UFRGS
UNIVERSIDADE FEDERAL
DO RIO GRANDE DO SUL

Trabalho de Diplomação

Resumo

Dezembro de 2004

Título: Implementação de um co-processador RSA
Orientador: André Inácio Reis
Co-orientador: Renato Perez Ribas
Aluno: Alcides Silveira Costa
Matrícula: 0069/99-1

Nota

Este trabalho foi realizado no primeiro ano de existência do projeto BRAFITEC-PAGINER, coordenado pelo Prof. Dr. Cláudio F. R. Geyer e com o objetivo de promover o intercâmbio acadêmico entre alunos de engenharia de computação da UFRGS e estudantes de engenharia do INPG – Institut National Polytechnique de Grenoble (França).

Durante o período de agosto de 2003 a junho de 2004, cursei o último ano do curso de Engenharia de Telecomunicações em Grenoble, o qual é composto por duas etapas: um semestre de disciplinas e um semestre de estágio em uma empresa ou laboratório da universidade.

Na segunda etapa, realizei meu estágio no laboratório de pesquisas TIMA, onde desenvolvi, durante quatro meses, meu projeto de final de estudos. Regressando ao Brasil, propus continuar com meu trabalho, realizando, assim, uma obra um pouco diferenciada. Logo, este trabalho divide-se em duas partes: uma feita no Brasil e escrita em português (este resumo) e outra feita na França e escrita em inglês. Além disso, no final da versão inglesa encontram-se as transparências e o código em VHDL apresentado para a banca avaliadora do meu projeto no Brasil.

Porto Alegre, 05 de janeiro de 2005.

Alcides Silveira Costa.

Introdução à versão brasileira

O presente trabalho tem por finalidade continuar com o projeto de final de estudos desenvolvido na França, INPG – *Départment Télécommunications* através do programa CAPES/BRAFITEC. Realizado no *TIMA Laboratory* sob orientação do Professor Régis Leveugle (PhD em microeletrônica – INP Grenoble) durante o período de fevereiro de 2004 até junho de 2004, foi avaliado por Jean-Louis Roch (PhD em Matemática Aplicada - Université Joseph Fourrier de Grenoble), obtendo nota final 14/20.

O tema proposto era o desenvolvimento em VHDL de um coprocessador criptográfico utilizando o algoritmo RSA [4]. Devido à complexidade do problema em manipular números de alta ordem, muito tempo foi despendido em busca de uma arquitetura capaz de executar o algoritmo em um tempo plausível. Várias arquiteturas foram estudadas e, após profunda análise, optamos por implementar uma arquitetura com seu núcleo baseado em pipeline [26]. Ao final do tempo de projeto, terminamos e validamos por meio de simulações a implementação de um módulo que realizava a multiplicação Montgomery [18].

De volta à UFRGS, venho propor a implementação do algoritmo de cifragem/decifragem RSA, sem o processo de geração de chaves. Isto foi definido pelo fato de sabermos que a geração de chaves pode ser feita via software. Além do mais, o processo de geração de chaves é realizado apenas uma vez, não havendo necessidade real de um hardware dedicado para isto. Logo, este documento relata o esforço no desenvolvimento final do hardware, sem preocupar-se com o processo de geração de chaves.

A estrutura desse trabalho está organizada da seguinte forma: este resumo apresenta, na primeira parte, uma síntese do trabalho realizado na França. Após, adicionou-se os resultados adquiridos na UFRGS. A versão detalhada com conceitos sobre o RSA, algoritmos de implementação estudados, diferentes arquiteturas e resultados anteriores está em inglês e anexada ao final deste documento. Maiores detalhes podem ser encontrados e serão referenciados no decorrer do texto.

Resumo

Síntese da versão inglesa

O surgimento da internet mudou radicalmente a maneira pela qual as pessoas trocam informações. Devido à sua crescente popularidade, aplicações como correio eletrônico, clientes de mensagens instantâneas, comércio eletrônico, transações bancárias e compras on-line estão se tornando parte de nossas vidas. Serviços como SMS e WAP estão crescendo em popularidade também. Entretanto, toda essa informação está sujeita a escuta. Uma pessoa pode interceptar sua informação se o sistema não prover mecanismos de segurança adequados para seus usuários. Tentando evitar problemas como estes, criptosistemas devem ser usados quando uma comunicação segura for necessária.

Criptografia é muito mais que apenas codificar e decodificar mensagens. Quando analisamos o mundo eletrônico, autenticação e identificação também são necessárias. Por exemplo, utilizamos autenticação a cada dia em nossas vidas, assinando documentos, cheques, etc. Entretanto, quando movemos para um mundo onde nossas decisões são tomadas eletronicamente, precisamos dispor de técnicas apropriadas.

Observando esse problema, Ronald Rivest, Adi Shamir, and Leonard Adleman desenvolveram em 1978 o criptosistema RSA (Rivest, Shamir, Adleman) [4]: um sistema de chave pública que permite tanto cifragem quanto assinaturas digitais (autenticação).

Em sistemas de chave pública, cada usuário possui um par de chaves. A chave pública é, obviamente, deixada pública enquanto a chave privada é mantida em segredo. A cifragem é realizada com a chave pública enquanto a decifragem é feita com a chave privada. A assinatura de um documento é realizada com a chave privada enquanto a autenticação é feita com a chave pública. Melhores detalhes sobre o funcionamento de sistemas de chave pública, como o RSA, podem ser encontrados na versão inglesa, seção 2, *Understanding Public-key Cryptosystems*.

O algoritmo de cifragem RSA é simples e está descrito na seção 3, *The RSA Cryptosystem*. Sendo a chave pública definida pelo par de números positivos (e, n) e, similarmente, a chave privada definida pelo par (d, n), temos

$$C \equiv E(M) \equiv M^e \pmod{n}$$

para a cifragem de uma mensagem M e

$$M \equiv D(C) \equiv C^d \pmod{n}$$

para realizar a decifragem de uma mensagem cifrada C.

Apesar de simples, o algoritmo envolve números de altíssima ordem (atualmente, M, e, n e d devem ser de, pelo menos, 1024 bits de tamanho, conforme [8]). Essa ordem de grandeza surge do fato que a segurança do RSA está baseada na dificuldade de fatorar grandes números: as chaves são calculadas matematicamente combinando dois números primos de alta ordem. Mesmo conhecendo-se o produto desses números primos (que faz parte da chave pública divulgada, n), a segurança do algoritmo é garantida pela complexidade de fatorar esse produto e se obter os valores secretos.

Sendo assim, a implementação desse algoritmo em hardware torna-se desafiadora, pois o problema concentra-se em encontrar uma forma de realizar uma exponenciação modular rapidamente.

Para um melhor entendimento do algoritmo, robusteza, tamanho das chaves e um simples exemplo mostrando sua funcionalidade, refira-se à versão inglesa, seção 3, *The RSA Cryptosystem*.

Vários algoritmos foram estudados para resolver esse problema [7, 14-18, 20, 24]. Dentre eles, foi escolhido o método de exponenciação de Montgomery [7] (seção 4.1.2), utilizando o método de multiplicação de Montgomery [18]. Mesmo assim, três arquiteturas diferentes também foram analisadas, todas sugerindo maneiras diferentes de executar o RSA. Estas eram: arquiteturas em pipeline [16], arquiteturas baseadas em CRT [15] e arquiteturas baseadas em RNS [17]. Após análise, foram tiradas as seguintes conclusões.

Arquiteturas baseadas em CRT

Apresentam falha de segurança, deixando margem para o atacante. Como os fatores *p* e *q*, necessários para a geração das chaves não são destruídos (são mantido dentro do chip), o atacante pode decifrá-los através de um método de ataque por hardware [19]. Entretanto, essas arquiteturas são as que apresentam a solução mais rápida atualmente, pois dividem o cálculo da execução do RSA em duas unidades de processamento com tamanho de dados reduzidos pela metade.

Arquiteturas baseadas em RNS

Convertendo números binários para um sistema de números diferente (RNS), uma arquitetura altamente paralelizada pode ser implementada. Entretanto, estas não são adequadas para tamanhos de chave pequena, pois os passos de conversão de binário para RNS e vice-versa consomem muito tempo de processamento. Além do mais, sua complexidade de implementação é a maior dentre as três arquiteturas propostas. Apresenta resultados satisfatórios quando o tamanho de chave é maior que 2048 bits.

Arquiteturas baseadas em Pipeline

Apesar de não apresentarem a mesma taxa de cifragem que as duas arquiteturas propostas anteriormente, não apresentam problemas de segurança em sua estrutura e não são tão complexas. Além do mais, sua taxa de cifragem de dados é mais que suficiente para muitas aplicações.

Optamos por implementar a arquitetura em pipeline, devido a sua simplicidade e eficiência. Maiores detalhes sobre os algoritmos e as arquiteturas estudadas podem ser encontrados na seção 4 da versão inglesa, *Design Methods*.

Depois de escolhida a arquitetura, partimos para a especificação do sistema, a qual pode ser vista na seção 5. Abaixo, temos a estrutura de blocos do sistema.

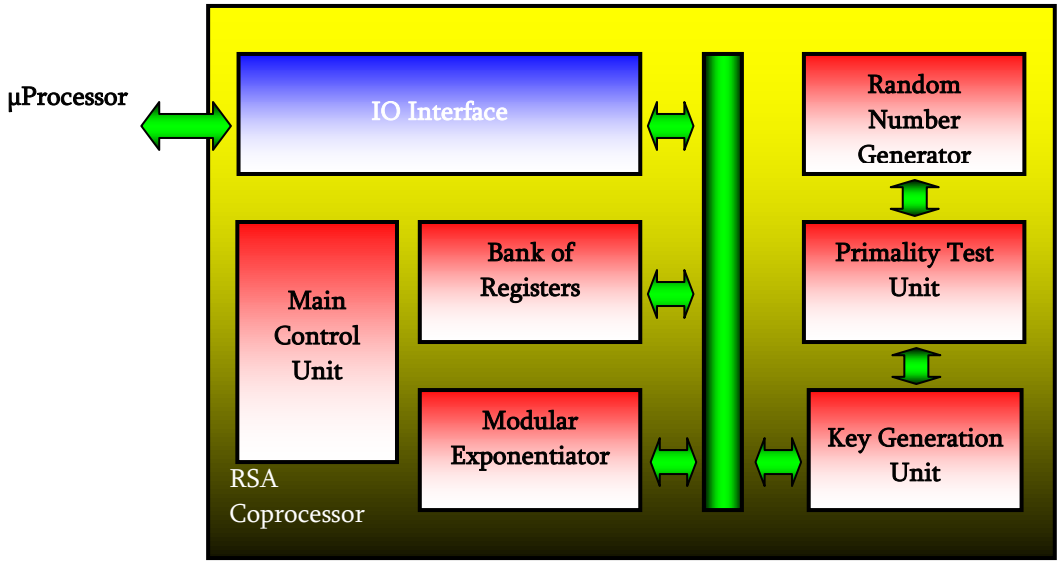


Figura 1 –Diagrama de Blocos

Cabe salientar que a idéia inicial era a de codificar todo o algoritmo RSA em VHDL. Entretanto, esse trabalho requer muito mais tempo de projeto. Perceba que vários blocos precisam ser implementados, cada um com características diferentes. Por exemplo, um bloco de geração de números aleatórios exige um tratamento totalmente diferente do bloco de testes de números primos. Logo, percebendo a inviabilidade de implementarmos todo o sistema em tempo hábil, decidimos concentrar nosso foco na unidade de exponenciação modular.

No final de quatro meses de trabalho, conseguimos terminar e validar por simulações um submódulo da unidade de exponenciação modular – a multiplicação de Montgomery – utilizando uma estrutura

em pipeline. Algoritmos, softwares utilizados e desenvolvidos para a validação do módulo podem ser analisados na seção 6 do documento em anexo.

Ao término do tempo de projeto, obtivemos um módulo totalmente parametrizável, podendo realizar a multiplicação de Montgomery com qualquer tamanho de operando, pois este seria quebrado em palavras definidas pelo usuário. Além disso, o trabalho foi escrito de maneira a conter ótimas referências, dando oportunidade para aqueles que querem conhecer a área de criptografia uma ótima introdução ao assunto.

Continuação

Trabalho realizado no Brasil

Chegando ao Brasil, foi proposta a continuação do trabalho com algumas restrições. Não seria implementada a parte de geração de chaves RSA no trabalho: apenas a continuação do algoritmo de exponenciação modular. Logo, tínhamos como meta implementar todos os blocos referentes a cifragem/decifragem.

Adaptação ao novo ambiente, reescrita de código e revalidação

A primeira dificuldade encontrada foi o novo ambiente de desenvolvimento utilizado. Todo o projeto tinha sido desenvolvido no Modelsim e sintetizado utilizando o Leonardo Spectrum da Mentor Graphics. Agora, estávamos utilizando o Quartus II Web edition da Altera. Houve necessidade de reescrita de código em alguns trechos devido a incompatibilidades encontradas no momento da síntese. Além do mais, *test benches* desenvolvidos em VHDL não eram mais necessários, desde que o ambiente da Altera utiliza *waveforms* para obter resultados de simulações. Logo, um certo tempo inicial foi despendido no aprendizado da nova ferramenta.

Passada esta etapa inicial, revalidamos o Módulo de multiplicação de Montgomery no Quartus II e extraímos os seus resultados. Conferimos com os resultados já adquiridos com o ModelSim (seção 6.1.1.1, *IP Implementation*) e todos fecharam. Entretanto, o trabalho original não descreve a arquitetura interna dos blocos PE (*processing element*), apenas mostra uma arquitetura básica do módulo de multiplicação modular (figura 4-3, versão inglês). Logo, detalharemos um pouco mais a arquitetura do coprocessador neste trabalho, começando pela a organização interna de um PE.

Processing Element (PE)

A figura 2 representa a implementação das linhas 3 a 14 do algoritmo da seção 6.1.1. Note que s_ff mantém o resultado da primeira soma (linha 3), o qual é usado para decidir se m_i (módulo) será somado ao resultado nas outras iterações ou não.

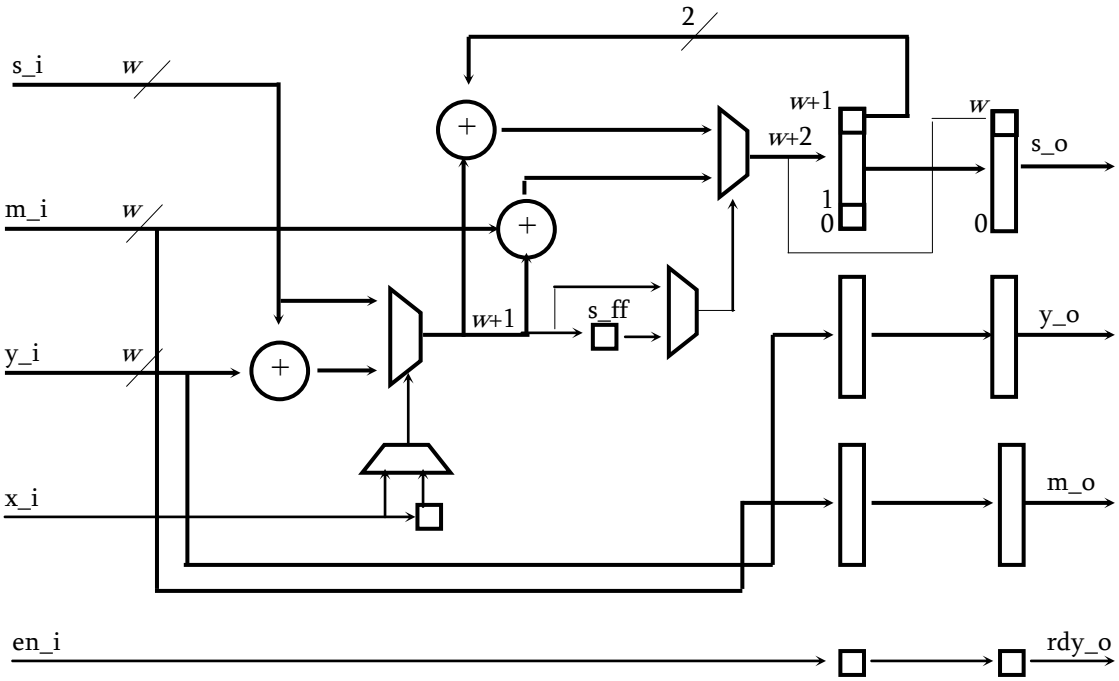


Figura 2 – Processing Element

Além disso, há necessidade de um contador para controle do for loop nas linhas 6 e 11, pois este é controlado pelo sinal de habilitação (en_i), que permanece ativo enquanto os dados de entrada forem válidos. As operações de deslocamento (linhas 8 a 13) são realizadas no registrador mais próximo da saída s_o.

Entretanto, o algoritmo de multiplicação Montgomery ainda não está pronto, pois este deve utilizar vários PE para ser construído, conforme explicado a seguir.

Montgomery Multiplication Unit (MM)

Figura abaixo mostra a maneira como foi implementado o multiplicador de Montgomery.

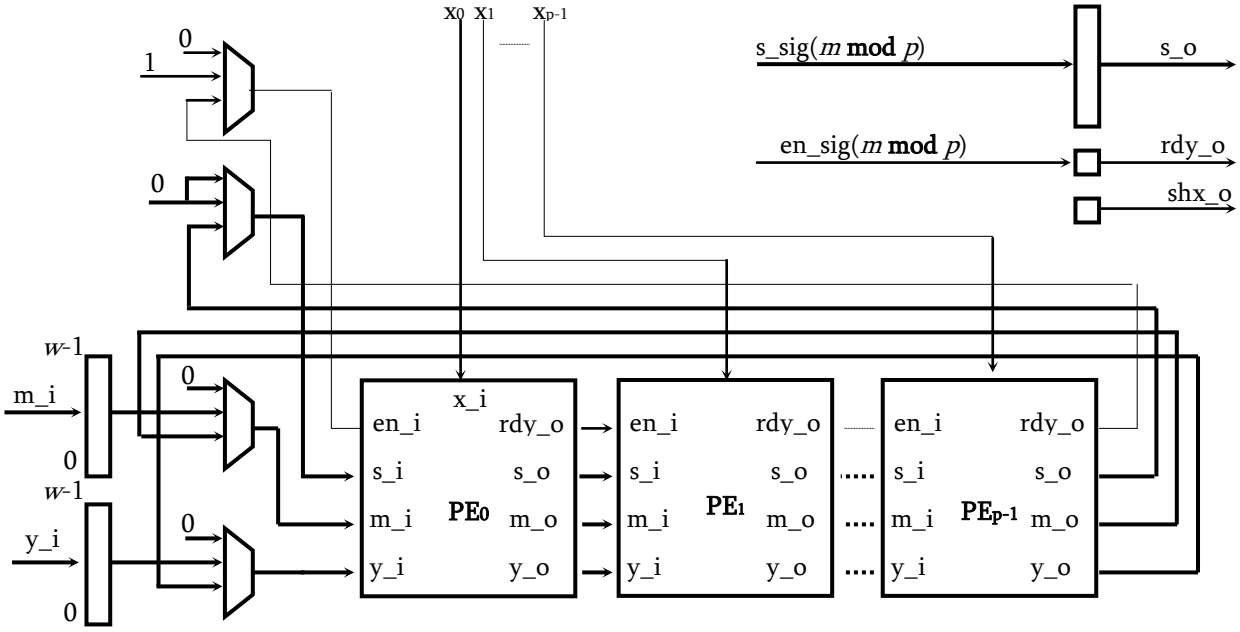


Figura 3 – Montgomery Pipeline

Salientamos que o pipeline mostrado na figura acima é realimentado e que sua saída é totalmente dependentes dos parâmetros *p* (número de unidades paralelas) e *m* (tamanho do módulo em bits). Logo, existe um controle especial para determinar quando o resultado do pipeline deve ser copiado para os registradores de saída. Outro sinal importante é o *shx_o*. Este indica quando o registrador x (figura 4-3, versão inglês) deve ser deslocado.

