Hardware RSA Accelerator

Group 3: Ariel Anders, Timur Balbekov, Neil Forrester

April 29, 2013

1 Overview

Our project is implementing the RSA cryptographic algorithm in Bluespec. The benefits of doing this in hardware are higher performance, reduced power usage and size, and cost. Having reusable IP that implements RSA would allow a device manufacturer to either reduce their power footprint, or skip inclusion of a processor in a device that otherwise would not need one.

An example application of our preliminary proposal could be an intelligence agencies' covert listening device with the added ability of secure communication through RSA protocol. Specialized hardware is useful here because the device needs to be small, low power, with the ability to run for long periods of time.

Alternatively, suppose you were designing a high performance router to create a secure VPN between remote sites, so that it appears that all the computers at all sites are on the local network. Keeping latency as low as possible, and throughput as high as possible, would be vital. Hardware support for Public Key Cryptography, such as RSA, could play an essential component in developing this router.

The main challenges we foresee in implementing RSA in Bluespec are creating a multi-precision arithmetic library with support for modulo, exponentiation, and multiplication. Once those problems are solved, the only remaining issue is writing a sensible interface.

2 Algorithms

All of our high-level RSA modules will be built around a single module that does modular exponentiation. This module will employ the Right-to-left binary algorithm, which we believe is a good compromise between speed, memory usage, and complexity. The goal of the algorithm is to calculate $b^e \bmod m$ for very large values of b, e, and m. If the bits of e are $e_1, e_2 \ldots e_n$:

$$e = \sum_{i=0}^{n} e_i 2^i \tag{1}$$

then:

$$b^e = \prod_{i=0}^{n} e_i b^{(2^i)} \tag{2}$$

and since:

$$a * b \bmod m = (a \bmod m) * (b \bmod m) \bmod m \tag{3}$$

then every intermediate result can be taken modulo m to keep the size of intermediate results manageable. Therefore, the following algorithm will compute $b^e \bmod m$ in a reasonable amount of time and memory:

```
$b$, $e$, and $m$ are the inputs to the algorithm.
$c \leftarrow 1$
**while** $e > 0$ **do**
    **if** $e \bmod 2 = 1$ **then**
        $c \leftarrow c * b \bmod m$
    **end if**
    $b \leftarrow b * b \bmod m$
    $e \leftarrow \lfloor e/2 \rfloor$
**end while**
$c$ is the result of the algorithm.
```

This very naturally suggests a circular pipeline in hardware. If parallelism is desired, then multiple circular pipelines may be put in parallel, with some logic at the front and back to manage handing out jobs to different circular pipelines, and collecting the results.

The only remaining problem is performing multiplication, modulo, and bit shifting which is implemented through the interleaved modular multiplication algorithm which requires bit shifts, additions, subtractions, and bitwise comparisons and does not take up excessive area.

Here is the algorithm for interleaved modular multiplication. N is the size of the numbers, in bits. For example, $N = 1024$. Also, x_i is the ith bit of x.

```
$x$, $y$, and $m$ are the inputs to the algorithm.
$p \leftarrow 0$
$i \leftarrow N - 1$
**while** $i \geq 0$ **do**
```

```
        $p \leftarrow p * 2$
        **if** $x_i = 1$ **then**
            $p \leftarrow p + y$
        **end if**
        **if** $p \geq m$ **then**
            $p \leftarrow p - m$
        **end if**
        **if** $p \geq m$ **then**
            $p \leftarrow p - m$
        **end if**
        $i \leftarrow i - 1$
    **end while**
    $p$ is the result of the algorithm.
```

3 Implementation in C

We have a working implementation of all our algorithms in C, that we wrote from scratch and verified its correctness with libgcrypt. The C implementation, of course, is unable to operate directly on 1024 bit integers, so we store them as arrays of 16 bit unsigned integers. And implement the RSA algorithm in chunks. As a result, performing bit shifts, additions, and comparisons takes somewhat more code than it would take to perform the corresponding operations in Bluespec. However, since we have now written a C implementation that does all the operations on chunked integers, we now feel confident in our ability to pipeline or otherwise break up any operations we find to be too big or long in our Bluespec implementation.

