 2008, respectively. She worked for the Egyptian Space Program for the first eighteen years, then she shifted to an academic career, in which she worked for more than five years at MSA University, and then she travelled as a visiting researcher to Japan, Kyushu Institute of Technology. After that, she continued on the same path at AUC, MTI, and then BUE University. In 2014, Ghada published a book about securing the satellite control link. She also published many scientific papers.. . .

8 References

References

[1] The MathWorks, Inc., "Synthesize Generated HDL Code from MATLAB Algorithms," MathWorks Help Center, 2026. [Online]. Available: https://www.mathworks.com/help/hdlcoder/gs/synthesize-generated-hdl-code-from-matlab-algorithms.html [Accessed: July 26, 2026].

[2] MathWorks. Basic HDL Code Generation Workflow, MATLAB & Simulink. Available at: https://www.mathworks.com/help/hdlcoder/gs/hdl-code-generation-workflow.html (Accessed: 27 May 2026).

[3] S. P. Kumar and P. A. Kumari, "An FPGA based architecture for Convolution Filtering using MATLAB and HDL Coding Technique," *i-Manager's Journal on Embedded Systems*, 2013.

[4] S. G. Pasha and V. V. Kohir, "OFDM based DVB-T system implementation using MATLAB and HDL coder," in *2017 International Conference on Computer Communication and Informatics (ICCCI)*, 2017.

[5] K. Thesni, K. Praveen, and L. Srivani, "Implementation and performance comparison of digital filter in FPGA," in *2020 6th International Conference on Control, Automation and Robotics (ICCAR)*, 2020.

[6] H. Pant, H. Bourai, and G. S. Rana, "Conversion of MATLAB code in VHDL using HDL Coder & Implementation of Code on FPGA," *HCTL Open International Journal of Technology Innovations and Research*, 2015.

[7] P. Banerjee, M. Haldar, A. Nayak, and V. Kim, "Overview of a compiler for synthesizing MATLAB programs onto FPGAs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 1, 2004.

[8] P. Banerjee, "An overview of a compiler for mapping MATLAB programs onto FPGAs," in *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, 2003.

[9] M. Haldar, A. Nayak, and N. Shenoy, "FPGA hardware synthesis from MATLAB," in *14th International Conference on VLSI Design*, 2001.

[10] M. Haldar, A. Nayak, and A. Choudhary, "A system for synthesizing optimized FPGA hardware from MATLAB (R)," in *IEEE/ACM International Conference on Computer Aided Design*, 2001.

[11] J. Han, "Application of MATLAB for FPGA design," in *2011 2nd International Conference on Artificial Intelligence, Management Science and Electronic Commerce (AIMSEC)*, 2011.

[12] T. M. Bhatt and D. McCain, "MATLAB as a development environment for FPGA design," in *Proceedings of the 42nd annual Design Automation Conference*, 2005.

[13] G. Elsayed, E. Soleit, and S. Kayed, "FPGA design and implementation for adaptive digital chaotic key generator," Bulletin of the National Research Centre, vol. 47, no. 1, p. 122, 2023.

[14] MathWorks. (2022, October 28). What Is HDL Coder? [Video]. YouTube (0:00). https://www.youtube.com/watch?v=0v_g0bzs4_c

[15] Xilinx, Inc. (2020). Vivado Design Suite user guide: High-level synthesis (UG902). bht-berlin.de

[16] G. Elsayed and S. I. Kayed, "A Comparative Study between MATLAB HDL Coder and VHDL for FPGAs Design and Implementation," Journal of International Society for Science and Engineering, vol. 5, no. 1, pp. 1-8, 2022.
[17] S. I. Kayed and G. Elsayed, "Optimizing Techniques for using MATLAB HDL CODER," presented at the AEAS 2022 Conference (presumably hosted by Ain Shams University), 2022.
[18] Elsayed, G. and Abbas, E. S. Dr. (2024). "FPGA design and implementation for Montgomery multiplication algorithm using MATLAB HDL coder." Computer Networks. (Scalable designs up to 4096-bit on Virtex-7).
[19] Huang, M., Gaj, K., & El-Ghazawi, T. (2011). "New Hardware Architectures for Montgomery Modular Multiplication Algorithm." IEEE Transactions on Computers. (Seminal paper with high-frequency results on Virtex-5/7).
[20] Erdem, S. S., Kaps, J. P., & Koç, Ç. K. (2017). "A General Digit-Serial Architecture for Montgomery Modular Multiplication." IEEE Transactions on VLSI Systems. (General digit-serial approach across multiple Virtex/Artix generations).
[21] Khan, S., Awwad, A. R., & Abdalla, A. A. (2018). "High-speed FPGA implementation of full-word Montgomery multiplier for ECC applications." Microprocessors and Microsystems. (High-speed designs on Virtex-7/6).
[22] Al-Shalabi, M. S., et al. (2017). "A Scalable and Highly Efficient Montgomery Modular Multiplier on FPGAs." IEEE Transactions on Circuits and Systems II. (Scalable architectures on Virtex-6).
[23] Abd-Elkader, A. A. H., et al. (2020). "FPGA-Based Optimized Design of Montgomery Modular Multiplier." IEEE Access. (Low-area, high-efficiency designs on Virtex-6).
[24] Jain, A., & Gupta, A. (2023). "FPGA Implementation of Enhanced Montgomery Modular for Fast Multiplication." ITM Web of Conferences. (Radix-4 architecture on Artix-7).
[25] Venkatesh, N., & Vasantha, M. (2020). "A resource-efficient Montgomery modular multiplier on FPGA." Journal of Ambient Intelligence and Humanized Computing. (Resource-efficient designs on Artix and Spartan families).
[26] Morales-Sandoval, M., et al. (2015). "High-performance modular exponentiation based on Montgomery multiplication for RSA on FPGAs." Microprocessors and Microsystems. (Focused on Spartan architectures).
[27] Kuang, S., et al. (2013). "Energy-Efficient High-Throughput Montgomery Modular Multipliers for RSA Cryptosystems." IEEE Transactions on VLSI Systems. (Energy-efficient designs on Spartan and Virtex).
[28] A. C. B. C. A. C. & P. S. S. A. (2018). "High-performance Pipelined FPGA Implementation of Montgomery Modular Multiplier." SciTePress. (High-speed pipelined designs on Virtex-7 and Virtex-5).
[29] Abd-Elkader, A. A. H., et al. (2021). "A compact FPGA-based Montgomery Modular Multiplier." Indonesian Journal of Electrical Engineering and Computer Science. (Cited for Virtex-6 and area-optimized designs).
[30] Liu, T., et al. (2023). "A Low-Cost High-Performance Montgomery Modular Multiplier Based on Pipeline Interleaving for IoT Devices." MDPI Electronics. (High-performance design on Virtex-7/Zynq for IoT).

[31] Sharma, B., & Singh, R. (2024). "A High-Speed Hardware Algorithm for Modulus Operation and its Application in Prime Number Calculation." arXiv. (Performance on Zybo/Artix-7 fabric).
[32] Gaj, K., Gaj, P., & Kaps, J. P. (2004). "FPGA Montgomery multiplier architectures - a comparison." 12th Annual IEEE Symposium on FCCM. (Early comparative work on Virtex-2 Pro and similar older generations).
[33] Montgomery, P. L. (1985). "Modular Multiplication Without Trial Division." Mathematics of Computation, Vol. 44, No. 170, pp. 519–521.