Escolha do algoritmo de exponenciação modular

Finalmente chegamos ao momento de implementar o RSA propriamente dito. Vários algoritmos foram estudados (seção 4, *Design Methods*). Entretanto, o que melhor se adaptou às nossas necessidades foi o algoritmo apresentado logo abaixo, estudado no Brasil e extraído de [16].

Exponenciação de Montgomery

Entradas: M, e, n, k

Saída: $M^e \bmod n$

1. $\overline{M} := \text{MonPro}(M, k)$

2. $\overline{x} := \text{MonPro}(1, k)$

3 **for** $i := k-1$ **down to** 0 **do**

4 $\bar{x} := \text{MonPro}(\bar{x}, \bar{x})$

5. **if** $e_i = 1$ **then** $\bar{x} := \text{MonPro}(\overline{M}, \bar{x})$

6 $x := \text{MonPro}(\bar{x}, 1)$

7. **return** x

O parâmetro k é uma constante e deve ser pré-calculado, assim como as chaves. Ele vale $2^{2m} \bmod n$, onde m equivale ao tamanho do módulo n em bits. Perceba, também, que o algoritmo usa somente a unidade de multiplicação de Montgomery, não havendo necessidade de uma unidade especial para o cálculo do algoritmo de Euclides, como proposto no algoritmo da seção 4.1.2.

O Coprocessador RSA

O coprocessador RSA desenvolvido também possui uma interface parametrizável onde os dados de entrada são quebrados em palavras. Dessa forma, podemos carregar os dados para dentro do coprocessador com uma palavra de tamanho *l*. Este tamanho independe do tamanho das palavras internas ao coprocessador (barramento interno), ou seja, os cálculos internos são realizados com um tamanho de palavra *w*. A figura abaixo ilustra a arquitetura interna do coprocessador RSA.

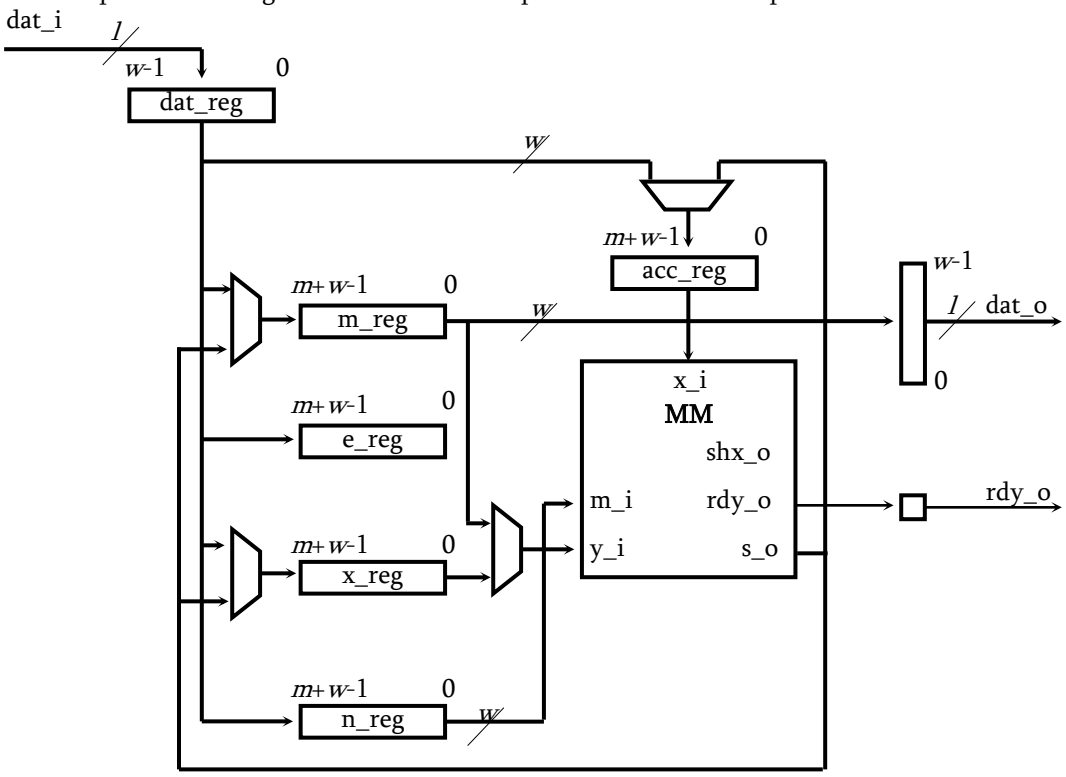


Figura 4 – Coprocessador RSA

Os registradores *dat_reg*, *m_reg*, *x_reg* e *n_reg* possuem *n*+w bits de tamanho. Este requisito é necessário para o funcionamento do algoritmo de multiplicação (capítulo 6). Os registradores *m_reg* e *x_reg* são rotacionadores de *w* bits e *acc_reg* é um registrador de deslocamento de *p* bits controlado pelo sinal *shx_o* (MM). Além disso, *acc_reg* também serve de acumulador para resultados intermediários.

Note que essas características permitem, por exemplo, sintetizar um coprocessador com uma quantidade mínima de pinos. Este é o caso de muitos Smart Cards que são utilizados em transações bancárias, onde o número de pinos não passa de oito.

Problemas com o algoritmo MWR2MM

O algoritmo proposto por [26] e apresentado no capítulo 6 apresenta algumas falhas. Na implementação do coprocessador RSA, foram descobertos alguns erros. Por exemplo, suponha o seguinte problema:

Sendo n=13, M=6 e=5, calcule $C = M^e \bmod n$.

Substituindo os valores, temos $C = 2$. Entretanto, se encontrarmos $C = 15$, este resultado não está totalmente errado, já que $C = 15 \bmod 13 = 2$. Ou seja, a maneira como foi implementado o algoritmo MWR2MM, retorna, em alguns casos, um resultado fora do intervalo [0, n). Logo, um bloco de correção deve ser adicionado na figura 4. Sua arquitetura é mostrada na figura abaixo.

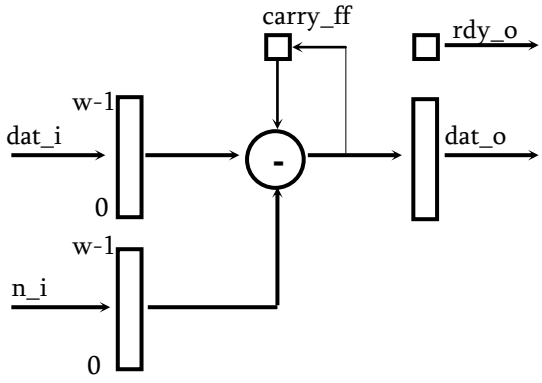


Figura 5 – Subtrator em pipeline

Note que a correção é uma subtração entre o resultado fora do intervalo [0, n) e n. Para isso, os dados de entrada são calculados, palavra a palavra, da mesma maneira que o cálculo da multiplicação Montgomery. A nova arquitetura da figura 4, então, é mostrada na figura 6.

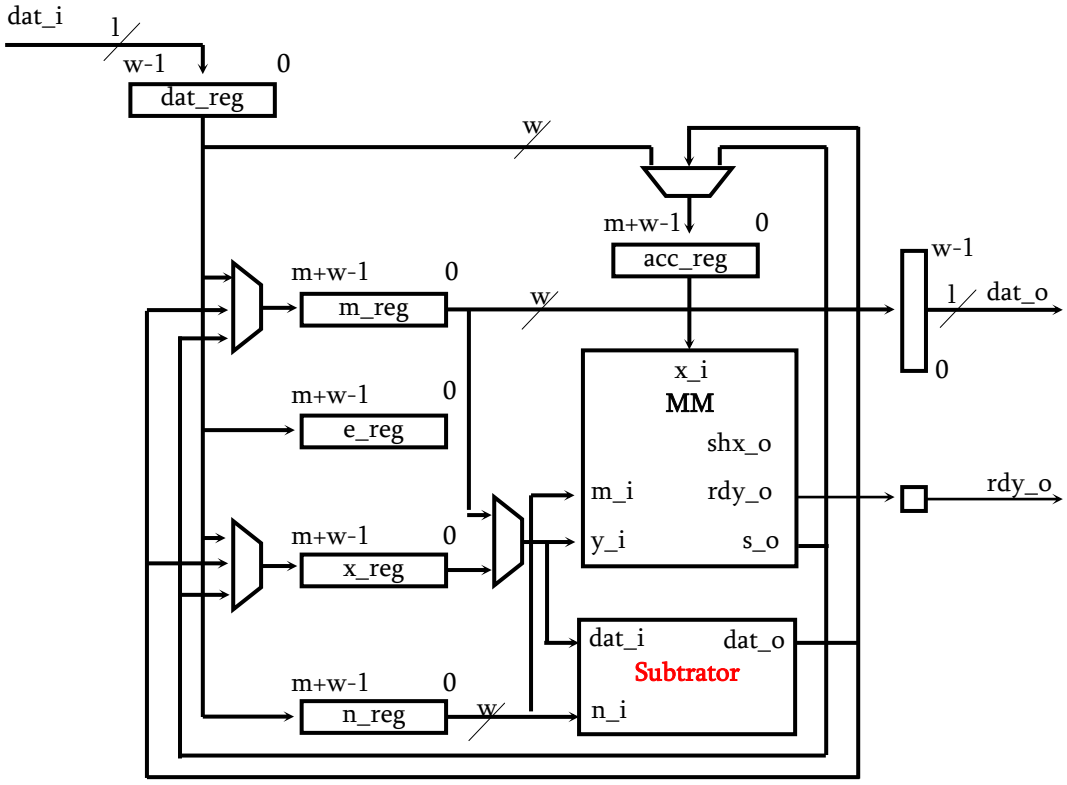


Figura 6 – Coprocessador RSA com correções

Resultados da Simulação

Sendo m o número de bits dos operandos, w o tamanho de uma palavra interna e l o tamanho do barramento de dados de entrada, analisamos os seguintes casos:

Caso I: $m = 8, \quad w = 4, \quad l = 2$ e $r = 2^m = 256$

Para os dados de entrada, temos:

$$M = 200, \quad d = 199, \quad e = 55, \quad n = 221 \text{ e } k = r^2 \bmod n = 256^2 \bmod 221 = 120$$

E, em binário:

M = 11 00 10 00

e = 00 11 01 11 $\quad\quad$ d = 11 00 01 11

n = 11 01 11 01 $\quad\quad$ k = 01 11 10 00

Os dados devem ser inseridos da palavra menos significativa à mais significativa, necessariamente. A ordem dos operandos deve ser: M/C, e/d, n e k. A figura abaixo mostra M, e, n, e k sendo carregados, respectivamente, no coprocessador (carregados aos pares de bits). Note que um sinal de reset foi inserido para inicializar o sistema.

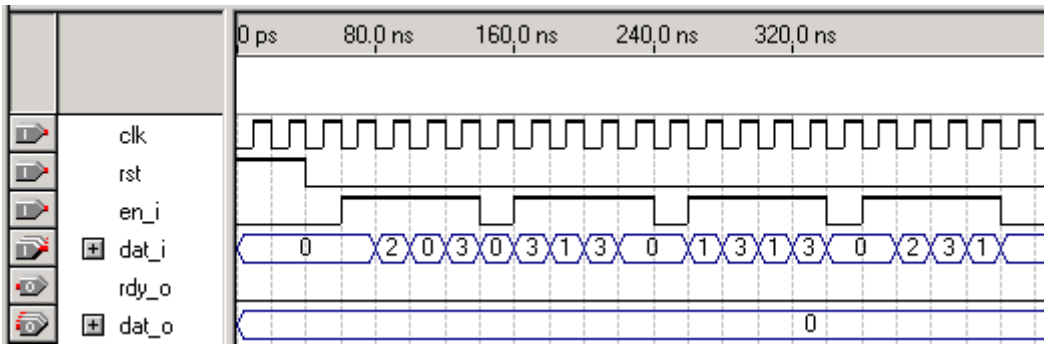
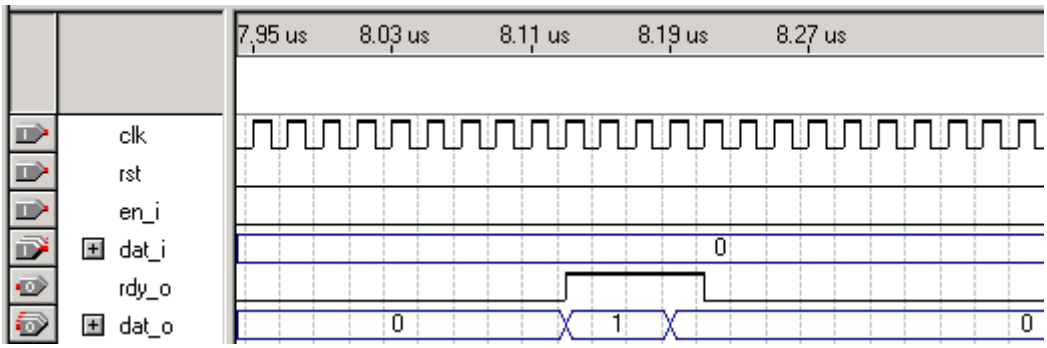


Figura 7 – Simulação (entradas - M, e, n e k)



Transcorrido certo tempo, obtivemos o seguinte resultado:

Figura 8 – Simulação (resultado - C)

Ou seja, $C = 200^{55} \bmod 221 = 00010101_b = 21$. Carregando C, d, n, e k novamente nas entradas do coprocessador, devemos obter M, a mensagem cifrada. As figuras a seguir ilustram esse processo.

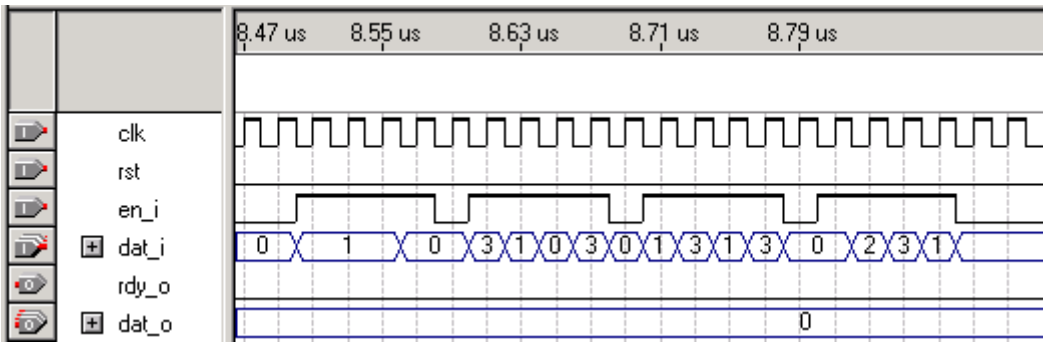


Figura 9 – Simulação (entradas – *C, d, n* e *k*)

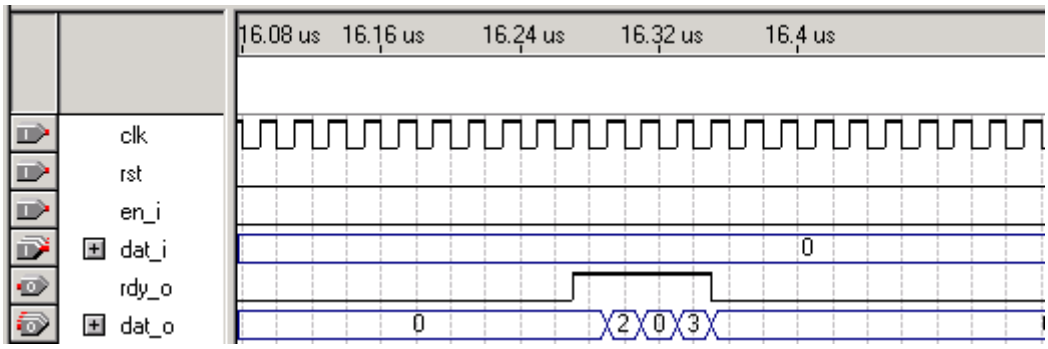


Figura 10 – Simulação (resultado – *M*)

$M = 21^{199} \bmod 221 = 11001000_b = 200$. Deciframos a mensagem.