4 Microarchitecture

Our project is divided into two important modules: `ModExpt.bsv` and `ModMultIllvd.bsv`. `ModExpt.bsv` performs modular exponentiation, while `ModMultIllvd.bsv` performs modular multiplication using the interleaved modular multiplication algorithm described above. The modular exponentiator instantiates two modular multipliers. The high level diagram in Figure 1 depicts the interface between the modular multipliers and the modular exponentiator (though only one multiplier is shown for simplicity).

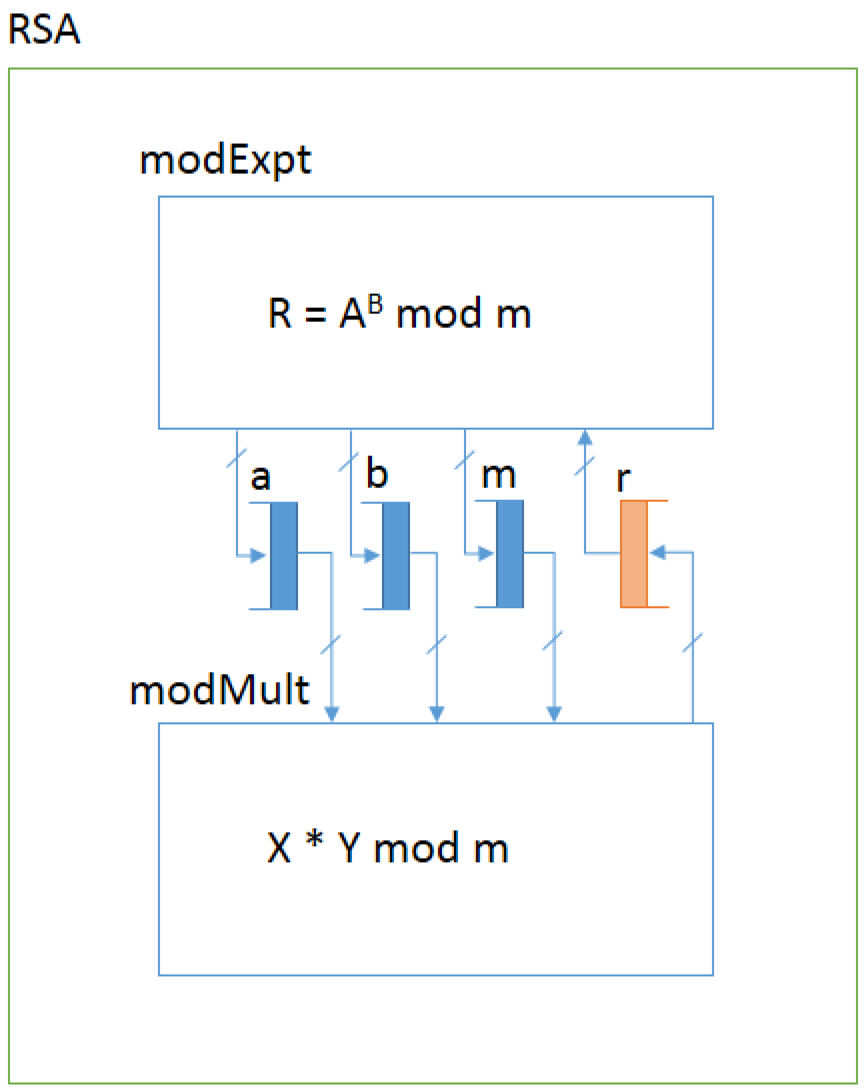

Figure 1: High level overview. Note that only one of two multipliers is depicted.

The current plan is that large integers of (for example) 1024 bits will be represented in Bluespec as `Int#(1024)`. This will probably be practical, as we never instantiate combinational multipliers on these types, only bit shifts, adders, and comparators, which shouldn't take up much area. However, performing comparisons and arithmetic on long bit lengths will adversely affect the cycle time of the design. In performance critical modules, like the modular multiplier, it will be highly beneficial to operate on the string in chunks: for example, the 64 hardware DSP48 blocks can synthesize a long add/subtract chain using internal routing resources.

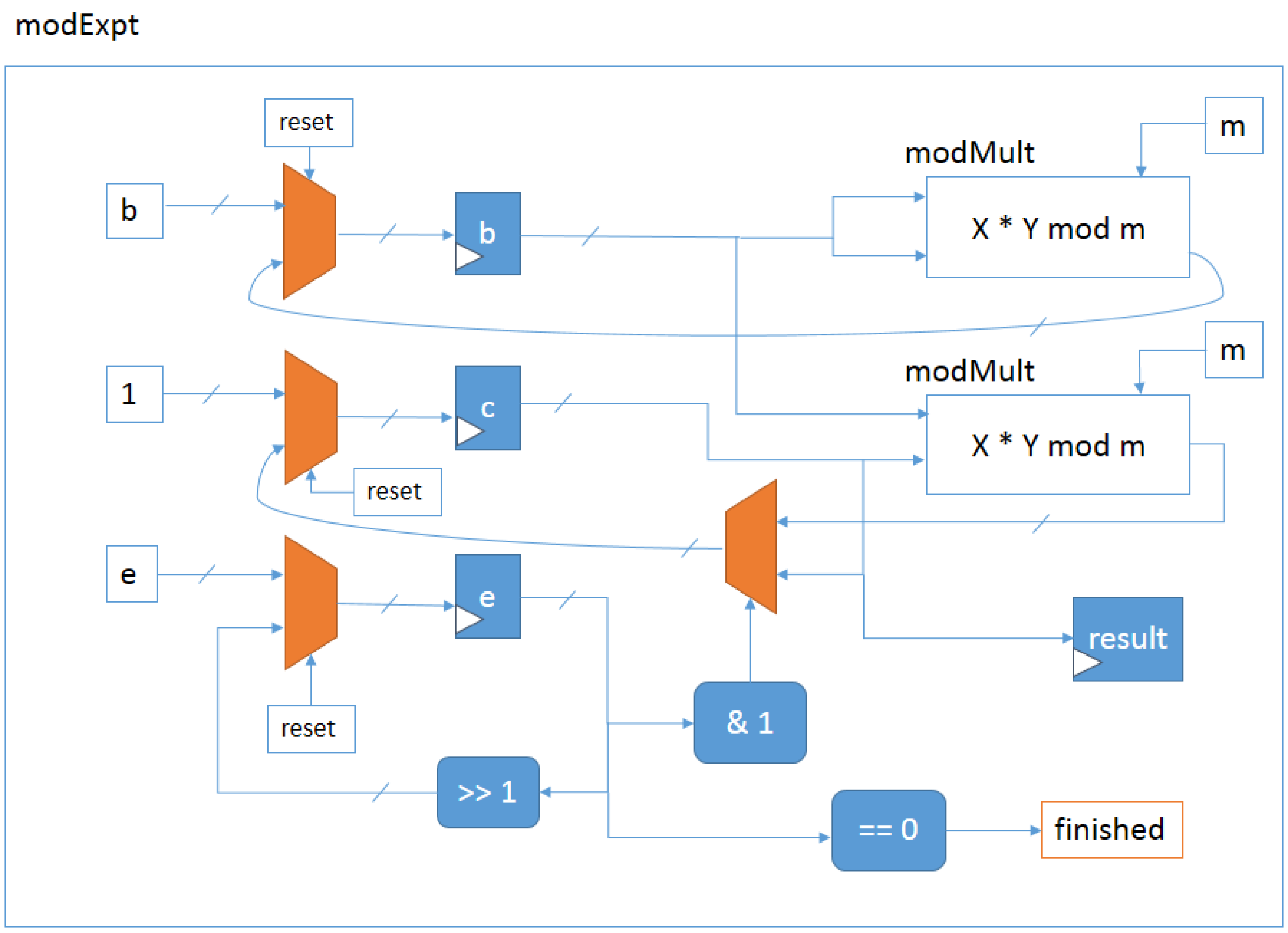

Figure 2: Modular exponentiation

4.1 Right to Left Binary Modular Exponentiator

The modular exponentiator is a circular pipeline (depicted in Figure 2). On each cycle of the pipeline it supplies inputs to the two multipliers. When the multipliers complete, it stores the results back into the registers. However one result is discarded if the low bit of `e` is 0. In fact, our actual implementation will probably not invoke the multiplier if its result will be discarded anyway. However, this is simply an optimization, and doesn't hugely affect the overall plan. On every iteration, the value of `e` is right-shifted by one bit. When `e` is zero, the loop terminates.