Caso II: $m = 16, \quad w = 4, \quad l = 2$ e $r = 2^m = 65536$

Entrada: $M = 18001, \quad d = 31247, \quad e = 503, \quad n = 41989$ e $k = r^2 \bmod n = 65536^2 \bmod 41989 = 38453$

Seguindo os mesmos passos da sistemática de testes do caso I, temos:

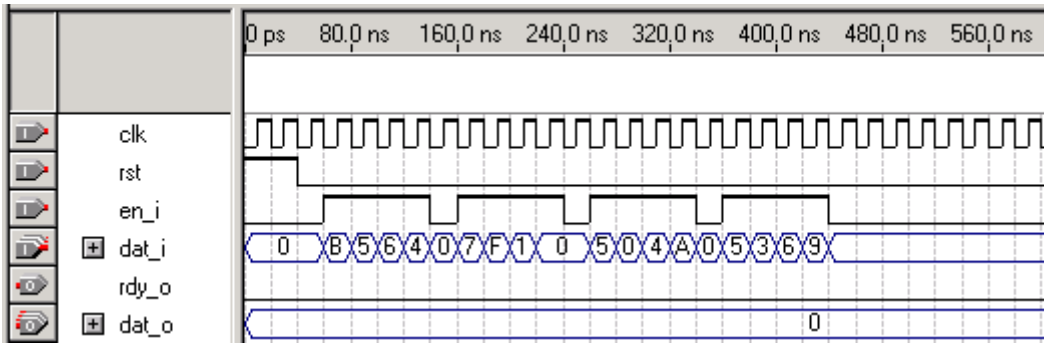


Figura 11 – Simulação (entradas - *M, e, n* e *k*)

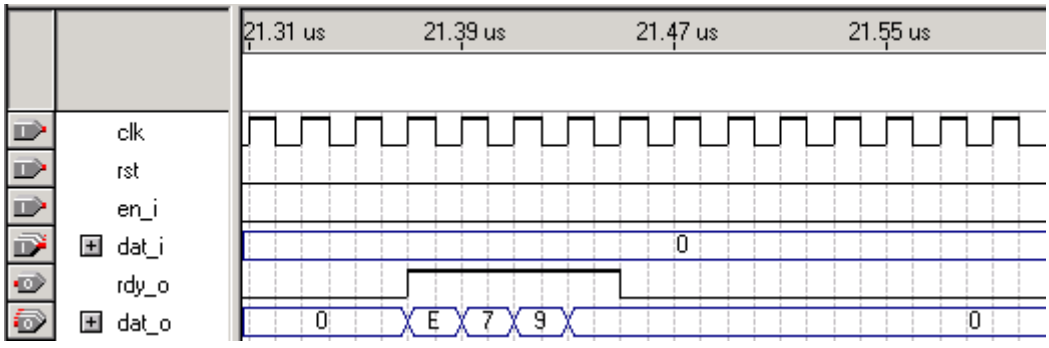


Figura 12 – Simulação (resultado - *C*)

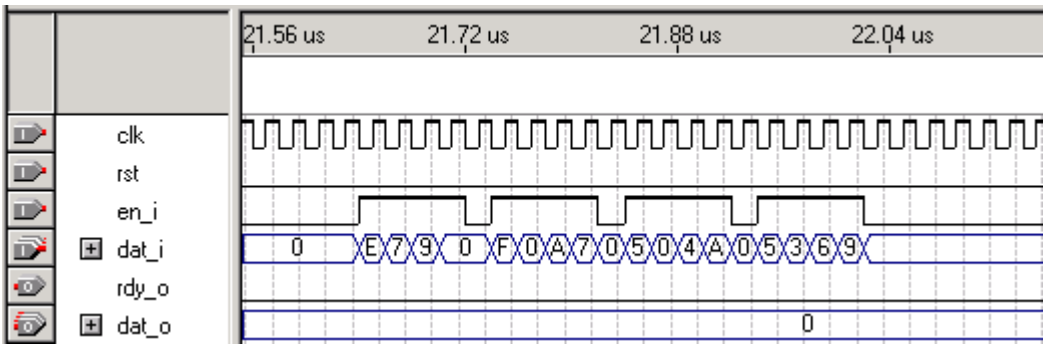


Figura 13 – Simulação (entradas – *C, d, n* e *k*)

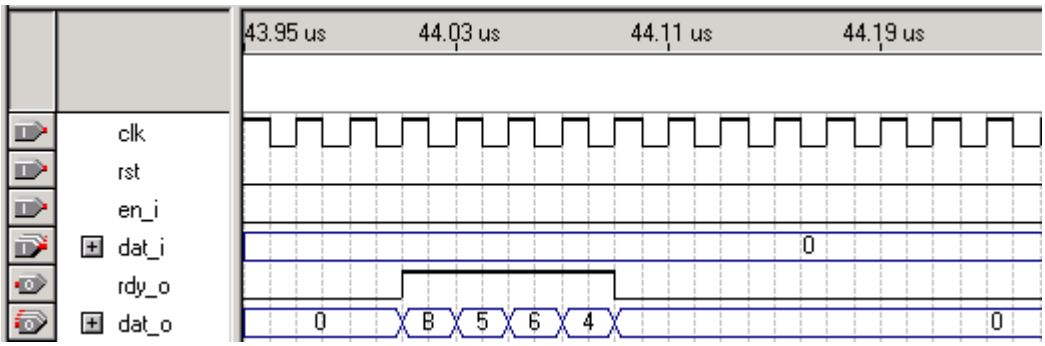


Figura 14 – Simulação (resultado – *M*)

Novamente, os resultados confirmam a funcionalidade do coprocessador desenvolvido.

Resultados da Síntese

Sendo $m = 1024$, $w = 32$ e $l = 8$, obtemos os seguintes resultados utilizando o sintetizador XST da Xilinx e codificação one-hot:

Total de Registradores: 373

91	Flip-flops
1	Registrador de 10 bits
256	Registradores de 32 bits
17	Registradores de 34 bits
2	Registradores de 6 bits
6	Registradores de 8 bits

Total de Multiplexadores: 165

1	Multiplexador de 1 bit de 1056-para-1
164	Multiplexadores 2-para-1

Total de Somadores/Subtratores: 57

1	subtrator de 10 bits
17	somadores de 32 bits
2	subtratores de 33 bits
34	somadores de 24 bits
2	somadores de 6 bits
1	somador de 8 bits

Total de Comparadores: 2

2	Comparadores de 9 bits (menor igual)

Total de pinos de IO: 20

O dispositivo selecionado foi um Virtex Pro II (2vp100ff1696-6). Abaixo segue um relatório sobre o total de recursos utilizados:

Number of Slices	: 12829	out of 44096	(29%)
Number of Slice Flip Flops	: 8977	out of 88192	(10%)
Number of 4 input LUTs	: 22562	out of 88192	(25%)
Number of bonded IOBs	: 19	out of 1164	(1%)
Number of GCLKs	: 1	out of 16	(6%)

Freqüência máxima de operação : 99.636MHz

Caminho Crítico:

mp_block/current_state_FFd4 (FF) -> mp_block/MP0/cs_reg1_33 (FF)

Tempo: 6.863ns lógica e 3.174ns roteamento (68.4% lógica e 31.6% roteamento)

Sabendo que a equação do número de ciclos para a cifragem de m bits é dada por:

$$N_{mc} = \frac{m}{l} + (m+2)\cdot e\,\frac{m}{p} \quad \text{e} \qquad \text{(melhor caso)}$$

$$N_{pc} = \frac{m}{l} + 2\cdot(m+2)\cdot e\left(\frac{m}{p} + \frac{m}{w}\right), \qquad \text{(pior caso)}$$

podemos estimar a taxa média de bits cifrados por segundo, o qual pode ser aproximada por:

$$T_{cifragem} = \frac{2\cdot freq}{N_{pc} + N_{mc}} \cong 24m\,bits/s\,.$$

Como m=1024 nos nossos testes, obtivemos uma taxa média de cifragem de 24Kbits/s.

Interface do Coprocessador RSA

Para o funcionamento do IP, as seguintes medidas devem ser tomadas':

Durante processo de síntese

- A largura do barramento de dados de entrada deve ser menor ou igual à do barramento de dados internos $w \geq l$.

Durante o funcionamento

- Mensagem a ser cifrada deve ser menor que o módulo n;
- Módulo n deve permanecer entre o intervalo $\left(2^{m-1}, 2^{m}\right)$;
- Parâmetro k deve ser pré-calculado;

Abaixo segue uma tabela com os pinos de entrada e saída do coprocessador recém desenvolvido:

Nome do Pino	Tamanho	Direção	Ativo	Descrição
clk	1	entrada	borda de subida	Relógio do Sistema
en_i	1	entrada	1 lógico	Habilita dados de entrada
dat_i	L	entrada	-	Dados de entrada (mensagem)
rdy_o	1	saída	1 lógico	Indica dados válidos na saída
dat_o	L	saída	-	Dados de saída (mensagem cifrada)

Tabela 1 – descrição dos pinos do coprocessador RSA

Conclusão da versão brasileira

O presente trabalho mostrou diversos passos considerados no desenvolvimento de um coprocessador RSA. Do aprendizado de um novo assunto até o resultado final, muitas decisões foram tomadas.

Apesar de existirem soluções mais rápidas, esta implementação garante a segurança dos dados cifrados, pois não mantém internamente os fatores primos intermediários para acelerar o seu cálculo (CRT). Apresentando apenas uma unidade de cálculo, a arquitetura desenvolvida tem a grande vantagem de consumir menos recursos da arquitetura alvo, fato não verdadeiro em arquiteturas RNS.

Muitas dificuldades foram encontradas no decorrer do caminho: um assunto novo, uma arquitetura desafiadora, um algoritmo problemático e ferramentas de desenvolvimento não disponíveis para o desenvolvimento. Entretanto, chegamos ao final do tempo de projeto com um IP funcional e reutilizável para qualquer tamanho de chave, pelo menos enquanto o RSA existir e for considerado seguro.

Mesmo assim, ainda podemos realizar algumas melhorias no coprocessador. Por exemplo, atualmente o IP não suporta que a largura do barramento de dados externo seja maior ou igual do que a largura do barramento de dados interno (l >= w). Leituras após a validação do projeto [16], constataram que uma melhoria no algoritmo pode levar a taxas de 50 Kb/s, não importando se for o melhor ou o pior caso. Esta melhoria aconteceria com a duplicação dos recursos utilizados do dispositivo alvo. Além disso, o coprocessador não aceita mensagens maiores que n e menores que r, uma precaução ainda não tomada.

E, finalmente, apesar de existirem soluções mais rápidas, a velocidade de cifragem atingida neste trabalho é mais que suficiente para muitas aplicações. Por exemplo, alguém utilizando uma Smart Card para autorizar o débito em sua conta bancária, com certeza, não se importaria em esperar cinco centésimos de segundo para assinar uma mensagem digitalmente. Em outro exemplo, podemos citar o envio de informações telefônicas no início de uma chamada por celular (código do aparelho e número do telefone do assinante), onde este tempo gasto não faria a menor diferença para o usuário (e o protegeria da clonagem de seu aparelho).

DÉPARTEMENT
TÉLÉCOMMUNICATIONS

ENSIMAG
ENSERG

Projet de fin d'étude

Rapport final

Juin 2004

Sujet: Implémentation d'un co-processeur RSA

Nom et prénom du responsable: Leveugle, Régis

Nom et prénom du tuteur de l'INPG: Jean-Louis, Roch

Nom, prénom et option de l'étudiant: Costa, Alcides, Architecture des Equipements

Résumé

Ce travail décrit la conception d'un co-processeur RSA. Les fondements des cryptosystèmes à clef publique sont présentés, en montrant leur importance et leurs applications. L'algorithme RSA est ensuite présenté et expliqué. Trois architectures différentes utilisées aujourd'hui pour l'implémentation matérielle de cet algorithme sont ensuite étudiées et analysées: CRT, RNS et l'architecture basée sur un pipeline. Des algorithmes arithmétiques existants sont également détaillés. Après l'analyse des avantages et des inconvénients de chaque solution, nous avons décidé d'implémenter une architecture pipeline qui calcule l'algorithme de multiplication de Montgomery.

Le circuit correspondant a été décrit en VHDL et simulé avec ModelSim. Les résultats obtenus ont montré que l'architecture développée est entièrement fonctionnelle. Elle peut manipuler des nombres de taille quelconque simplement en choisissant la taille des données du pipeline. Finalement, nous pouvons réaliser un taux d'encryptage/décryptage d'environ 25 Kbits/s avec une horloge de 80,9 Mhz et des entrées de 1024 bits.

Abstract

This work describes the design of a scalable RSA coprocessor. By introducing the basics of public-key cryptosystems we provide a good background in cryptography, showing its importance and applications. The RSA algorithm is presented and explained. Three different architectures used today are studied and analyzed: CRT, RNS and Pipelined-based architectures. Existing arithmetic algorithms are also covered. After realizing advantages and disadvantages of each solution we decided to implement a pipelined-based architecture which computes the Montgomery Multiplication algorithm.

By doing this, we coded this architecture in VHDL and simulated it with ModelSim. The results obtained proved that the developed architecture is fully functional. It can manipulate any number of bits simply by selecting the pipeline word size. Finally, with an 80.9 MHz clock and 1024-bit inputs, we can achieve an encryption/decryption rate of about 25 Kb/s.

Contents

Introduction

The advent of the Internet has radically changed the way people exchange information. Due to its growing popularity, applications like electronic mail, instant messengers, electronic commerce, electronic banking, and online shopping are becoming part of our lives. Cell phones services like SMS and WAP are growing in their popularity as well. However, all this information is vulnerable to eavesdropping. A third party group may tap your information if the system doesn't provide adequate security for users. To avoid this problem, cryptographic algorithms are used when secure communication is needed.

Since ancient times, cryptography has been used mainly for military purposes. Its first use dates back to 1900 BC, when a scribe in Egypt used a derivation of the standard hieroglyphic of the day to communicate [12]. However, the Roman emperor Julius Caesar is considered to be one of the first people to have employed encryption for securing messages [11]. Caesar decided his standard algorithm would shift each letter of the Roman alphabet a predetermined number of places. He informed all of his generals of his decision. By following this example and shifting the contemporary English alphabet over 3 places, a message like ZEBRA would be ciphered as CHEUD. Despite of being an unbreakable cryptography system at that time, this system can be currently broken in a few seconds.

Today's cryptography is concerned with more than just encrypting and decrypting messages. When we move to an electronic world, authentication and identification schemas are needed. Whenever we log on a remote computer to access our bank account, or we shop online using our credit card, we are subjected to eavesdropping and possibly forgery.

By observing this problem, Ronald Rivest, Adi Shamir, and Leonard Adleman developed in 1978 the RSA (**R**ivest, **S**hamir, **A**dleman) cryptosystem [4]: a public-key cryptosystem that offers both encryption and digital signatures (authentication).

Public key cryptosystems are not the only ones used in applications. Secret key cryptosystems and elliptic curve cryptosystems are largely exploited as well. However, a detailed explanation about these last cryptosystems is outside the scope of this work. Only cryptography tools associated to the RSA cryptosystem will be discussed.

It's not the purpose of this work to write lines of codes or draw pages of logic schematics. The purpose of this work is to present the design of a RSA coprocessor and give a theoretical base to the reader in the cryptography field. This allows interested parties to understand the project and proceed with the work, if necessary. For this reason, the first three sections are reserved to give a good background to the reader. The other sections will be reserved for the project itself, detailed below.

Section 1 provides a basic introduction to the field of cryptography. It gives a brief explanation about its main concepts, some applications where it can be found, and its importance to the electronic world.

In section 2 public-key cryptosystems are covered. Also, two systems related to public-key cryptosystems are introduced: secret-key cryptosystems and hash functions.

Section 3 approaches the RSA cryptosystem, explaining its algorithm in a very practical way.

Section 4 introduces some modular exponentiation algorithms used to compute RSA and how they can be applied in digital systems. It presents and compares the three most used architectures found in the literature today: CRT, RNS and Pipelined based architectures.

Section 5 specifies our proposed coprocessor organization in a block diagram form and its related pinout interface.

Finally, in section 6 we describe the techniques used to implement some blocks introduced in the section 5 and its results.

1. A Glance at Cryptography

This section provides the reader a basic introduction to the field of cryptography. It gives a brief explanation about its main concepts, some applications where it can be found, and its importance to the electronic world. This chapter was extracted from [1].

1.1. Concepts

Cryptography today might be summed up as the study of techniques and applications that depend on the existence of difficult problems. We should also say that it is the science of using mathematics to secure information and create a high degree of trust. *Cryptanalysis* is the study of how to compromise (defeat) cryptographic mechanisms, and *cryptology* (from the Greek *kryptos logos*, meaning "hidden word") is the discipline of cryptography and cryptanalysis combined. To most people, cryptography is concerned with keeping communications private. Indeed, the protection of sensitive communications has been the emphasis of cryptography throughout much of its history. However, this is only one part of today's cryptography.

Encryption is the transformation of data into a form that is as close as possible to an understandable form of reading without the appropriate knowledge (a key; see below). Its purpose is to ensure privacy by keeping information hidden from anyone who it is not allowed, even those who have access to the encrypted data. *Decryption* is the reverse of encryption; it is the transformation of encrypted data back into an intelligible form.

Encryption and decryption generally require the use of some secret information, referred to as a *key*. For some encryption mechanisms, the same key is used for both encryption and decryption; for other mechanisms, the key used for encryption and decryption is different. Data to be encrypted is called *Plaintext. Ciphertext* is the encrypted data.

Today's cryptography is more than encryption and decryption. *Authentication* is as fundamental as privacy. We use authentication throughout our everyday lives (when we sign our name to some document, for instance). As we move to a world where our decisions and agreements are communicated electronically, we need to have electronic techniques for providing authentication.

Cryptography provides mechanisms for such procedures. A *digital signature* binds a document to the possessor of a particular key, while a *digital timestamp* binds a document to its creation at a particular time. These cryptographic mechanisms can be used to control access to a shared disk drive, a high security installation, or a pay-per-view TV channel.

The field of cryptography encompasses other uses as well. With just a few basic cryptographic tools, it is possible to build elaborate schemes and protocols that allow us to pay using electronic money, to prove we know certain information without revealing the information itself, and to share a secret quantity in such a way that a subset of the shares can reconstruct the secret.

While modern cryptography is growing increasingly diverse, cryptography is fundamentally based on problems that are difficult to solve. A problem may be difficult because its solution requires some secret knowledge, such as decrypting an encrypted message or signing some digital document without the key. The problem may also be hard because it is intrinsically difficult to complete, such as finding a message that produces a given hash value (explained in section 3.2.2).

An encryption system together with a corresponding decryption system is a *cryptosystem* [2]. Two classes of cryptosystems are *secret-key* and *public-key cryptosystems*. In secret-key cryptosystems, also referred to as *symmetric cryptography*, the same key is used for both encryption and decryption. The most popular secret-key cryptosystem in use today is the *Data Encryption Standard*, also known as *DES*.

In public-key cryptosystems, each user has a *public key* and a *private key*. The public key is made public while the private key remains secret. Encryption is performed with the public key while decryption is done with the private key. The *RSA public-key cryptosystem* is the most popular form

of public-key cryptography. RSA stands for Rivest, Shamir, and Adleman, the inventors of the RSA cryptosystem.

The *Digital Signature Algorithm* (*DSA*) is also a popular public-key technique, though it can only be used only for signatures, not encryption. *Elliptic curve cryptosystems* (*ECCs*) are cryptosystems based on mathematical objects known as elliptic curves. Elliptic curve cryptography has been gaining in popularity recently. Lastly, the *Diffie-Hellman key agreement protocol* is a popular public-key technique for establishing secret keys over an insecure channel.

1.2. Cryptography Applications

Privacy is perhaps the most obvious application of cryptography. Cryptography can be used to implement privacy simply by encrypting the information intended to remain private. In order for someone to read this private data, one must first decrypt it.

There are a large number of applications in which it is being currently in use. For example, simple cryptography systems can be used for *secure communication*, *identification*, *authentication*, and *secret sharing*. However, more complicated applications include systems for *electronic commerce*, *certification*, *secure electronic mail*, *key recovery*, and *secure computer access*. A better explanation of these applications can be read in the next lines below.

1.2.1. Secure Communication

Secure communication is the most straightforward use of cryptography. Two people may communicate securely by encrypting the messages sent between them. This can be done in such a way that a third party eavesdropping may never be able to decipher the messages. While secure communication has existed for centuries, the *key management*[1] problem has prevented it from becoming commonplace. Thanks to the development of public-key cryptography, it is possible to create a large-scale network of people who can communicate securely with one another even if they had never communicated before.

1.2.2. Identification and Authentication

Identification and authentication are two widely used applications of cryptography. Identification is the process of verifying someone's or something's identity. For example, when you withdraw some money from a bank, a teller asks you to see your identification (a drive's license, for instance). By doing this, he or she verifies the identity of the owner's account (your identity). This same process can be done electronically by using cryptography. Every automatic teller machine (ATM) card is associated with a "secret" personal identification number (PIN), which binds the owner to the card and thus to the account. When the card is inserted into the ATM, the machine prompts the cardholder for the PIN. If the correct PIN is entered, the machine identifies that person as the rightful owner and it grants him access. Another important application of cryptography is authentication. Authentication is similar to identification, in that both allow an entity access to resources (such as an Internet account), but authentication is broader because it does not necessarily involve identifying a person or entity. Authentication merely determines whether that person or entity is authorized for whatever is in question.

1.2.3. Secret Sharing

Another application of cryptography, called secret sharing, allows the trust of a secret to be distributed among a group of people. For example, in a (k, n)-threshold scheme, information about a secret is distributed in such a way that k-out of n people ($k \leq n$) have enough information to determine the secret, but any set of k - 1 people do not. In any secret sharing scheme, there are designated sets of people whose cumulative information suffices to determine the secret. In some

[1] The various processes that deals with the creation, distribution, authentication, and storage of keys.

implementations of secret sharing schemes, each participant receives the secret after it has been generated. In other implementations, the actual secret is never made visible to the participants, although the purpose for which they sought the secret (for example, access to a building or permission to execute a process) is allowed.

1.2.4. Electronic Commerce

Over the past few years there has been a growing amount of business conducted over the Internet. This form of business is called electronic commerce or *e*-commerce. *E*-commerce is comprised of online banking, online brokerage accounts, and Internet shopping, to name a few of the many applications. One can book plane tickets, make hotel reservations, rent a car, transfer money from one account to another, buy compact disks (CDs), clothes, books and so on all while sitting in front of a computer. However, simply entering a credit card number on the Internet leaves one open to fraud. One cryptographic solution to this problem is to encrypt the credit card number (or other private information) when it is entered online, another is to secure the entire session. When a computer encrypts this information and sends it out on the Internet, it is incomprehensible to a third party viewer. The web server ("Internet shopping center") receives the encrypted information, decrypts it, and proceeds with the sale without fear that the credit card number (or other personal information) slipped into wrong hands. As more and more business is conducted over the Internet, the need for protection against fraud, theft, and corruption of vital information increases.

1.2.5. Certification

Another application of cryptography is certification. Certification is a scheme by which trusted agents such as certifying authorities vouch for unknown agents, such as users. The trusted agents issue vouchers called certificates which each have some inherent meaning. Certification technology was developed to make identification and authentication possible on a large scale.

1.2.6. Key Recovery

Key recovery is a technology that allows a key to be revealed under certain circumstances without the owner of the key revealing it. This is useful for two main reasons: first of all, if a user loses or accidentally deletes his or her key, key recovery could prevent a disaster. Secondly, if a law enforcement agency wishes to eavesdrop on a suspected criminal without the suspect's knowledge (akin to a wiretapping) the agency must be able to recover the key. Key recovery techniques are in use in some instances. However, the use of key recovery as a law enforcement technique is somewhat controversial.

1.2.7. Remote Access

Secure remote access is another important application of cryptography. The basic system of passwords certainly gives a level of security for secure access, but it may not be enough in some cases. For instance, passwords can be eavesdropped, forgotten, stolen, or guessed. Many products supply cryptographic methods for remote access with a higher degree of security.

1.2.8. Other Applications

Cryptography is not confined to the world of computers. Cryptography is also used in cellular (mobile) phones as a means of authentication, that is, it can be used to verify that a particular phone has the right to bill to a particular phone number. This prevents people from stealing ("cloning") cellular phone numbers and access codes. Another application is to protect phone calls from eavesdropping using voice encryption.

1.3. Importance of Cryptography

Cryptography allows people to transfer to an account the confidence found in the physical world to the electronic world, thus allowing people to do business electronically without worries of deceit and

deception. Every day hundreds of thousands of people interact electronically, whether it is through *e*-mail, *e*-commerce, ATM machines, or cellular phones. The constant increase of information transmitted electronically has lead to an increased reliance on cryptography.

1.3.1. Cryptography on the Internet

The Internet, comprised of millions of interconnected computers, allows nearly instantaneous communication and transfer of information, around the world. People use *e*-mail to correspond with one another. The World Wide Web is used for online business, data distribution, marketing, research, learning, and a myriad of other activities.

Cryptography makes secure web sites and electronic safe transmissions possible. For a web site to be secure all of the data transmitted between the computers where the data is kept and where it is received must be encrypted. This allows people to do online banking, online trading, and make online purchases with their credit cards, without worrying that any of their account information is being compromised. Cryptography is very important to the continued growth of the Internet and electronic commerce.

E-commerce is increasing at a very rapid rate. By the turn of the century, commercial transactions on the Internet are expected to total hundreds of billions of dollars a year. This level of activity could not be supported without cryptographic security. It has been said that one is safer using a credit card over the Internet than within a store or restaurant. It requires much more work to capture credit card numbers over computer networks than it does to simply walk by a table in a restaurant and take a credit card receipt. These levels of security, though not yet widely used, give the means to strengthen the foundation with which *e*-commerce can grow.

People use *e*-mail to conduct personal and business matters on a daily basis. *E*-mail has no physical form and may exist electronically in more than one place at a time. This poses a potential problem as it increases the opportunity for an eavesdropper to catch the transmission. Encryption protects *e*-mail by rendering it very difficult to read by any unintended party. Digital signatures can also be used to authenticate the origin and the content of an *e*-mail message.

1.3.2. Authentication

In some cases cryptography allows you to have more confidence in your electronic transactions than you do in real life transactions. For example, signing documents in real life still leaves one vulnerable to the following scenario. After signing your will, agreeing to what is put forth in the document, someone can change that document and your signature is still attached. In the electronic world this type of falsification is much more difficult because digital signatures are built using the contents of the document being signed.

1.3.3. Access Control

Cryptography is also used to regulate access to satellite and cable TV. Cable TV is set up so people can watch only the channels they pay for. Since there is a direct line from the cable company to each individual subscriber's home, the Cable Company will only send those channels that are paid for. Many companies offer pay-per-view channels to their subscribers. Pay-per-view cable allows cable subscribers to "rent" a movie directly through the cable box. What the cable box does is decode the incoming movie, but not until the movie has been "rented." If a person wants to watch a pay-per-view movie, he/she calls the cable company and requests it. In return, the Cable Company sends out a signal to the subscriber's cable box, which unscrambles (decrypts) the requested movie.

Satellite TV works slightly differently since the satellite TV companies do not have a direct connection to each individual subscriber's home. This means that anyone with a satellite dish can pick up the signals. To alleviate the problem of people getting free TV, they use cryptography. The trick is to allow only those who have paid for their service to unscramble the transmission; this is done with receivers ("unscramblers"). Each subscriber is given a receiver; the satellite transmits

signals that can only be unscrambled by such a receiver. Pay-per-view works in essentially the same way as it does for regular cable TV.

As seen, cryptography is widely used. Not only is it used over the Internet, but also it is used in phones, televisions, and a variety of other common household items. Without cryptography, hackers could get into our *e*-mail, listen in on our phone conversations, tap into our cable companies and acquire free cable service, or break into our bank accounts.

2. Understanding Public-key Cryptosystems

This section gives more details about public-key cryptosystems, providing more information about the concepts involved in cryptography. Two systems related to public-key cryptosystems are introduced; secret-key cryptosystems and hash functions. These systems are introduced because they can be found in many applications using public-key cryptosystems (section 3.4). Also, a comparison between secret-key cryptosystems and public-key cryptosystem is written.

This chapter was extracted from [1].

2.1. Public-key Cryptosystems

In traditional cryptography, the sender and receiver of a message know and use the same secret key. The sender uses the secret key to encrypt the message, and the receiver uses the same secret key to decrypt the message. This method is known as secret key or symmetric cryptography. The main challenge is getting the sender and receiver to agree on the secret key without anyone else finding out. If they are in separate physical locations, they must trust a courier, a phone system, or some other transmission medium to prevent the disclosure of the secret key. Anyone who overhears or intercepts the key in transit can later read, modify, and forge all messages encrypted or authenticated using that key. The generation, transmission and storage of keys is called *key management*. All cryptosystems must deal with key management issues. Because all keys in a secret-key cryptosystem must remain secret, secret-key cryptography often has difficulty providing secure key management, especially in open systems with a large number of users.

In order to solve the key management problem, Whitfield Diffie and Martin Hellman introduced the concept of public-key cryptosystems in 1976 [3]. Public-key cryptosystems have two primary uses, encryption and digital signatures. In their system, each person gets a pair of keys, one called the public key and the other called the private key. The public key is published, while the private key is kept secret. The need for the sender and receiver to share secret information is eliminated: all communications involve only public keys, and no private key is ever transmitted or shared. In this system, it is no longer necessary to trust the security of some means of communications. The only requirement is that public keys be associated with their users in a trusted (authenticated) manner (for instance, in a trusted directory). Anyone can send a confidential message by just using public information, but the message can only be decrypted with a private key, which is in the sole possession of the intended recipient. Furthermore, public-key cryptography can be used not only for privacy (encryption), but also for authentication (digital signatures) and other various techniques.

In a public-key cryptosystem, the private key is always linked mathematically to the public key. Therefore, it is always possible to attack a public-key system by deriving the private key from the public key. Typically, the defense against this is to make the problem of deriving the private key from the public key as difficult as possible. For instance, some public-key cryptosystems are designed such that deriving the private key from the public key requires the attacker to factor a large number, in such a way that it is computationally infeasible to perform the derivation. This is the idea behind the RSA public-key cryptosystem.

2.1.1. Encryption

When Alice wishes to send a secret message to Bob, she looks up Bob's public key in a directory, uses it to encrypt the message and sends it off. Bob then uses his private key to decrypt the message and read it. No one listening in can decrypt the message. Anyone can send an encrypted message to Bob, but only Bob can read it (because only Bob knows Bob's private key).

2.1.2. Digital Signatures

To sign a message, Alice does a computation involving both her private key and the message itself. The output is called a digital signature and is attached to the message. To verify the signature, Bob

does a computation involving the message, the purported signature, and Alice's public key. If the result is correct according to a simple, prescribed mathematical relation, the signature is verified to be genuine; otherwise, the signature is fraudulent, or the message may have been altered.

2.2. Systems related to Public-key Cryptosystems

2.2.1. Secret-key Cryptosystem

Secret-key cryptosystems is sometimes referred to as symmetric cryptography. It is the more traditional form of cryptography, in which a single key can be used to encrypt and decrypt a message. Secret-key cryptography not only deals with encryption, but it also deals with authentication. One such technique is called *message authentication codes*.

The main problem with secret-key cryptosystems is getting the sender and receiver to agree on the secret key without anyone else finding out. This requires a method by which the two parties can communicate without fear of eavesdropping.

Public-key cryptography has come to overcome this deficiency of secret-key cryptosystems by establishing secure means for sending keys in a trusted way. Public-key cryptography is not meant to replace secret-key cryptography, but rather to supplement it, to make it more secure (see key agreement protocol, 3.4.2). Secret-key cryptography remains extremely important and is the subject of much ongoing study and research.

2.2.2. Hash functions

A *hash function* H is a transformation that takes an input m and returns a fixed-size string, which is called the hash value h (that is, h = H(m)). Hash functions with just this property have a variety of general computational uses, but when employed in cryptography, the hash functions are usually chosen to have some additional properties.

The basic requirements for a cryptographic hash function are as follows.

- The input can be of any length.
- The output has a fixed length.
- H(x) is relatively easy to compute for any given x.
- H(x) is one-way.
- H(x) is collision-free.

A hash function H is said to be *one-way* if it is hard to invert, where "hard to invert" means that given a hash value h, it is computationally infeasible to find some input x such that H(x) = h. If, given a message x, it is computationally infeasible to find a message y not equal to x such that H(x) = H(y), then H is said to be a *weakly collision-free* hash function. A *strongly collision-free* hash function H is one for which it is computationally infeasible to find any two messages x and y such that H(x) = H(y).

The hash value represents concisely the longer message or document from which it was computed; this value is called the *message digest*. One can think of a message digest as a "digital fingerprint" of the larger document. Examples of well known hash functions are MD2 and MD5 and SHA.

Perhaps the main role of a cryptographic hash function is in the provision of message integrity checks and digital signatures. Since hash functions are generally faster than encryption or digital signature algorithms, it is typical to compute the digital signature or integrity check to some document by applying cryptographic processing to the document's hash value, which is small compared to the document itself. Additionally, a digest can be made public without revealing the contents of the document from which it is derived.

Note that sometimes information is not supposed to be accessed by anyone, and in these cases, the information may be stored in such a way that reversing the process is virtually impossible. For

instance, on a typical multi-user system, no one is supposed to know the list of passwords of everyone on the system. Often hash values of passwords are stored instead of the passwords themselves. This allows the users of the system to be confident their private information is actually kept private while still enabling an entered password to be verified (by computing its hash and comparing that result against a stored hash value). This scheme is applied in the widely used operating system UNIX [2]

2.3. Advantages and disadvantages of public-key cryptosystems

The primary advantage of public-key cryptography is increased security and convenience: private keys never need to be transmitted or revealed to anyone. In a secret-key system, by contrast, the secret keys must be transmitted (either manually or through a communication channel) since the same key is used for encryption and decryption. A serious concern is that there may be a chance that an enemy can discover the secret key during transmission.

Another major advantage of public-key systems is that they can provide digital signatures that cannot be repudiated. Authentication via secret-key systems requires the sharing of some secret and sometimes requires trust of a third party as well. As a result, a sender can repudiate a previously authenticated message by claiming the shared secret was somehow compromised by one of the parties sharing the secret. For example, there are secret-key authentication systems involving a central database that keeps copies of the secret keys of all users; an attack on the database would allow widespread forgery. Public-key authentication, on the other hand, prevents this type of repudiation; each user has sole responsibility for protecting his or her private key. This property of public-key authentication is often called non-repudiation.

A disadvantage of using public-key cryptography for encryption is speed. There are many secret-key encryption methods that are significantly faster than any currently available public-key encryption method. Nevertheless, public-key cryptography can be used with secret-key cryptography to get the best of both worlds. For encryption, the best solution is to combine public- and secret-key systems in order to get both the security advantages of public-key systems and the speed advantages of secret-key systems. Such a protocol is called a *digital envelope* (section 3.4.3).

Public-key cryptography may be vulnerable to impersonation, even if users' private keys are not available. A successful attack on a certification authority will allow an adversary to impersonate whomever he or she chooses by using a public-key certificate from the compromised authority to bind a key of the adversary's choice to the name of another user.

In some situations, public-key cryptography is not necessary and secret-key cryptography alone is sufficient. These include environments where secure secret key distribution can take place, for example, by users meeting in private. It also includes environments where a single authority knows and manages all the keys, for example, a closed banking system. Since the authority knows everyone's keys already, there is not much advantage for some to be "public" and others to be "private." Note, however, that such a system may become impractical if the number of users becomes large; there are not necessarily any such limitations in a public-key system.

Public-key cryptography is usually not necessary in a single-user environment. For example, if you want to keep your personal files encrypted, you can do so with any secret key encryption algorithm using, say, your personal password as the secret key. In general, public-key cryptography is best suited for an open multi-user environment.

2.4. Some applications liked to Public-key Cryptosystems

2.4.1. Authentication and Digital Signature

Authentication is any process through which one proves and verifies certain information. Sometimes one may want to verify the origin of a document, the identity of the sender, the time and date a document was sent and/or signed, the identity of a computer or user, and so on. A *digital signature* is a cryptographic means through which many of these may be verified. The digital signature of a

document is a piece of information based on both the document and the signer's private key. It is typically created through the use of a hash function and a private signing function (encrypting with the signer's private key), but there are other methods.

Every day, people sign their names to letters, credit card receipts, and other documents, demonstrating they are in agreement with the contents. That is, they authenticate that they are in fact the sender or originator of the item. This allows others to verify that a particular message did indeed originate from the signer. However, this is not foolproof, since people can 'lift' signatures off one document and place them on another, thereby creating fraudulent documents. Written signatures are also vulnerable to forgery because it is possible to reproduce a signature on other documents as well as to alter documents after they have been signed.

Digital signatures and hand-written signatures both rely on the fact that it is very hard to find two people with the same signature. People use public-key cryptography to compute digital signatures by associating something unique with each person. When public-key cryptography is used to encrypt a message, the sender encrypts the message with the public key of the intended recipient. When public-key cryptography is used to calculate a digital signature, the sender encrypts the "digital fingerprint" of the document with his or her own private key. Anyone with access to the public key of the signer may verify the signature.

Suppose Alice wants to send a signed document or message to Bob. The first step is generally to apply a hash function to the message, creating what is called a message digest. The message digest is usually considerably shorter than the original message. In fact, the job of the hash function is to take a message of arbitrary length and shrink it down to a fixed length. To create a digital signature, one usually signs (encrypts) the message digest as opposed to the message itself. This saves a considerable amount of time, though it does create a slight insecurity (addressed below). Alice sends Bob the encrypted message digest and the message, which she may or may not encrypt. In order for Bob to authenticate the signature he must apply the same hash function as Alice to the message she sent him, decrypt the encrypted message digest using Alice's public key and compare the two. If the two are the same he has successfully authenticated the signature. If the two do not match there are a few possible explanations. Either someone is trying to impersonate Alice, the message itself has been altered since Alice signed it or an error occurred during transmission.

There is a potential problem with this type of digital signature. Alice not only signed the message she intended to but also signed all other messages that happen to hash to the same message digest. When two messages hash to the same message digest it is called a *collision*; the collision-free properties of hash functions are a necessary security requirement for most digital signature schemes. A hash function is secure if it is very time consuming to figure out the original message given its digest. However, there is an attack called the *birthday attack* that relies on the fact that it is easier to find two messages that hash to the same value than to find a message that hashes to a particular value. Its name arises from the fact that for a group of 23 or more people the probability that two or more people share the same birthday is better than 50%.

In addition, someone could pretend to be Alice and sign documents with a key pair he claims is Alice's. To avoid scenarios such as this, there are digital documents called certificates that associate a person with a specific public key.

Digital timestamps may be used in connection with digital signatures to bind a document to a particular time of origin. It is not sufficient to just note the date in the message, since dates on computers can be easily manipulated. It is better that timestamping is done by someone everyone trusts, such as a certifying authority. There have been proposals suggesting the inclusion of some unpredictable information in the message such as the exact closing share price of a number of stocks; this information should prove that the message was created *after* a certain point in time.

2.4.2. Key agreement protocol

A *key agreement protocol*, also called a *key exchange protocol*, is a series of steps used when two or more parties need to agree upon a key to use for a secret-key cryptosystem. These protocols allow people to share keys freely and securely over any insecure medium, without the need for a previously-established shared secret.

Suppose Alice and Bob want to use a secret-key cryptosystem to communicate securely. They first must decide on a shared key. Instead of Bob calling Alice on the phone and discussing what the key will do, which would leave them vulnerable to an eavesdropper; they decide to use a key agreement protocol. By using a key agreement protocol, Alice and Bob may securely exchange a key in an insecure environment. One example of such a protocol is called the Diffie-Hellman key agreement. In many cases, public-key cryptography is used in a key agreement protocol. Another example is the use of digital envelopes for key agreement.

2.4.3. Digital Envelope

When we are using secret-key cryptosystems, users must first agree on a session key, that is, a secret key to be used for the duration of one message or communication session. In completing this task there is a risk the key will be intercepted during transmission. This is part of the key management problem. Public-key cryptosystems offers an attractive solution to this problem within a framework called a digital envelope.

The digital envelope consists of a message encrypted using secret-key cryptography and an encrypted secret key. While digital envelopes usually use public-key cryptography to encrypt the secret key, this is not necessary. If Alice and Bob have an established secret key, they could use this to encrypt the secret key in the digital envelope.