4.2 Interleaved Modular Multiplier

The interleaved modular has the advantage of not requiring long multiplies, and works with only left shifts, addition, subtraction, and comparison. Unfortunately, a step of the algorithm requires comparing the entire length of the data in the worst case. Additionally, there are 3 possible add/subtract steps at every step of the algorithm. Therefore, the propagation delay of each

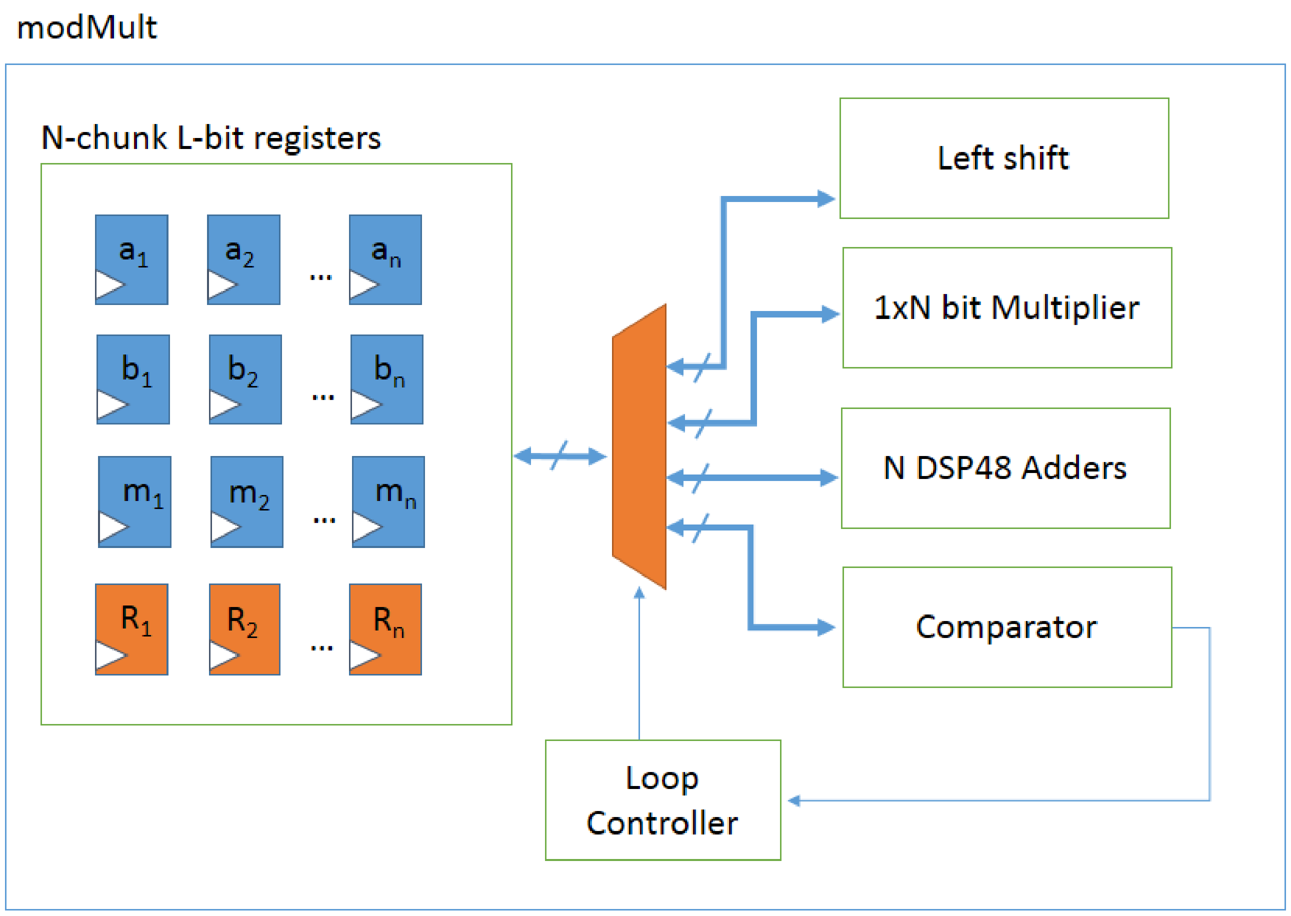

Figure 3: Interleaved modular multiplication

step of the algorithm may be prohibitive without pipelining. We will first investigate the naive, unpipelined approach and see if it offers acceptable (100s of millisecond) performance. If it does not meet this target, then we will explicitly break the data into chunks to increase the clock speed.

An overview of the module is pictured in Figure 3.

4.3 Naive Modular Multiplier

An alternative to interleaved modular multiplication is the Naive approach. The naive modular multiplier does not use any of the specialized algorithms specifically tuned for hardware implementations. After implementing the algorithm in C we have deemed it unfit for implementation of the FPGA and have inhibited further research on this approach.

An overview of this module is pictured in Figure 4.

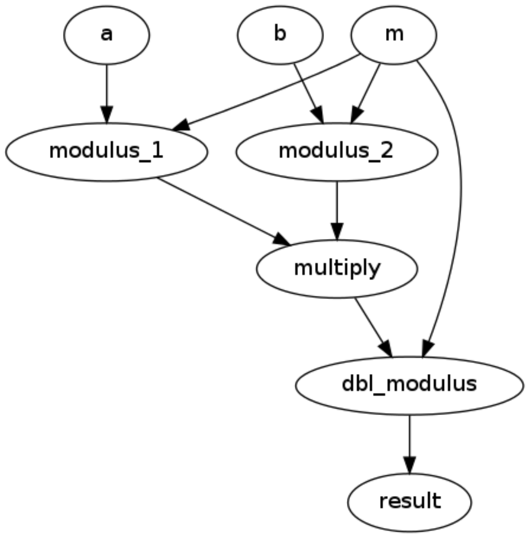

Figure 4: Naive modular multiplication

5 Implementation Status and Planned Exploration

Since the main focus of our project is divided between the two important modules: modular exponentiator and modulus multiplier, we decided to pursue implementing these modules first. At the beginning of this task, we had not decided upon integer representation; therefore, we were required to build multiple interfaces to test different types of data.