Suppose Alice wants to send a message to Bob using secret-key cryptography for message encryption and public-key cryptography to transfer the message encryption key. Alice chooses a secret key and encrypts the message with it, then encrypts the secret key using Bob's public key. She sends Bob both the encrypted secret key and the encrypted message. When Bob wants to read the message he decrypts the secret key, using his private key, and then decrypts the message, using the secret key. In a multi-addressed communications environment such as e-mail, this can be extended directly and usefully. If Alice's message is intended for both Bob and Carol, the message encryption key can be represented concisely in encrypted forms for Bob and for Carol, along with a single copy of the message's content encrypted under that message encryption key.

Alice and Bob may use this key to encrypt just one message or they may use it for an extended communication. One of the nice features about this technique is they may switch secret keys as frequently as they would like. Switching keys often is beneficial because it is more difficult for an adversary to find a key that is only used for a short period of time.

Not only do digital envelopes help solve the key management problem; they increase performance (relative to using a public-key system for direct encryption of message data) without sacrificing security. The increase in performance is obtained by using a secret-key cryptosystem to encrypt the large and variably sized amount of message data, reserving public-key cryptography for encryption of short-length keys. In general, secret-key cryptosystems are much faster than public-key cryptosystems.

The digital envelope technique is a method of key exchange, but not all key exchange protocols use digital envelopes.

2.4.4. Identification

Identification is a process through which one ascertains the identity of another person or entity. In our daily lives, we identify our family members, friends, and coworkers by their physical properties, such as voice, face or other characteristics. These characteristics, called biometrics, can only be used

on computer networks with special hardware. Entities on a network may also identify one another using cryptographic methods.

An identification scheme allows Alice to identify herself to Bob in such a way that someone listening in cannot pose as Alice later. One example of an identification scheme is a zero-knowledge proof. Zero knowledge proofs allow a person (or a server, web site, etc.) to demonstrate they have certain piece of information without giving it away to the person (or entity) they are convincing. Suppose Alice knows how to solve the Rubik's cube and wants to convince Bob she can without giving away the solution. They could proceed as follows. Alice gives Bob a Rubik's cube which he thoroughly messes up and then gives back to Alice. Alice turns away from Bob, solves the puzzle and hands it back to Bob. This works because Bob saw that Alice solved the puzzle, but he did not see the solution.

This idea may be adapted to an identification scheme if each person involved is given a "puzzle" and its answer. The security of the system relies on the difficulty of solving the puzzles. In the case above, if Alice were the only person who could solve a Rubik's cube, then that could be her puzzle. In this scenario Bob is the verifier and is identifying Alice, the prover.

The idea is to associate with each person something unique; something only that person can reproduce. This in effect takes the place of a face or a voice, which are unique factors allowing people to identify one another in the physical world.

Authentication and identification are different. Identification requires that the verifier check the information presented against all the entities it knows about, while authentication requires that the information be checked for a single, previously identified, entity. In addition, while identification must, by definition, uniquely identify a given entity, authentication does not necessarily require uniqueness. For instance, someone logging into a shared account is not uniquely identified, but by knowing the shared password, they are authenticated as one of the users of the account. Furthermore, identification does not necessarily authenticate the user for a particular purpose.

3. The RSA Cryptosystem

The present section approaches the RSA cryptosystem, explaining its algorithm in a very practical way. It also covers its main characteristics like its speed compared to other secret-key cryptosystems, its strength against attacks, and its security closely related to the key sizes. At the end of this section, we have solved a very simple example using the RSA algorithm. The contents of sections 3.1 to 3.4 were estracted from [1, 4-5, 8-10].

3.1. The RSA algorithm

The RSA cryptosystem is a public-key cryptosystem that offers both encryption and digital signatures (authentication). Ronald Rivest, Adi Shamir, and Leonard Adleman developed the RSA system in 1978; RSA stands for the first letter in each of its inventors' last names.

The RSA algorithm works as described below:

First, compute n, where

$$n = pq \tag{3.1.1}$$

and p and q should be two large prime numbers chosen randomly. The number n is called *modulus*.

Then, pick up an integer d to be a large, random integer which is *relatively prime* to $(p-1)(q-1)$. That is, check that d satisfies:

$$\gcd(d,(p-1)\cdot(q-1)) = 1. \tag{3.1.2}$$

Here, gcd means "greatest common divisor". Finally, compute another number e from p, q, and d to be the *multiplicative inverse* of d modulo $(p\text{-}1).(q\text{-}1)$. Thus we have

$$ed \equiv 1 (\mathrm{mod}(p-1)\cdot(q-1)). \tag{3.1.3}$$

The *encryption key* is thus the pair of positive integers (e, n). Similarly, the *decryption key* is the pair of positive integers (d, n). Each user makes his encryption key public, and keeps the corresponding decryption key private (these integers should properly be subscripted as in n_A, e_A, and d_A, since each user has his own set).

Now, to encrypt a message M using a public encryption key (e, n), proceed as follows:

Represent the message as an integer between 0 and $n - 1$. If necessary, break a long message into a series of blocks, and represent each block as such an integer. The purpose here is not to encrypt the message but only to get it into the numeric form necessary for encryption.

Then, encrypt the message by raising it to the e^{th} power modulo n. That is, the result (the ciphertext C) is the remainder when M^e is divided by n.

To decrypt the ciphertext, raise it to another power d, again modulo n. The encryption and decryption algorithms E and D are thus:

$$C \equiv E(M) \equiv M^e \,(\mathrm{mod}\ n), \tag{3.1.4}$$

for a message M.

$$M \equiv D(C) \equiv C^d \,(\mathrm{mod}\ n), \tag{3.1.5}$$

for a ciphertext C.

Note that encryption does not increase the size of a message. Both the message M and the ciphertext C are integers in the range 0 to n-1.

3.1.1. Encryption

Suppose Alice wants to send a message M to Bob. Alice creates the ciphertext C by exponentiating $C = M^e \bmod n$, where e and n are Bob's public key. She sends C to Bob. To decrypt, Bob also

exponentiates: $M = C^d \bmod n$; the relationship between e and d ensures that Bob correctly recovers M. Since only Bob knows d, only Bob can decrypt this message.

3.1.2. Digital Signature

Suppose Alice wants to send a message M to Bob in such a way that Bob is assured the message is both authentic, has not been tampered with, and from Alice. Alice creates a digital signature S by exponentiating: $S = M^d \bmod n$, where d and n are Alice's private key. She sends M and S to Bob. To verify the signature, Bob exponentiates and checks that the message M is recovered: $M = S^e \bmod n$, where e and n are Alice's public key.

Thus encryption and authentication take place without any sharing of private keys: each person uses only another's public key or their own private key. Anyone can send an encrypted message or verify a signed message, but only someone in possession of the correct private key can decrypt or sign a message.

For interested readers, a formal proof and explanation of the RSA algorithm can be found in [4].

3.2. RSA Speed

An "RSA operation", whether encrypting, decrypting, signing, or verifying is essentially a modular exponentiation. This computation is performed by a series of modular multiplications.

In practical applications, it is common to choose a small public exponent for the public-key. In fact, entire groups of users can use the same public exponent, each with a different modulus (there are some restrictions on the prime factors of the modulus when the public exponent is fixed). This makes encryption faster than decryption and verification faster than signing. With the typical modular exponentiation algorithms used to implement the RSA algorithm, public-key operations take $O(k^2)$ steps, private-key operations take $O(k^3)$ steps, and key generation takes $O(k^4)$ steps, where k is the number of bits in the modulus. "Fast multiplication" techniques, such as methods based on the Fast Fourier Transform (FFT), require asymptotically fewer steps. In practice, however, they are not as common due to their greater software complexity and the fact that they may actually be slower for typical key sizes.

By comparison, DES and other private-key cryptosystems are much faster than the RSA algorithm. DES is generally at least 100 times faster in software and between 1,000 and 10,000 times faster in hardware than RSA, depending on the implementation. Implementations of the RSA algorithm will probably narrow the gap a bit in coming years, due to high demand, but private-key cryptosystems will get faster as well.

3.3. RSA Robustness

There are a few possible interpretations of "breaking" the RSA system. The most damaging would be for an attacker to discover the private key corresponding to a given public key; this would enable the attacker both to read all messages encrypted with the public key and to forge signatures. The obvious way to do this attack is to factor the public modulus, n, into its two prime factors, p and q. From p, q, and e, the public exponent, the attacker can easily get d, the private exponent. The hard part is factoring n; the security of RSA depends on factoring being difficult. In fact, the task of recovering the private key is equivalent to the task of factoring the modulus: you can use d to factor n, as well as use the factorization of n to find d.

It is not necessarily true that a large number is more difficult to factor than a smaller number. For example, the number 10^{1000} is easier to factor than the RSA-155. To keep abreast of the state of art in factoring, RSA Security [13] administers with quarterly cash awards a challenge called RSA Factoring Challenge, where the most important result thus far was the factorization of the RSA-155 (a number with 155 digits). Its factorization was completed in August 1999, after seven months, by a group performing the necessary computations on 300 workstations and PCs. The factorization of this 512-bit number is crucial as 512 is the default key size used for the major part of the *e*-commerce on

Internet. The result indicates that a well-organized group of users using distributed systems might be able to break a 512-bit key in just a couple of days.

As a curiosity, we mention that the RSA-155 factorization is

109417386415705274218097073220403576120037329454492059909138421314763499842889347847179972578912673324976257528997818337970765372440271467435315933543338 97

=

102639592829741105772054196573991675900716567808038066803341933521790711307779

*

106603488380168454820927220360012878679207958575989291522270608237193062808643.

For more information about the RSA Factoring challenge, see [9].

What is true in general is that a number with large prime factors is more difficult to factor than a number with small prime factors (still, the running time of some factoring algorithms depends on the size of the number only and not on the size of its prime factors). This is why the size of the modulus in the RSA algorithm determines how secure an actual use of the RSA cryptosystem is. Namely, an RSA modulus is the product of two large primes; with a larger modulus, the primes become larger and hence an attacker needs more time to factor it. Yet, remember that a number with large prime factors might possess certain properties making it easy to factor. For example, this is the case if the prime factors are very close to each other (see next section).

It has not been proven that factoring must be difficult, and there remains a possibility that a quick and easy factoring method might be discovered, though factoring researchers consider this possibility remote.

Another way to break the RSA cryptosystem is to find a technique to compute e^{th} roots mod n. Since $C = M^e \bmod n$, the e^{th} root of C mod n is the message M. This attack would allow someone to recover encrypted messages and forge signatures even without knowing the private key. This attack is not known to be equivalent to factoring. No general methods are currently known that attempt to break the RSA system in this way. However, in special cases where multiple related messages are encrypted with the same small exponent, it may be possible to recover the messages.

The attacks just mentioned are the only ways to break the RSA cryptosystem in such a way as to be able to recover all messages encrypted under a given key. There are other methods, however, that aim to recover single messages; success would not enable the attacker to recover other messages encrypted with the same key. Some people have also studied whether part of the message can be recovered from an encrypted message.

It should also be noted that hardware improvements alone will not weaken the RSA cryptosystem, as long as appropriate key lengths are used. In fact, hardware improvements should increase the security of the cryptosystem.

Of course, there are also attacks that aim not at the cryptosystem itself but at a given insecure implementation of the system; these do not count as "breaking" the RSA system, because it is not any weakness in the RSA algorithm that is exploited, but rather a weakness in a specific implementation. For example, if someone stores a private key insecurely, an attacker may discover it. One cannot emphasize strongly enough that to be truly secure, the RSA cryptosystem requires a secure implementation; mathematical security measures, such as choosing a long key size, are not enough. In practice, most successful attacks will likely be aimed at insecure implementations and at the key management stages of an RSA system.

3.4. Key sizes

The size of a key in the RSA algorithm typically refers to the size of the modulus n. The two primes, p and q, which compose the modulus, should be of roughly equal length; this makes the modulus

harder to factor than if one of the primes is much smaller than the other. If one chooses to use a 768-bit modulus, the primes should each have length approximately 384 bits. If the two primes are extremely close or their difference is close to any predetermined amount, then there is a potential security risk, but the probability that two randomly chosen primes are so close is negligible.

The best size for a modulus depends on one's security needs. The larger the modulus, the greater the security, but also, the slower the RSA algorithm operations. One should choose a modulus length upon consideration, first, of the value of the protected data and how long it needs to be protected, and, second, of how powerful one's potential threats might be.

As showed in the section 4.3, 512-bit RSA keys may be factored for less than $1,000,000 in cost and eight months of effort. This means that 512-bit keys no longer provide sufficient security for anything more than very short-term security needs.

Currently, it is recommended key sizes of 1024 bits for corporate use and 2048 bits for extremely valuable keys like the root key pair used by a certifying authority.

Several recent standards specify a 1024-bit minimum for corporate use. Less valuable information may well be encrypted using a 768-bit key, as such a key is still beyond the reach of all known key breaking algorithms.

It is typical to ensure that the key of an individual user expires after a certain time, say, two years. This gives an opportunity to change keys regularly and to maintain a given level of security. Upon expiration, the user should generate a new key being sure to ascertain whether any changes in cryptanalytic skills make a move to longer key lengths appropriate. Of course, changing a key does not defend against attacks that attempt to recover messages encrypted with an old key, so key size should always be chosen according to the expected lifetime of the data. The opportunity to change keys allows one to adapt to new key size recommendations. RSA Laboratories [8] publishes recommended key lengths on a regular basis.

Users should keep in mind that the estimated times to break the RSA system are averages only. A large factoring effort, attacking many thousands of moduli, may succeed in factoring at least one in a reasonable time. Although the security of any individual key is still strong, with some factoring methods there is always a small chance the attacker may get lucky and factor some key quickly.

As for the slowdown caused by increasing the key size, doubling the modulus length will, on average, increase the time required for public key operations (encryption and signature verification) by a factor of four, and increase the time taken by private key operations (decryption and signing) by a factor of eight. The reason public key operations are affected less than private key operations is that the public exponent can remain fixed while the modulus is increased, whereas the length of the private exponent increases proportionally. Key generation time would increase by a factor of 16 upon doubling the modulus, but this is a relatively infrequent operation for most users.

It should be noted that the key sizes for the RSA system (and other public-key techniques) are much larger than those for secret-key cryptosystems like DES, but the security of an RSA key cannot be compared to the security of a key in another system purely in terms of length.

3.5. A Simple Example

After providing the reader with the basis of the RSA algorithm, let's try a little example. Initially, we should generate two random prime numbers required by the algorithm. Let $p = 2$ and $q = 5$. Though these numbers are not used in practical applications, they serve as a good example.

From equation 4.1.1, we have,

$$n = pq = 2 \cdot 5 = 10.$$

To compute the private key, we should apply equation 4.1.2, as showed below:

$$\gcd(d, (p-1)\cdot(q-1)) = \gcd(d, (2-1)\cdot(5-1)) = \gcd(d, 4) = 1 \Rightarrow d = 7.$$

As may be seen, we chose $d = 7$. We could have chosen $d = 3$, $d = 5$, $d = 9$ and so on. However, remember that d is an exponent and if we pick a big number we will deal with large computations. Here, our goal is to show an example that can be performed using just paper and a pencil.

Our next step is to find the public key. This can be accomplished by solving equation 4.1.3. Then, we have

$$ed \equiv 1(\mathrm{mod}(p-1)\cdot(q-1)) \Rightarrow 7\cdot e \equiv 1(\mathrm{mod}(2-1)\cdot(5-1)) \Rightarrow 7\cdot e \equiv 1(\mathrm{mod}\,4) \Rightarrow e = 3.$$

Finally, we possess both keys: the private one (7, 10) and the public one (3, 10). For encrypting a message, for instance, M=3, we proceed as described bellow.

By taking equation (4.1.4), we have,

$$C \equiv E(M) \equiv M^e \,(\mathrm{mod}\, n) \Rightarrow C \equiv 3^3 \,(\mathrm{mod}\, 10).$$

Hence, our enciphered message C is

$$C = 7.$$

Now, to decrypt C, we apply,

$$M \equiv D(C) \equiv C^d \,(\mathrm{mod}\, n) \Rightarrow 7^7 \,(\mathrm{mod}\, 10),$$

Then, resulting in

$$M = 3.$$

Note that despite handling small key sizes and short messages we have reached results of the same order of magnitude as 10^6 (7^7 = 823543)! This example has clearly demonstrated that the RSA algorithm needs too much computation for encrypting/decrypting. Also, the public key was made smaller than the private one. This reason is obvious: it makes encryption faster than decryption and verification faster than signing. A better explanation can be found in section 4.2.

We didn't choose p and q by chance. As a matter of fact, the product pq gave us as a result the *modulus* n=10. There is no doubt that modulo10 operations are much easier to solve than any other modulo operation. You should just take the least significant number of an integer to solve its modulo (82354**3** mod 10 = **3**). Of course, we still have modulo 1 and modulo 2 operations, but I couldn't create any practical examples with these operations.

You should also have perceived that p and q are picked randomly. But truly random numbers are difficult to come by software. This poses a challenge for software developers implementing cryptography as computers are logical and deterministics. For this reason, computer-generated random numbers are sometimes called pseudorandom numbers. As an example we can refer to the linear congruence method and the elementary cellular automaton method [5].

4. Design Methods

This section introduces some modular exponentiation algorithms used to compute RSA and how they can be applied in digital systems. Also, it presents and briefly explains the three most used architectures found in the literature nowadays: CRT, RNS and Pipelined based architectures. Finally, at the end of the section a good comparison among these architectures is written, allowing us to choose which type of implementation can bring us the best results.

4.1. Modular Exponentiation Algorithms

There are many arithmetic algorithms for implementing RSA in hardware in the technical literature. Most of then are concerned in finding a fast way of solving the RSA algorithm. By being more specific, these algorithms are centered in solving operations with large size operands, i.e.,

$$C \equiv M^e \pmod{n}$$

where M, e and n have more than 512 bits. As mentioned in section 3.3, 512-bit keys are not secure enough and 1024-bit key sizes must be used to give a certain level of security to the user. That's why at the present moment various researchers are working on discovering a faster way to perform operations with large size operands in hardware/software.

The modular exponentiation algorithm found in many articles is practically the same: the binary method. It seems to be the most adequate. Others algorithms, e.g., m-ary method, factor method, power three method, addition chains and recording binary method can also be found [6]. However, they were not cited in the documentation researched [15-17]. Sections 4.1.1 and 4.1.2 were extracted from [6].

4.1.1. Binary Exponentiation Method

The binary method scans the bits of the exponent either from left to right or from right to left. A squaring is performed at each step, and depending on the scanned bit value, a subsequent multiplication is performed. We explain the left-to-right binary method below. Interested readers can find more information about the right-to-left binary method in [6].

Let k be the number of bits of e and the binary expansion of e be given by

$$e = (e_{k-1}e_{k-2}\cdots e_1 e_0) = \sum_{i=0}^{k-1} e_i 2^i$$

for $e_i \in \{0,1\}$. The binary method for computing $C = M^e \pmod{n}$ is given below (by scanning the power bits from left to right):

The Binary Exponentiation Method LR

Inputs: M, e, n

Output: $C = M^e \pmod{n}$

1. **if** $e_{k-1} = 1$ **then** $C := M$ **else** $C := 1$

2. **for** $i = k-2$ **downto** 0

2a. $C := C \cdot C \pmod{n}$

2b. **If** $e_i = 1$ **then** $C := C \cdot M \pmod{n}$

3. **return** C

Assuming $e > 0$, the total number of multiplications is:

- $(k-1)+(k-1) = 2(k-1)$, (maximum)
- $(k-1)+0 = k-1$ or (minimum)
- $(k-1)+1/2(k-1) = 3/2(k-1)$, (average)

where we assume that $e_{k-1} = 1$.

Steps 2a and 2b can be replaced by any modular multiplication algorithm.

4.1.1.1. Interleaving multiplication and reduction

Let A_i and B_i be the bits of the k-bits positive integers A and B, respectively. The product P can be written as

$$P = A \cdot B = A \cdot \sum_{i=o}^{k-1} B_i 2^i = \sum_{i=o}^{k-1} (A \cdot B_i) 2^i.$$

This formulation yields the shift-add multiplication algorithm. We also reduce the partial product modulo n at each step:

The Interleaving Multiplication and Reduction Method

Inputs: A, B

Output: P

1. $P := 0$

2. **for** $i = 0$ **to** $k-1$

2a. $P := 2P + A \cdot B_{k-1-i}$

2b. $P := P \pmod{n}$

3. **return** P

In line 2b, we have a modular division. The multiplication step is then followed by a division algorithm in order to compute the remainder. However, we are not interested in the quotient; we only need the remainder.

Therefore, the steps of the division algorithm can somewhat be simplified in order to speed up the process. The reduction step can be achieved by making one of the well-known sequential division algorithms. In the following sections, we describe the restoring and the nonrestoring division algorithms for computing the remainder of P when divided by n.

4.1.1.2. Restoring Division Algorithm

Let R_i be the remainder obtained during the i^{th} step of the division algorithm. Since we are not interested in the quotient, we ignore the generation of the bits of the quotient in the following algorithm. The procedure given below first left-aligns the operands P and n. Since P is $2k$-bit number and n is a k-bit number, the left alignment implies that n is shifted k bits to the left, i.e., we start with $2^k n$. Furthermore, the initial value of R is taken to be P, i.e., $R_0 = P$. We then subtract the shifted n from P to obtain R_1; if R_1 is positive or zero, we continue to the next step. If it is negative the remainder is restored to its previous value.

The Restoring Division Algorithm

Inputs: P, n

Output: $R = P \bmod n$

1. $R_0 \coloneqq P$

2. $n \coloneqq 2^k n$

3. **for** $i = 1$ **to** k

4. $R_i \coloneqq R_{i-1} - n$

5. **if** $R_i < 0$ **then** $R_i \coloneqq R_{i-1}$

6. $n \coloneqq n/2$

7. **return** R_k

In Step 5 of the algorithm, we check the sign of the remainder; if it is negative, the previous remainder is taken to be the new remainder, i.e., a restore operation is performed. If the remainder R_i is positive, it remains as the new remainder, i.e., we do not restore. The restoring division algorithm performs k subtractions in order to reduce the $2k$-bit number P modulo the k-bit number n.

4.1.1.3. Nonrestoring Division Algorithm

The nonrestoring division algorithm allows a negative remainder. In order to correct the remainder, a subtraction or an addition is performed during the next cycle, depending on the whether the sign of the remainder is positive or negative, respectively. This is based on the following observation: Suppose $R_i = R_{i-1} - n < 0$, then the restoring algorithm assigns $R_i \coloneqq R_{i-1}$ and performs a subtraction with the shifted n, obtaining

$$R_{i+1} = R_i - n/2 = R_{i-1} - n/2.$$

However, if $R_i = R_{i-1} - n < 0$, then one can instead let R_i remain negative and add the shifted n in the following cycle. Thus, one obtains

$$R_{i+1} = R_i + n/2 = (R_{i-1} - n) + n/2 = R_{i-1} - n/2,$$

which would be the same value. The steps of the nonrestoring algorithm, which implements this observation, are given below:

The Nonrestoring Division Algorithm

Inputs: P, n

Output: $R = P \bmod n$

1. $R_0 \coloneqq P$

2. $n \coloneqq 2^k n$

3. **for** $i = 1$ **to** k

4. **if** $R_{i-1} > 0$ **then** $R_i \coloneqq R_{i-1} - n$

5. **else** $R_i \coloneqq R_{i-1} + n$

6. $n \coloneqq n/2$

7. **if** $R_k < 0$ **then** $R \coloneqq R + n$

8. **return** R_k

Note that the nonrestoring division algorithm requires a final restoration cycle in which a negative remainder is corrected by adding the last value of n back to it.

4.1.1.4. Montgomery's Multiplication Algorithm

In 1985, P. L. Montgomery introduced an efficient algorithm [18] for computing $R = a \cdot b \bmod n$ where a, b, and n are k-bit binary numbers. The algorithm is particularly suitable for implementation on general-purpose computers (signal processors or microprocessors) which are capable of performing fast arithmetic modulo a power of 2. The Montgomery reduction algorithm computes the resulting k-bit number R without performing a division by the modulus n. Via an ingenious representation of the residue class modulo n, this algorithm replaces division by n operation with division by a power of 2. This operation is easily accomplished on a computer since the numbers are represented in binary form. Assuming the modulus n is a k-bit number, i.e., $2^{k-1} < n < 2^k$, let r be 2^k. The Montgomery reduction algorithm requires that r and n be relatively prime, i.e., gcd(r, n) = gcd(2^k, n) = 1. This requirement is satisfied if n is odd. In the following we summarize the basic idea behind the Montgomery reduction algorithm.

Given an integer $a < n$, we define its n-residue with respect to r as

$$\bar{a} = a \cdot r \bmod n.$$

It is straightforward to show that the set

$$\{i \cdot r \bmod n \mid 0 \le i \le n-1\}$$

is a complete residue system, i.e., it contains all numbers between 0 and n-1. Thus, there is a one-to-one correspondence between the numbers in the range 0 and n-1 and the numbers in the above set. The Montgomery reduction algorithm exploits this property by introducing a much faster multiplication routine which computes the n-residue of the product of the two integers whose n-residues are given. Given two n-residues $\bar{a}$ and $\bar{b}$, the Montgomery product is defined as the n-residue

$$\bar{R} = \bar{a} \cdot \bar{b} \cdot r^{-1} \bmod n$$

where r^{-1} is the inverse of r modulo n, i.e., it is the number with the property

$$r^{-1} \cdot r = 1 \bmod n.$$

The resulting number $\bar{R}$ is indeed the n-residue of the product

$$R = a \cdot b \bmod n$$

since

$$\bar{R} = \bar{a} \cdot \bar{b} \cdot r^{-1} \bmod n = a \cdot r \cdot b \cdot r \cdot r^{-1} \bmod n = a \cdot b \cdot r \bmod n.$$

In order to describe the Montgomery reduction algorithm, we need an additional quantity, n', which is the integer with the property

$$r \cdot r^{-1} - n \cdot n' = 1.$$

The integers r^{-1} and n' can both be computed by the extended Euclidean algorithm [14]. The Montgomery product algorithm, which computes

$$u = \bar{a} \cdot \bar{b} \cdot r^{-1} (\bmod n)$$

given $\bar{a}$ and $\bar{b}$, is given next:

Montgomery Product

Inputs: $\bar{a}, \bar{b}, n, n', r$

Output: $u = \bar{a} \cdot \bar{b} \cdot r^{-1} \pmod{n}$

1. $t := \bar{a} \cdot \bar{b}$

2. $m := t \cdot n' \bmod r$

3. $u = (t + m \cdot n)/r$

4. **if** $u \geq n$ **then return** $u - n$

else return u

The most important feature of the Montgomery product algorithm is that the operations involved are multiplications modulo r and divisions by r, both of which are intrinsically fast operations since r is a power 2. The MonPro algorithm can be used to compute the product of a and b modulo n, provided that n is odd.

Montgomery Multiplication

Inputs: $\bar{a}, \bar{b}, n, r$

Output: $u = a \cdot b \pmod{n}$

1. Compute n' using the extended Euclidean algorithm.

2. $\bar{a} := a \cdot r \bmod n.$

3. $\bar{b} := b \cdot r \bmod n.$

4. $\bar{x} := \text{MonPro}(\bar{a}, \bar{b}, n', n, r)$ (montgomery product)

5. $x := \text{MonPro}(\bar{x}, 1, n', n, r)$ (montgomery product)

6. **return** x

However, the preprocessing operations, especially the computation of n', are rather time-consuming. Thus, it is not a good idea to use the Montgomery product computation algorithm when a single modular multiplication is to be performed.

4.1.2. Montgomery Exponentiation

The Montgomery product algorithm is more suitable when several modular multiplications with respect to the same modulus are needed. Such is the case when one needs to compute a modular exponentiation, i.e., the computation of $M^e \bmod n$. In the following we summarize the modular exponentiation operation which makes use of the Montgomery product function MonPro. The exponentiation algorithm uses the binary method showed in section 4.1.1.

Montgomery Exponentiation

Inputs: M, e, n {n is an odd number}

Output: $M^e \bmod n$

1. Compute n' using the extended Euclidean algorithm.

2. $\bar{M} := M \cdot r \bmod n$

3. $\bar{x} := 1 \cdot r \bmod n$

4. **for** $i := k - 1$ **down to** 0 **do**

5. $\bar{x} := \text{MonPro}(\bar{x}, \bar{x})$

6. **if** $e_i = 1$ **then** $\bar{x} := \text{MonPro}(\overline{M}, \bar{x})$

7. $x := \text{MonPro}(\bar{x}, 1)$

8. **return** x

Thus, we start with the ordinary residue M and obtain its *n*-residue $\overline{M}$ using a division-like operation, which can be achieved, for example, by a series of shift and subtract operations. Additionally, Steps 2 and 3 require divisions. However, once the preprocessing has been completed, the inner-loop of the binary exponentiation method uses the Montgomery product operations which performs only multiplications modulo 2^k and divisions by 2^k. When the binary method finishes, we obtain the *n*-residue $\bar{x}$ of the quantity $x = M^e \bmod n$. The ordinary residue number is obtained from the *n*-residue by executing the MonPro function with arguments $\bar{x}$ and 1. This is easily shown to be correct since

$$\bar{x} = x \cdot r \pmod{n}$$

immediately implies that

$$x = \bar{x} \cdot r^{-1} \pmod{n} = \bar{x} \cdot 1 \cdot r^{-1} \pmod{n} = \text{MonPro}(\bar{x}, 1).$$

The resulting algorithm is quite fast as was demonstrated by many researchers and engineers who have implemented it. However, this algorithm can be refined and made more efficient, particularly when the numbers involved are multi-precision integers [7].

4.2. RSA Architectures

By researching the technical literature [15-17, 20-23, 26], we can basically find three different branches of study in the RSA architecture field: Chinese Remainder Theorem (CRT) based architectures, Pipelined based architectures and Residue Number System (RNS) based architectures. It should be mentioned that all of these architectures feature the Montgomery modular multiplication algorithm in their implementation.

4.2.1. CRT Based Architecture

The Chinese Remainder Theorem technique is known to reduce the RSA computation by divide-and-conquer method, i.e., by splitting the computation into two distinguished parts. Some studies have proven that CRT can improve the overall throughput of the system up to 4 times when the factors *p* and *q* have the same bit size [15]. However, to perform these steps, the factors of the modulus *n*, *p* and *q*, are assumed to be known. The Chinese Remainder Theorem (CRT) can be stated as follows:

Let $m_0, m_1, \ldots, m_{n-1}$ be pairwise relatively prime positive integers and let $x_0, x_1, \ldots, x_{n-1}$ be any integers which satisfy the linear congruence system in one variable given by [15]

$$X \equiv x_0 \pmod{m_0}$$
$$X \equiv x_1 \pmod{m_1}$$
$$\vdots$$
$$X \equiv x_{n-1} \pmod{m_{n-1}}$$

has a unique solution modulo $m_0 \cdot m_1 \ldots m_{n-1}$.

By CRT, the computation of $M = C^d \bmod n$ can be partitioned into two parts:

$$M_p = C_p^{d_p} \pmod{p} \qquad (4.2.1.1)$$

$$M_q = C_q^{d_q} (\mathrm{mod}\, q) \tag{4.2.1.2}$$

where

$$C_p = C(\mathrm{mod}\, p), d_p = d(\mathrm{mod}\, p-1), \tag{4.2.1.3}$$

$$C_q = C(\mathrm{mod}\, q), d_q = d(\mathrm{mod}\, q-1), \tag{4.2.1.4}$$

This reduces computation time since $d_p, d_q < d$ and $C_p, C_q < C$. In fact, their sizes are about half the original sizes. Finally, we compute M by CRT as follows:

$$M = \left(M_p \left(q^{-1} (\mathrm{mod}\, p)\right) q + M_q \left(p^{-1} (\mathrm{mod}\, q)\right) p\right) (\mathrm{mod}\, n). \tag{4.2.1.5}$$

Figure 4-1 shows a diagram of a CRT-based architecture. Notice that before applying the RSA algorithm (equation 3.1.5) some pre-computation is required (Modular Units Blocks). These blocks perform equations 4.2.1.4 and 4.2.1.3. Afterwards, the modular exponentiation units execute the RSA algorithm on $k/2$ bits operands (equations 4.2.1.1 and 4.2.1.2). The Montgomery algorithm can be applied to boost this module. At last, the post-processing unit converts the $k/2$-bits plaintext into a k bit plaintext by CRT (equation 4.2.1.5).

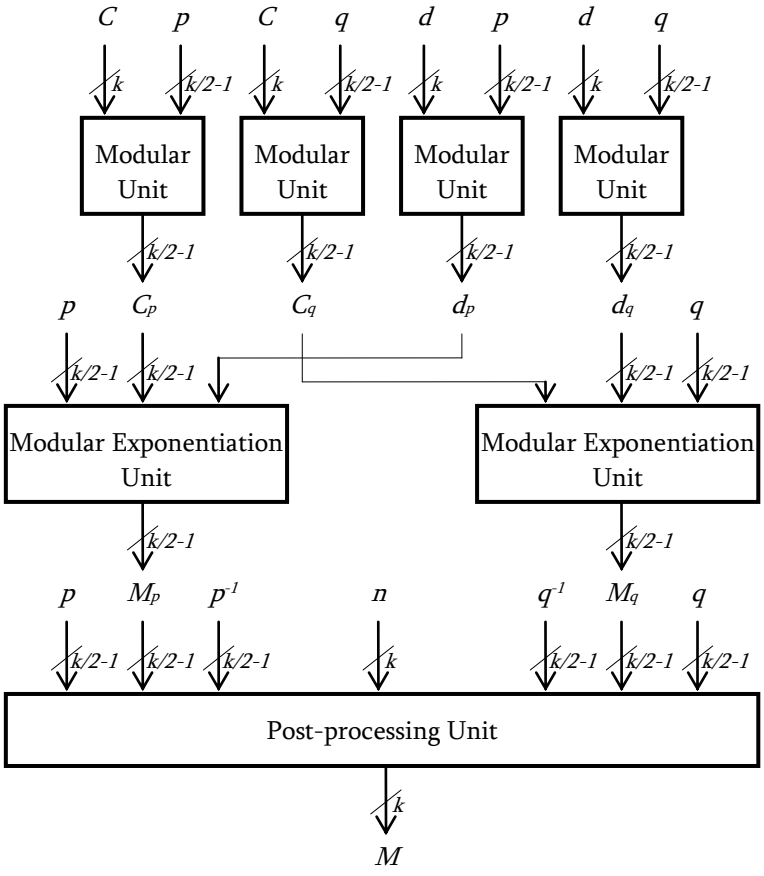


Figure 4-1: CRT Based Architecture

The basic advantages of this architecture are that it can be easily parallelized due to the initial pre-computation. Also, the exponentiation unit works on $k/2$-bit operands, boosting RSA computation in 4 times if p and q have half the size of the modulus n. Only in the last stage (post-processing unit) it will be employed k-bit computations.

However, as demonstrated in [19], CRT-Based architectures are not secure since they can be attacked by hardware (Bellcore Attacks), i.e., injecting spikes in the circuit when encrypting/decrypting a

message (by introducing any electrical noise during ciphering time). As an example, assume that during the decryption of a C ciphertext a random error occurs during the computation of M_p (equation 4.2.1.1). This yields a faulty decrypted message M_p^*, whereas the computation of M_q is done correctly (equation 4.2.1.2). The combination of M_p^* and M_q via equation (4.2.1.5) will yield an incorrect decrypted message M^*. For M^* it holds that $M - M^* \neq 0$ but $M - M^* \equiv 0 \bmod q$. Therefore, one obtains the factorization of n by computing

$$\gcd\left(\left(C-(M^*)^e\right) \bmod n, n\right) = q\,.$$

Many researchers are trying to find a solution to this problem through different algorithms [20-24]; however, their solutions are still not secure. On average, CRT based architectures can encrypt 400 kbits/s [15].

4.2.2. RNS Based Architecture

In Residue Number System (RNS), an integer x is represented by $x = \{x[a_1], x[a_2], \ldots, x[a_m]\}$, where $x[a_i] = x \bmod a_i$. The set $a = \{a_1, a_2, \ldots, a_m\}$ is called base and the number of elements m is its base size. Each element inside the set a is also called modulus. The components in the base are required to satisfy $\gcd(a_i, a_j) = 1$ for $i \neq j$, i.e., they must be pairwise relative primes [17].

In RNS, the result of any arithmetic operation must be inside its dynamic range, i.e., it should be within de legitimate interval [0, *N-1*], where

$$N = \prod_{i=1}^{m} a_i\,.$$

Within this dynamic range every number can be represented by a unique set of residues. Each integer number x, in this dynamic range is mapped onto the legitimate range and represented as an *m-tuple* of residue digits $\{r_1, r_2, \ldots, r_m\}$. RNS can also represent negative numbers, however, the dynamic range changes. If N is odd, the range becomes $[-(N-1)/2, (N-1)/2]$. Otherwise, the range is $[-N/2, N/2-1]$. The example above is intended to show how to create a small residue number system.

Let an RNS has 2 moduli: $a_1 = 3$ and $a_2 = 5$. For this system, $N = \prod_{i=1}^{2} a_i = 3 \cdot 5 = 15$. The legitimate range of the system is $[0, N-1] = [0,15-1] = [0,14]$ for positive numbers and $[-(N-1)/2, (N-1)/2] = [-(15-1)/2, (15-1)/2] = [-7,7]$ as the N is odd. The table below shows the complete system:

Signed Integer Numbers	**Unsigned Integer Numbers**	Mod 3	Mod 5	**Signed Integer Numbers**	**Unsigned Integer Numbers**	Mod 3	Mod 5
0	**0**	0	0	**-7**	**8**	2	3
1	**1**	1	1	**-6**	**9**	0	4
2	**2**	2	2	**-5**	**10**	1	0
3	**3**	0	3	**-4**	**11**	2	1
4	**4**	1	4	**-3**	**12**	0	2
5	**5**	2	0	**-2**	**13**	1	3
6	**6**	0	1	**-1**	**14**	2	4
7	**7**	1	2	**x**	**x**	x	x

Table 4-1: RNS representation

The number $\{2,1\}$ in this number system can represent either -4 or 11, depending on the representation required (unsigned or signed representation). An operation like 4-2=2, using this residue system, can be expressed either

$$4-2=\{1,4\}-\{2,2\}=\{1-2 \bmod 3, 4-2 \bmod 5\}=\{2,2\}=2$$ unsigned representation

or

$$4+(-2)=\{1,4\}+\{1,3\}=\{1+1 \bmod 3, 4+3 \bmod 5\}=\{2,2\}=2\,.$$ signed representation

Also, we have

$$3\cdot(-2)=\{0,3\}\cdot\{1,3\}=\{0\cdot 1 \bmod 3, 3\cdot 3 \bmod 5\}=\{0,4\}=-6\,.$$ signed representation

Notice that addition, subtraction and multiplication are inherently carry-free, which means that each digit of the result is a function of only one digit from each operand and independent of the others. This is the most attractive feature of RNS that enables us to design highly parallel structure for computation in order to gain high speed for DSP applications and large bit number operations.