5.1 Implemented Modules

The modules we implemented are: `SceMiLayer.bsv, RSAPipeline.bsv, RSA.bsv, RSAPipelinetypes.bsv` `ModMultIlvd.bsv, ModExpt.bsv, Memory.bsv`

SceMiLayer.bsv We developed two SceMiLayer modules to test different types of designs: one for importing vmh files, and another for importing libgcrypt for simulating our c code.

RSAPipeline.bsv This pipeline is based off the audio pipeline in the previous labs. It's interfaces with SceMiLayer to retrieve input data and push output data to the test bench.

RSAPipelinetypes.bsv This is the general header file where we define all constant values. This will make the overall product modular and easy for others to change core elements of design such as number of bits per chunk.

RSA.bsv This is a dedicated alternative driver for the RSA module that performs cosimulation with libgcrypt

ModMultIlvd.bsv The interleaved modulus multiplier based on the Montgomery Algorithm. This function computes $a * b \bmod m$.

ModExpt.bsv The modulus exponentiator described in section ?. This module created two modulus multiplier to computer $b^e \bmod m$.

Memory.bsv This memory module implements BRAM functions: initialize, put, and get.

The system is presently working correctly for 1024 bit RSA operations in simulation. The design performs software co-simulation using libgcrypt: the robust software library performs the same operations as the hardware RSA module, and compares the results at intermediate steps. It takes approximately 30 seconds to simulate one RSA operation, and it is possible to perform randomized, long-term verification of the target module.

5.2 Integer Representation Explorations

1. The first interface was a simple `Int#(1024)` representation. We created a simple adder in order to synthesize our design.