For interested readers, a good tutorial on RNS can be found on the web [25].

Figure 4-2 shows a simple diagram of a RNS-based architecture.

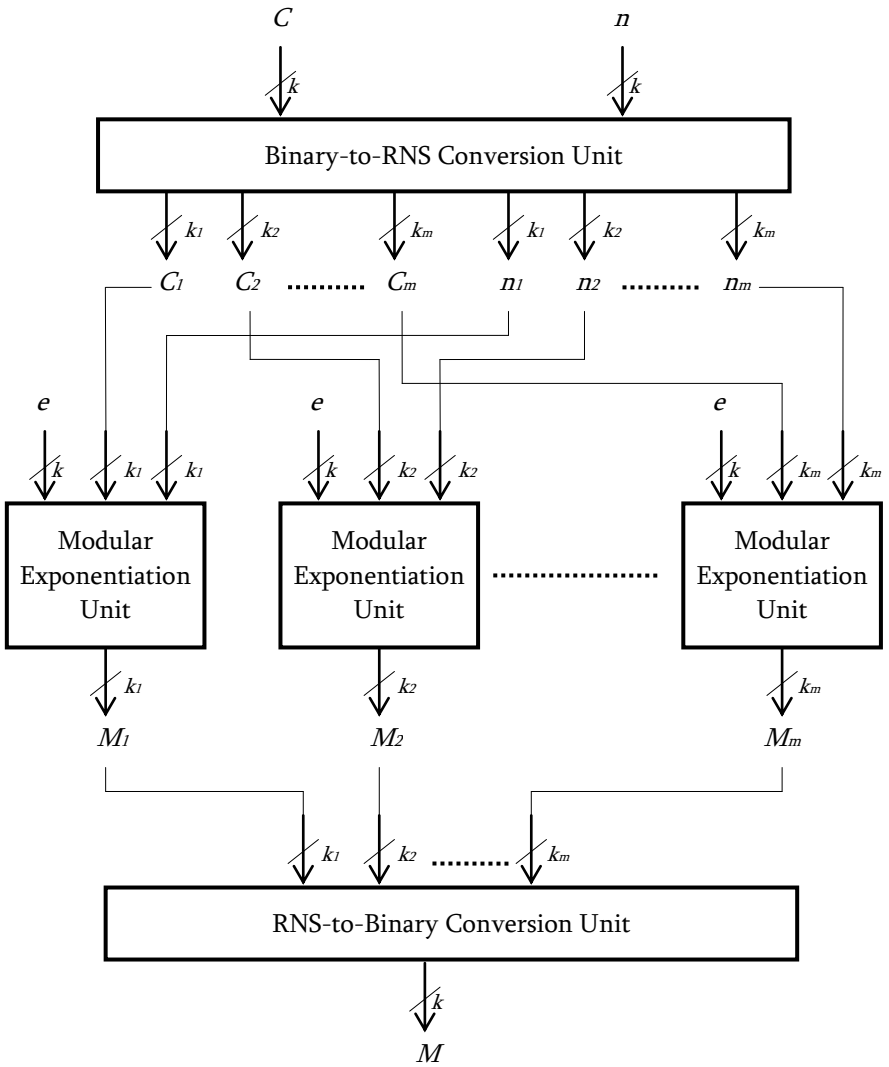


Figure 4-2: RNS Based Architecture

Notice there is no operation on the factors p and q, which means that CRT can be applied. But remember from section 4.2.1 that CRT is still insecure. Also, Montgomery's algorithm may be employed in the Modular Exponentiation Unit, improving the system performance.

The main point of this architecture is to develop good conversion units. To do that, one should find an appropriate base size m and its modulus. In figure 4-2, the base size was chosen to be m, having each modulus k_i bits. Also, the number of parallel units is identical to m. In this case, we have the maximum parallelism. However, this is not necessary and the numbers of processing units can be chosen to be less than the base size. Less hardware is used but time-sharing processing now is required in each modular exponentiation unit. It should be mentioned the base set need to be precomputed and stored in a ROM.

The great disadvantage of this architecture is that when the operand sizes are small, i.e., RSA key size less than 1024 bits, the pre and post processing are very time-consuming, becoming this technique inadequate. Its design is complex too. However, according to [17], this implementation becomes quite good when involving large key sizes. On average, RNS based architectures can encrypt 300 kbits/s [17].

4.2.3. Pipelined Architecture

The great advantage of this architecture is that there is no need to worry about the factors p and q. As the modulus is not in its factored form ($n=pq$) the only way to discover them is through factorization, something infeasible. Thus, these architectures are secure against Bellcore attacks.

The concern here is focused on developing a fast modular exponentiation unit where the operand sizes X and Y have the same size of the modulus N. Futhermore, we can break up these operands into several words and apply pipeline techniques. As an example of architectures featuring these characteristics we can cite [16] and [26].

This architecture requires no pre and post processing. All the design effort is concentrated on the modular exponentiation unit. As modular exponentiation operations perform a series of modular multiplications only the modular multiplication unit is depicted in figure 4-3.

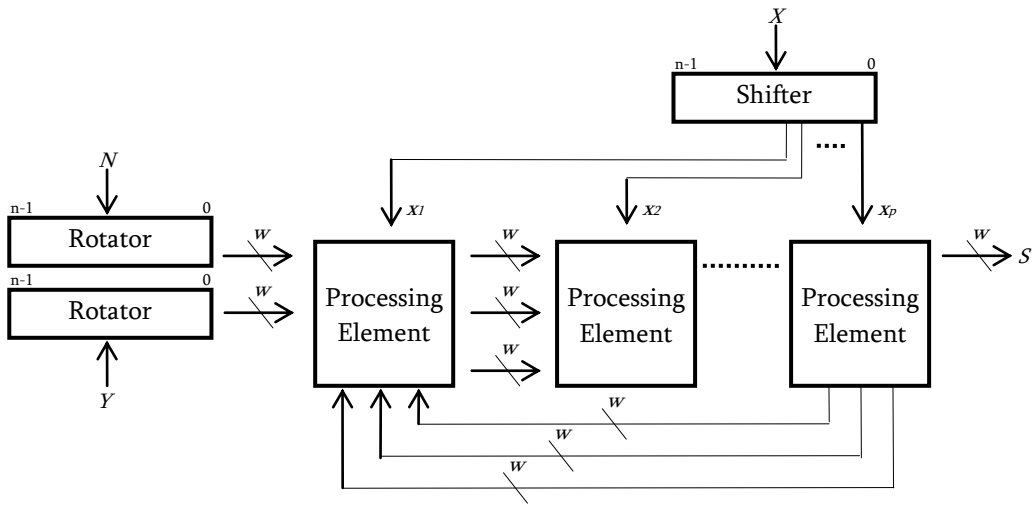


Figure 4-3: Pipeline Based Architecture

Notice that as data is transferred word-serially to the pipeline registers which store Y and N work as rotators. The processing elements itself must relay the received words to the next units in the pipeline. All paths are w bits wide, except for the x_i (only 1 bit). The values of x_i comes from a p-shift register where p equals to the number of processing elements in the pipeline. The register for S can be a shift register since its contents are not reused.

On average, pipeline based architectures can encrypt 40 kbits/s [16].

4.3. Conclusions

As could be seen in section 4.2.1, many documents showed that CRT-based architectures are still not reliable. According to Aumüller [19] only sophisticated hardware countermeasures (sensors, filters, etc.) in combination with software countermeasures will be able to provide security. Also, Aumüller demonstrated that many current smartcards with RSA coprocessors are susceptible to Bellcore attacks.

By trying to find a solution to this problem, other articles were studied [20-23] but none of them showed to be trustworthy. As they are still looking for a solution to a recent discovered fault, they couldn't find a trusted method of encrypting using CRT (every new article points some errors to the latest ones). These latest papers discouraged us to develop any CRT-based architecture until a secure method be found.

Summarizing, the choice of a CRT-based architecture implies more control hardware and software security schemes due to Bellcore attacks to the benefit of faster encryption.

RNS based-architectures demonstrated to be another efficient method to sign messages quickly (section 4.2.2). By converting binary numbers to a different number system, a parallelized architecture can be implemented, and high-speed data rates can be achieved. Also, these architectures can be boosted with CRT techniques. However, they are not intended to RSA cryptosystems with small key sizes, since the conversion steps from one number system to another are very time-consuming.

RNS-based architectures seem to be more suitable when the key size is more than 2048 bits. But at the present moment 1024-bit RSA key sizes are more than enough to secure data as explained in the chart below:

Protection Lifetime of Data	**Present -2010**	**Present - 2030**	**Present – 2031 and beyond**
Minimum RSA key size	1024 bits	2048 bits	3072 bits

Table 4-2: Recommended RSA keys sizes based on protection life [8]

In addition to that, as stated by [17], current 1024-bit CRT-based architectures have practically the same performance of 1024-bit RNS-based architectures. Thus, we should either choose a very complex architecture design or hardware and software countermeasures.

Pipelined architectures can also implement high-speed RSA encryption. In order to improve speed at higher bit-lengths it is necessary to break the multiplication up into stages, and pipeline the calculation. This improved performance significantly [16, 26]. Depite being up to 10 times slower than RNS and CRT based architectures, pipelined-based architectures provide security and less area occupied in the chip (they don't need special hardware countermeasures or heavy pre and post computations units). Furthermore, RSA applications are not intended to be used in a whole section of a high-speed communication. We should use private-key cryptosystems instead. RSA applications are more suitable to sign messages, verify authenticity or to begin a secure high-speed communication as showed in section 2.4.3.

After analyzing each case separately we have chosen the pipelined-based architecture to design the RSA coprocessor. It has the great advantage that there is no need to worry about security countermeasures or a very complex design. Moreover, the throughput obtained for a typical 1024-bit key size is approximately 40 kb/s [16], i.e., this rate is more than enough to sign messages with our smart cards in our daily life. Therefore, our RSA system will be based on a pipelined modular exponentiation unit. More details will be covered in the next sections.

5. Coprocessor IP Specification

In the previous section we presented specific arithmetic algorithms and three different implementations of RSA systems used nowadays. After analyzing the advantages and disadvantages among them we finally decided to implement a pipelined-based architecture.

Figure 5-1 shows our proposed block diagram. To avoid confusion in the schema, we preferred not to include connections from the main control unit to other blocks. The vertical green line connecting all the blocks is the system data bus. More details are covered in the next sections.

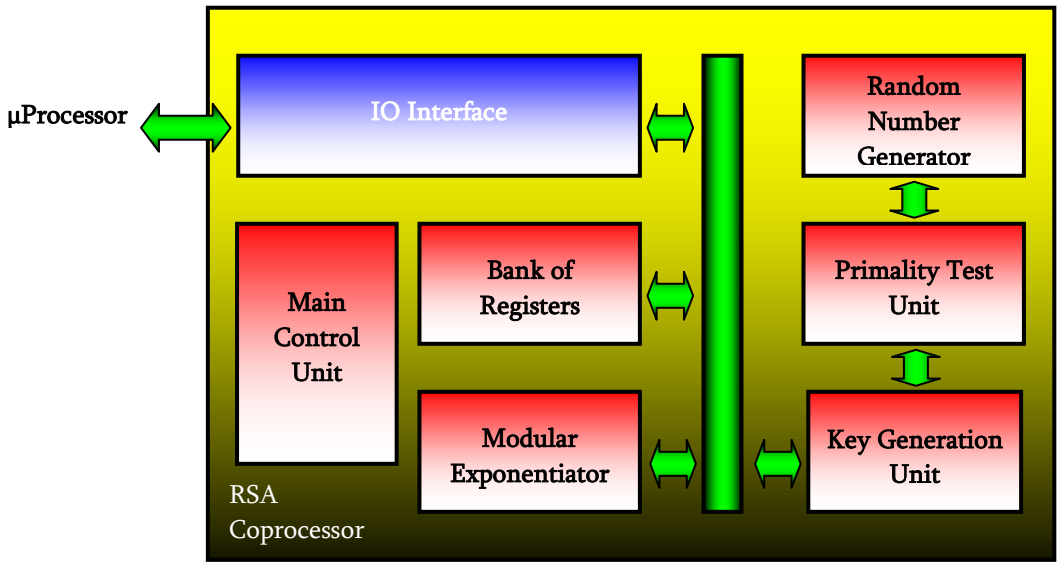


Figure 5-1: Block Diagram

5.1. IO Interface

This block connects the outside microprocessor interface to the IP. Many types of microprocessors with different data bus sizes can be used to control it. So, this IP is adjustable in accordance with the user's need (parameterizable input/output data bus). Table 5-1 summarizes this cell.

Pin Name	Size	Direction	Active	Description
System Signals				
Clock	1	input	rising	System clock. Data is transferred on every positive clock edge.
Reset	1	input	high	Asynchronous system reset.
Data Signals				
DataBus[4x2^n-1:0]	S[2]	input output	-	Data input and output. The parameter *n* must be an integer in the range of {0, 1, 2, 3}, i.e., this IP supports 4, 8, 16 and 32 bits input bus sizes. This parameter must be defined before synthetization.

Table 5-1: IO Interface cell pinout

Internal and external data buses have the same size.

[2] S = Scalable

5.2. Bank of Registers

When the coprocessor is working (busy is high) the bank of registers is inaccessible and the operands in it are organized depending on the value of Mode (see later section 5.7). Figure below illustrates this organization.

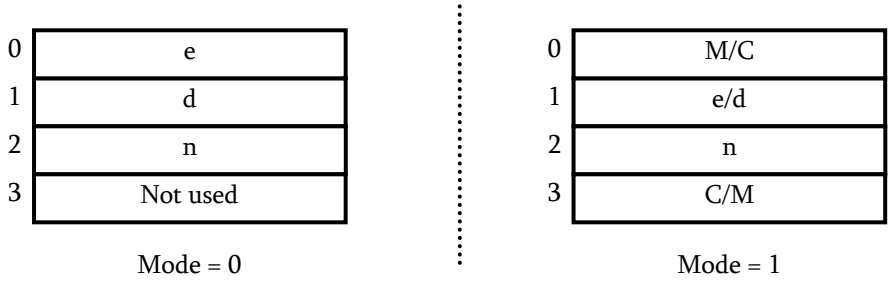


Figure 5-2: Memory Organization

When Mode is set to '0', the RSA coprocessor is working in key generation mode. No data is required before computation since keys will be generated randomly. To start the key generation process, busy must be set to high. After finishing, the memory will be arranged as shown in figure 5-2 on the left.

However, if Mode is set to '1', the RSA coprocessor will work on encryption/decryption mode. Data (M/C, e/d and n) must be loaded in memory before beginning the computation, and afterwards *start* must be set to '1'. When *start* changes back to '0', the computation is finished and the operands will be arranged as showed in figure 5-2 on the right. The final result will be stored in C/M.

Cell input and output ports are described in table below (table 5-2).

Pin Name	Size	Direction	Active	Description
Clock	1	input	Rising	System clock
DataBus	S	input output	-	Connects system memory to systems bus.
Read	1	input	high	Write contents of selected memory address to the system bus.
Write	1	input	high	Write contents of system bus to selected memory address.
SelAdd	2	input	-	Select one of the four possible 1024-bit registers located in the bank of register to be written/read.

Table 5-2: Input and output port for the Bank of Register

5.3. Random Number Generator

This module generates 512-bit odd random numbers. It is nothing more than a counter which is incremented every clock cycle. However, to guarantee 512-bit odd numbers in the output the first and last bits are always set to 1. Figure 5-3 shows it.

Figure 5-3

The module ports are described in the table below.

Pin Name	Size	Direction	Active	Description
Clock	1	input	rising	System clock
RNGout	512	output	-	Generator output

Table 5-3: port description for the Random Number Generator Unit

However, this module is very simple. Better random number generators can be found in [27] and [28].

5.4. Primality Test Unit

It tests if a 512-bit number is prime or not. Its ports are described in the table below.

Pin Name	Size	Direction	Active	Description
Clock	1	input	rising	System clock
Start	1	input output	high	Load input data and starts computation. It remains high until the required operation is finished.
isPrime	1	output	high	Indicates if the input data is a prime number or not. Set to zero when detects a start rising edge.
DataIn	512	input	-	Input data
DataOut	1024	output	-	Output data

Table 5-4: port description for the Primality Test Unit

Since our RSA system needs always two 512-bit prime numbers to generate keys (p and q factors), the output of this module gives p concatenated to q. The first found prime number is stored inside the block while the second one is being tested. Thus, the output will only be available when two consecutive prime numbers are found. This characteristic will be clarified to the reader in the next section.

5.5. Key Generation Unit

By taking the two factors, p and q, this cell computes the pair of keys (e, n) and (d, n). The modulus, the private and the public exponent have 1024 bits. Data input and data output ports are 1024 bit wide. As both factors have the same size (512 bits), we send them to the input port as a concatenated 1024-bit number. Therefore, the input can be thought as a 1024-bit operand. This cell uses the primality test unit since random prime numbers are required to accomplish its task.

After finishing the computation, all key operands will be stored in their respective memory addresses and start will be set to '0' (see section 5.2). In the following lines we summarize its ports.

Pin Name	Size	Direction	Active	Description
Clock	1	input	rising	System clock
Start	1	input output	high	Load input data and starts computation. It remains high until the operation required is finished. Afterwards, all operands (e, d and n) are in their respective memory addresses.
AskPQ	1	input output	high	Ask two 512-bit random prime numbers to be computed by the Primality Test Unit. It remains high until pq are in its input port.
DataIn	1024	input	-	Input data
DataOut	1024	output	-	Output Data. Connected to system bus.

Table 5-5: port description for the Key Generator Unit

5.6. Modular Exponentiator

This module computes the RSA algorithm. All input operands (*M/C*, *e/d* and *n*) must be in their respective addresses before starting. When computation is over the final result is stored in memory (see figure 5.2).

Table 5-6 specifies its input and output ports.

Pin Name	Size	Direction	Active	Description
Clock	1	input	rising	System clock
Start	1	input output	high	Load input data from memory and starts computation. It remains high until the operation required is finished. When start changes back to zero, the final result (*C/M*) is stored in its respective memory address.
IOData	1024	input output	-	Input/Output data. Connected to system bus.

Table 5-6: port description for the Modular Exponentiator

5.7. Main Control Unit

It controls all signals from later modules. These signals are divided by categories i.e, signals coming/going from/to specific blocks. They are described in the table below.