2. The second type of interface is most similar to our C implementation where integers are stored as 64 - 16 bit chunks.

3. The third type of interface uses BRAM to store chunks of the integer throughout the implementation. (This is currently incomplete)

5.2.1 Difficulties Encountered

We created a simple adder in order to synthesize our design. Since we had doubts about the success of this representation this was a vital step before continuing our design. Our concerns were well-founded for the simple `Int#(1024)` representation: the simple addition of two `Int#(1024)` were unable to synthesize.

5.3 Planned Exploration

5.3.1 Trade-Offs: Cycle Time Estimation

There are two major loops that run in the algorithm: the exponentiation loop, and the modular multiplication loop. Each loop performs a bitwise operation on the data, so it needs to perform a cycle for every bit of data. For a 1024 bit block size, both the outer (exponentiation) and inner (multiplication) loops will require 1024 cycles. Therefore, at the minimum, the algorithm requires N^2 cycles to perform one operation. However, since we were unable to synthesize a 1024 bit wide adder, we will have to operate on the data in chunks. This necessitates adding additional nested (pipelined) loops inside the algorithm.

For a N-bit block size with M chunks, we would need to perform an additional $\frac{2N}{M}$ cycles of addition and $\frac{N}{M}$ cycles of comparison within every multiplication step in the worst case. Therefore, the worst case cycle count is given by:

$$cycles_{worst} = \frac{3N^3}{M} \tag{4}$$

For a 32-bit chunk size (N = 1024, M = 64), we would require 100 million cycles to complete an operation on the block. However, this performance is very unlikely and safe to ignore.

The best case clock speed would require only a single comparison (detecting a difference in the first chunk and aborting the full length) and no additional additions, which would defined by

$$cycles_{best} = N \cdot (N + 1) \tag{5}$$

For a 32-bit chunk size, this would necessitate a clock speed of 5 MHz to meet the performance of the Raspberry Pi.

The average case is similar in performance to the best case. In an empirical benchmark using our cycle-accurate simulation, we calculated the following performance for 16-bit chunk size:

$$cycles_{empirical} = 1.6 \cdot N^2 \tag{6}$$

Using this estimate, we will require a clock speed of 6 MHz to meet the Raspberry Pi performance, and a clock speed of 60 MHz to meet the performance of a modern dual core processor. To improve performance further, it is possible to optimize the processing requirements on the required comparison step in the interleaved modular multiply. It may be possible to explore faster comparison architectures by saving intermediate results from previous cycles.

5.4 Design Exploration

One of the major trade offs in our design is cycle period and number of cycles per encryption/decryption. During our preliminary synthesis exploration, we found that we cannot run the operation unpipelined on the FPGA: the maximum frequency we can support mapped to the device is 33 MHz.

The current 33 MHz critical path is due to the 1024 bit ripple-carry adders. If the adder were broken into 4 chunks (for example) then it might take up to 13 or 14 (correspondingly shorter) clock cycles to complete one iteration of the modular multiplier. We have therefore added a pipelined adder to the design, which takes multiple cycles to compute the sum of an addition, and returns the result through a FIFO when finished. This does not add complexity to the modular multiplier, at the cost of FFs required for the input and output FIFOs. The design is presently being synthesized.

This suggests that a modular multiplier might be able to perform multiple computations in parallel. It would be quite a bit of work to arrange, but we might be able to have several computations cycling through the circular pipeline at once. Each would have to be assigned a nonce in order to properly reorder results as they come out, but there's no reason in principle that a single modular multiplier couldn't perform all the computations for several independent modular exponentiators in parallel. Naturally though, this goes somewhat beyond the original scope of the project, and as we are all busy we hesitate to commit to taking on ever more complex tasks.

6 Synthesis Progress Report

We have implemented and verified our design works in simulation on the FPGA from beginning to end! As expected, our design still has not achieved timing to test our program on the FPGA. Here are some relevant snippits from our synthesis report:

```
  Slice Logic Utilization:
  Number of Slice Registers:              44,905 out of  69,120   64%
  Number of Slice LUTs:                   39,631 out of  69,120   57%
    Number used as Memory:                   206 out of  17,920    1%
      Number used as Dual Port RAM:           62

Slice Logic Distribution:
  Number of occupied Slices:              14,501 out of  17,280   83%

  Number of BlockRAM/FIFO:                    30 out of     148   20%
    Number using BlockRAM only:               30
    Total primitives used:
      Number of 36k BlockRAM used:             12
      Number of 18k BlockRAM used:             32
    Total Memory used (KB):                 1,008 out of   5,328   18%

Timing errors: 10188  Score: 59762757 (Setup/Max: 59442565, Hold: 320192)

Constraints cover 81111242 paths, 0 nets, and 250838 connections

Timing Summary:
---------------
Speed Grade: -1

   Minimum period: 29.797ns (Maximum Frequency: 33.561MHz)
   Minimum input arrival time before clock: 3.923ns
   Maximum output required time after clock: 6.527ns
   Maximum combinational path delay: 1.170ns
```

The critical path is through the 1024-bit adder:

```
Timing constraint: Default period analysis for Clock 'scemi_clk_port_clkgen/current_clk1'
  Clock period: 29.797ns (frequency: 33.561MHz)
  Total number of paths / destination ports: 74971937 / 69624
-------------------------------------------------------------------------
Delay:               29.797ns (Levels of Logic = 1036)
  Source:            scemi_dut_dut_dutIfc_m_dut/rsa_modexpt_modmult0_inputFIFO/data0_reg_1
  Destination:       scemi_dut_dut_dutIfc_m_dut/rsa_modexpt_modmult0_p_val_1031 (FF)
  Source Clock:      scemi_clk_port_clkgen/current_clk1 rising
  Destination Clock: scemi_clk_port_clkgen/current_clk1 rising

  Data Path: scemi_dut_dut_dutIfc_m_dut/rsa_modexpt_modmult0_inputFIFO/data0_reg_1032 to s
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
```

```
FD:C->Q            2   0.471   0.581  data0_reg_1032 (data0_reg_1032)
end scope: 'rsa_modexpt_modmult0_inputFIFO'
LUT2:I0->O         1   0.094   0.000  Madd_MUX_rsa_modexpt_modmult0_p_val_write_1_
MUXCY:S->O         1   0.372   0.000  Madd_MUX_rsa_modexpt_modmult0_p_val_write_1_
MUXCY:CI->O        1   0.026   0.000  Madd_MUX_rsa_modexpt_modmult0_p_val_write_1_
MUXCY:CI->O        1   0.026   0.000  Madd_MUX_rsa_modexpt_modmult0_p_val_write_1_
MUXCY:CI->O        1   0.026   0.000  Madd_MUX_rsa_modexpt_modmult0_p_val_write_1_
MUXCY:CI->O        1   0.026   0.000  Madd_MUX_rsa_modexpt_modmult0_p_val_write_1_
```

We have been following the synthesis reports to target our new designs in order to achieve timing. Our first area of improvement was modifying the modular multiplication algorithm for accessing the *ith* bit of an input operand. Instead, we only access the least significant bit and shift the operand on every cycle. This improved the overall performance, however, in observing the synthesis reports we have realized we need to break up the 1024 addition. Our next goal is to implement a chunk-wise addition and subtraction module to break up the critical path.

7 Verification

To verify the functionality of the RSA module, we initially separately compare the results of the encryption and decryption blocks to the results of a software implementation. The two private keys for encryption and decryption modules are passed by SceMi into the hardware. A SceMi testbench pushes a message to the encryption block, along with an enable signal, message, and public key of the software test-bench. The module generates an encrypted message, and the software test-bench uses its private key to decrypt and verify the correctness of the encrypted message.

For decryption, the process is reversed: the test-bench passes in an encrypted message instead of plain-text, and the decryption module uses the private key of the software test-bench to decrypt the message. The test-bench verifies the plain-text for correctness.

After individual testing of the blocks, we will add support for confirming the signature (authenticity checking) of the transmitted message. The test-bench will hash the plain-text message before encryption, and use the private key as the exponent (as if it was decrypting the hash). The hardware decryption module will use the test-bench's public key as the exponent (as during encryption) to retrieve the hash as calculated by the sender. If the hash of the decrypted message matches the original hash, then the message is genuine. The test-bench will purposefully tamper with the encrypted message to prove the correctness of the signature detection mechanism.

To prove correctness to the instructors, a simple test-bench will feed plain-text into an encryption-decryption block pair (connected via a FIFO).