Pin Name	Size	Direction	Active	Description
IO Interface Signals				
Clock	1	input	rising	System clock.
Reset	1	input	high	Asynchronously system reset.
Mode	1	input	-	Change the operation mode.
Read	1	input	high	Transfer data from the address specified by AddI.
Write	1	input	high	Transfer data to the address specified by AddI.
Start	1	output	-	Interrupt signal for the microprocessor.
Bank of Registers Signals				
ReadMemory	1	output	high	Transfers contents from selected memory address to interface shift register.
WriteMemory	1	output	high	Transfers contents from interface shift register to selected register.
Key Generation Unit Signals				
StartKG	1	input output	high	Start key generation process. It remains high until the operation required is finished. When operation is over then generated keys are stored in their respective places (see figure 5-5).
Modular Exponentiator Signals				
StartRSA	1	input output	high	Start encryption/decryption process. It remains high until the operation required is finished. The result is stored in its respective place when computation is over (see figure 5-5).

Table 5-7: port description for the Main Control Unit

Notice that Random Number Generator and Primality Test Unit are not controlled by this unit. They have a local control performed by the Key Generation Unit.

6. IP Implementation

As could be seen in section 4.1.2, the Montgomery exponentiation algorithm is the most suitable solution for implementing in digital systems. Furthermore, as it is cited in many articles nowadays [6-7, 15-17, 26] we can consider it as trustworthy implementation.

From time to time, key sizes must be changed since computation power is always increasing. To avoid rewrite code every time a key is broken a good solution is to write a scalable exponentiator architecture. Since we chose a pipelined-based architecture (section 4.3) our goal here is to describe the design of a scalable Montgomery Multiplier (MM) with no limitation on the maximum number of bits manipulated by the multiplier. To do such an operation, we need to break up the operands into small words according to the available area and/or desired performance.

6.1. Multiple Word Raidx-2 Montgomery Multiplication Algorithm

This section was was extracted from [26].

The use of short precision words reduces the broadcast problem in the circuit implementation. The broadcast problem corresponds to the increase in the propagation delay of high-fanout signals. Also, a word-oriented algorithm provides the support we need to develop scalable hardware units for the MM. Next paragraphs explain the algorithm proposed by [26].

Let us consider w-bit words. For operands with n bits of precision, $e = \lceil (n+1)/w \rceil$ words are required. The extra bit used in the calculation of e is needed since it is known that S (internal variable of the algorithm) is in the range $[0; 2M-1]$, where M is the modulus. Thus the computations must be done with an extra bit of precision. The input operands will need an extra 0 bit value at the leftmost bit position in order to have the precision extended to the correct value.

The operand Y (multiplicand) is scanned word-by-word, and the operand X (multiplier) is scanned bit-by-bit. We will make use of the following notation:

$$M = \left(M^{(e-1)}, \ldots, M^{(1)}, M^{(o)}\right),$$
$$Y = \left(Y^{(e-1)}, \ldots, Y^{(1)}, Y^{(o)}\right),$$
$$X = \left(x_{m-1}, \ldots, x_1, x_0\right),$$

where the words are marked with superscripts and the bits are marked with subscripts. The concatenation of vectors A and B is represented as (A, B). A particular range of bits in a vector A from position i to position j, $j > i$, is represented as A_{ij}. The bit position i of the k_{th} word of A is represented as $A_i^{(k)}$. The algorithm is given below.

Multiple-Word Radix-2 Montgomery Multiplication Algorithm (MWR2MM)

Inputs: X, Y, M

Output: $S = X \cdot Y \cdot r^{-1} \bmod M$

1. $S = 0$

2. **for** $i := 0$ **to** $n-1$ **do**

3. $\left(C, S^{(0)}\right) := x_i Y^{(0)} + S^{(0)}$

4. **if** $S_0^{(0)} = 1$ **then**

5. $\left(C, S^{(0)}\right) := \left(C, S^{(0)}\right) + M^{(0)}$

6. **for** $j := 1$ **to** $e-1$ **do**

7. $\left(C, S^{(j)}\right) := C + x_i Y^{(j)} + M^{(j)} + S^{(j)}$

8. $S^{(j-1)} := \left(S_0^{(j)}, S_{w-1..1}^{(j-1)}\right)$

9. $S^{(e-1)} := \left(C, S_{w-1..1}^{(e-1)}\right)$

10. **else**

11. **for** $j := 1$ **to** $e-1$ **do**

12. $\left(C, S^{(j)}\right) := C + x_i Y^{(j)} + S^{(j)}$

13. $S^{(j-1)} := \left(S_0^{(j)}, S_{w-1..1}^{(j-1)}\right)$

14. $S^{(e-1)} := \left(C, S_{w-1..1}^{(e-1)}\right)$

The MWR2MM algorithm computes a partial sum *S* for each bit of *X*, scanning the words of *Y* and *M*. Once the precision is exhausted, another bit of *X* is taken, and the scan is repeated. Thus, the algorithm imposes no constraints to the precision of operands. The arithmetic operations are performed in precision *w* bits, and they are independent of the precision of operands. What varies is the number of loop iterations required to accomplish the modular multiplication. The carry variable *C* must be in the set {0, 1, 2}. This condition is imposed by the addition of the three vectors *S, M* and *xiY*. To have containment in the addition of 3 *w*-bit words and a maximum carry value C_{max} (generated by previous word addition), the following equation must hold:

$$3 \cdot \left(2^w - 1\right) + C_{\max} \leq 2^w \cdot C_{\max} + 2^w - 1$$

which results in $C_{\max} \geq 2$. Thus, choosing $C_{\max} = 2$ is enough to satisfy the containment condition.

The dependency between operations within the loop for *j* restricts their parallel execution due to dependency on the carry. However, parallelism is possible among instructions in different *j* loops. See the dependency graph for the MWR2MM algorithm in figure 6-1.

Each circle in the graph represents an atomic computation and is labeled according to the type of action performed. Task A corresponds from lines 3 to 5: test the least significant bit of *S* to determine if *M* should be added to *S* during this and addition of words from *S*, *xiY* and *M* (depending on the test performed). Task B corresponds to operations from 7 to 9. We observe from this graph that the degree of parallelism and pipelining can be very high. Each column in the graph may be computed by a separate processing element (PE), and the data generated from one PE may be passed to another PE in a pipelined fashion.

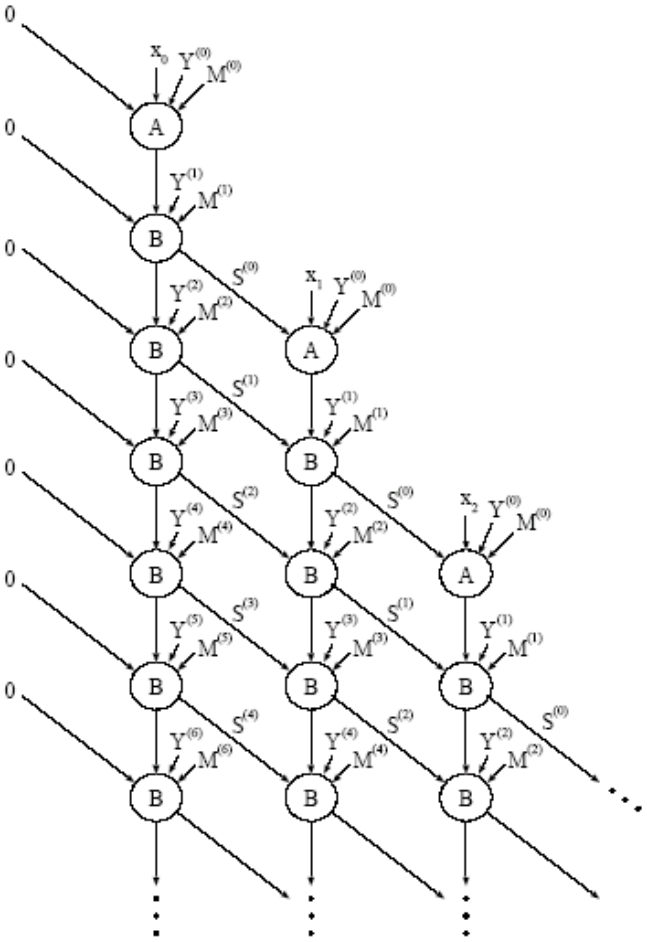


Figure 6-1: dependency graph for the MWR2MM Algorithm [26]

An example of the computation executed for 5-bit operands is shown in Figure 6-2 for the word size of $w = 1$ bit. Since the *jth* word of each input operand is used to compute word *j*-1 of *S*, the last B task in each column must receive $M_{(e)} = Y_{(e)} = 0$ as inputs. This condition is enough to guarantee that $M_{(e-1)}$ will be generated based only on the internal PE information. Note also that there is a delay of 2 clock cycles between processing a column for *xi* and a column for *xi+1*. The total execution time for the computation shown in Figure 6-2 is 14 clock cycles.

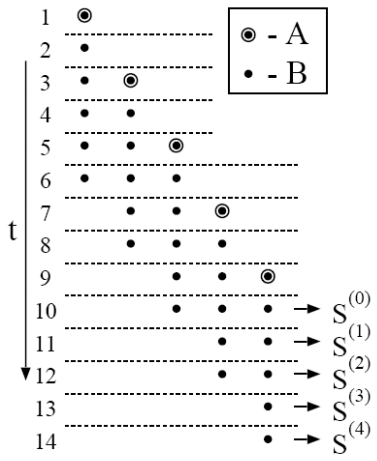


Figure 6-2: An example of computation for 5-bit operands, where w=1 bit [26]

Tasks A and B are performed on the same hardware module. The local control circuit of the module must be able to read the least significant bit of $S^{(0)}$ at the beginning of the operation, and keep this value for the entire operand scanning. Recall that the even condition of $S_0^{(0)}$ determines if the processing unit should add M to the partial sum during the pipeline cycle. The pipeline cycle is the sequence of steps that a PE needs to execute to process all words of the input operands.

The maximum degree of parallelism that can be attained with this organization is found as

$$p_{\max} = \left\lceil \frac{e+1}{2} \right\rceil .$$

It is easy to see from Figure 6-2 that $p_{\max} = 3$. When less than $p_{\max}$ units are available, the total execution time will increase, but it is still possible to perform the full precision computation with the smaller circuit.

Recall the modulus M must be a k-bit number. Also, r is 2^k and gcd(r, n) = gcd(2^k, n) = 1 (see section 4.1.1.4).

6.2. Simulation Results

Firstly, the processing element was coded and simulated. This is the basic block to build our pipeline (see figure 4.3). Later, we designed the pipeline and tested it.

The processing element performs operations from lines 3 to 14 in the MWR2MM algorithm. It computes only one interaction of the for-loop. Notice that we have two conditions: one condition determines if $S_0^{(0)} = 1$ (line 4) and another one decides if Y must be added to the partial result *(C, S)*. So, our test bench must be carefully chosen to certify it will execute all the lines of the algorithm. Next lines describe our test bench:

- **Test 1:**

It executes lines 3, 4, 11, 12, 13 and 14. It adds Y to the partial result as well (can be thought as an if statement).

x = 1

S = 255 = 00 11 11 11 11 Y = 109 = 00 01 10 11 01 M = 53 = 00 00 11 01 01

(C, S) = 182 = 00 10 11 01 10

- **Test 2:**

It executes lines 3, 4, 5, 6, 7, 8 and 9. It adds Y as well.

x = 1

S = 101 = 00 01 10 01 01 Y = 48 = 00 00 11 00 00 M = 56 = 00 00 11 10 00

(C, S) = 102 = 00 01 10 01 10

- **Test 3:**

It executes the same lines of test 1. It doesn't add Y.

x = 0

S = 90 = 00 01 01 10 10 Y = 91 = 00 01 01 10 11 M = 87 = 00 01 01 01 11

(C, S) = 45 = 00 00 10 11 01

- **Test 4:**

It executes the same lines of test 2. It doesn't add *Y*.

x = 0

S = 121 = 00 01 11 10 01 Y = 253 = 00 11 11 11 01 M = 52 = 00 00 11 01 00

(C, S) = 86 = 00 01 01 01 10

These results were compute by hand.

We simulated the VHDL source code in ModelSim and we got the same answers (see figure 6-3 and 6-5). We chose 2-bit processing element word-size (*w*) and 8-bit operand inputs (*n*). The order of *Sout* is reversed (from the least to the most significant word). The partial answer is 10-bit wide as explained in section 6.1.1

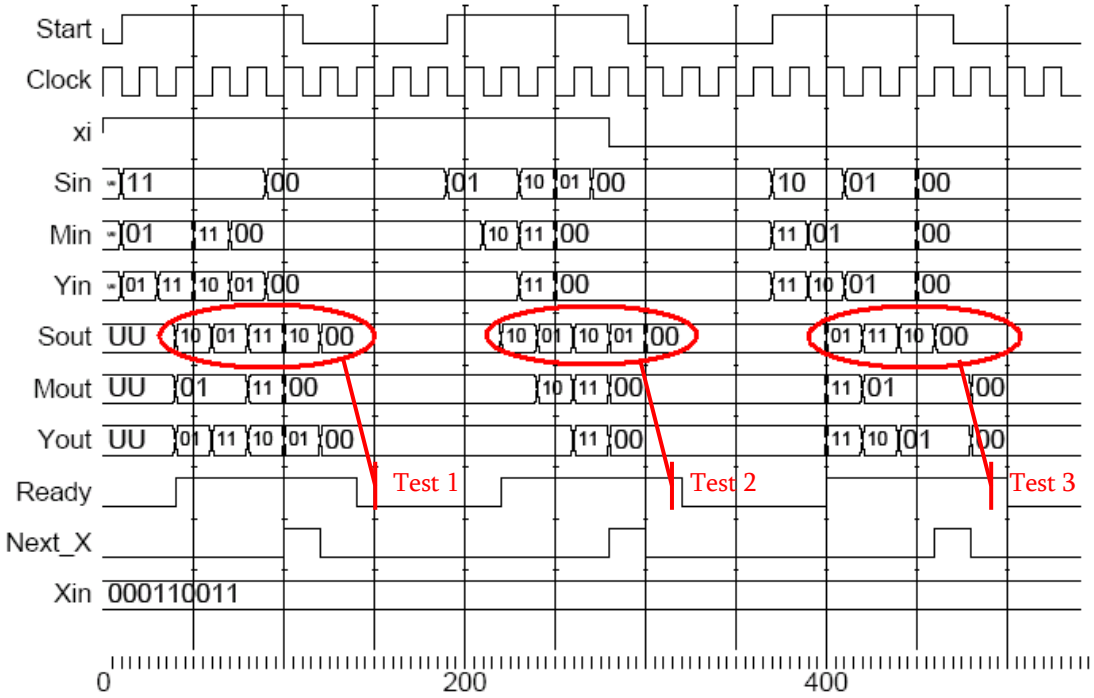


Figure 6-3: Processing element ModelSim simulation for w=2 and n=8

Data is valid only when ready is high. Notice that we need to wait one clock cycle until the first output is ready. Also, the processing element is still computing when start is low (one more clock cycle).

We also developed simple a program to perform tests. The same answers were achieved as showed in figure 6-4. The output is in decimal form and it has the same order of figure 6-3.

```
C:\Documents and Settings\Owner\My Documents\INPG\PFE\Documentos\C++>MWR2MM
x: 1                    x: 1                    x: 0

Si: 255                 Si: 101                 Si: 90

M: 53                   M: 56                   M: 87

Y: 109                  Y: 48                   Y: 91

S0[0]: 2                S0[0]: 2                S0[0]: 1
S0[1]: 1                S0[1]: 1                S0[1]: 3
S0[2]: 3                S0[2]: 2                S0[2]: 2
S0[3]: 2                S0[3]: 1                S0[3]: 0
S0[4]: 0                S0[4]: 0                S0[4]: 0
Press any key to continue . . .
```

Figure 6-4: Processing Element test software for w=2 and n=8

We also tested inputs when they have maximum values. Both methods (ModelSim simulation and test software) got the same results. See figures 6-5 and 6-6.

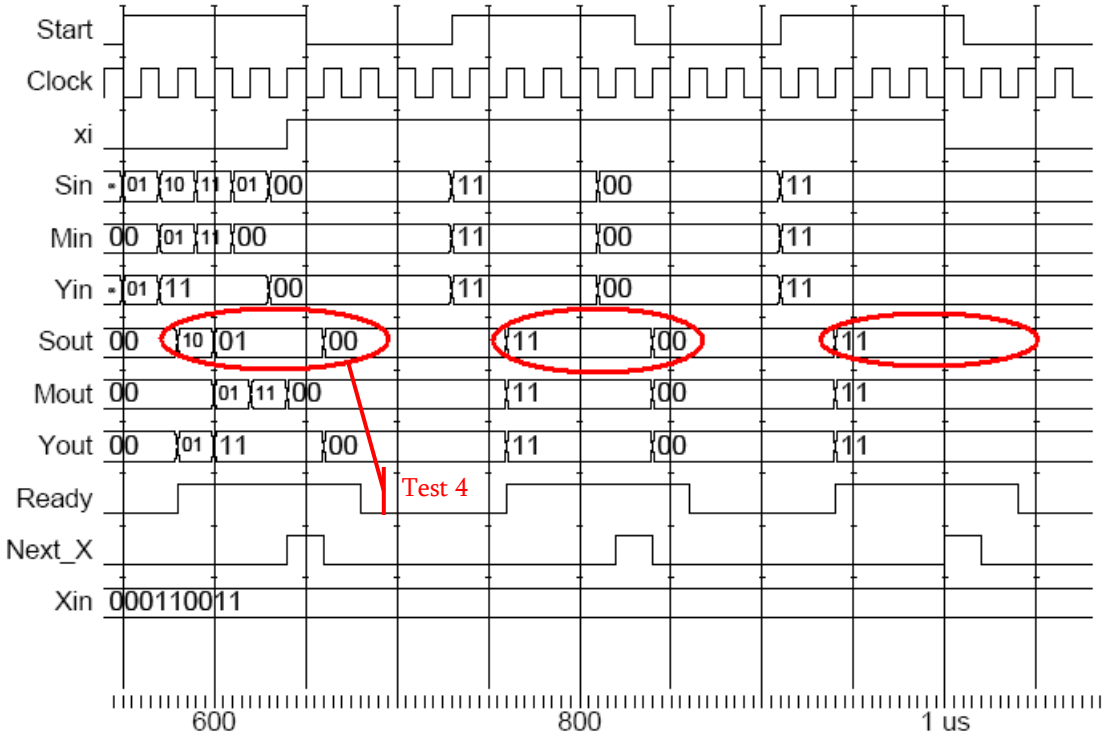


Figure 6-5: Processing element ModelSim simulation for w=2 and n=8

```
C:\Documents and Settings\Owner\My Documents\INPG\PFE\Documentos\C++>MWR2MM
x: 0                    x: 1                    x: 1

Si: 121                 Si: 255                 Si: 1023

M: 52                   M: 255                  M: 1023

Y: 253                  Y: 255                  Y: 1023

S0[0]: 2                S0[0]: 3                S0[0]: 3
S0[1]: 1                S0[1]: 3                S0[1]: 3
S0[2]: 1                S0[2]: 3                S0[2]: 3
S0[3]: 1                S0[3]: 3                S0[3]: 3
S0[4]: 0                S0[4]: 0                S0[4]: 3
Press any key to continue . . .
```

Figure 6-6: Processing Element test software for w=2 and n=8

We also did many others different random tests by using both the software and ModelSim. We got always the same answers. Thus, we prove the processing element is functional. Now, let's analyze and test the implementation of the Montgomery pipeline.

Since the Montgomery pipeline is scalable we will perform tests over its parameter (w and n) with the same input vectors. By doing this we will assure the pipeline is functional. The following test bench will be used. All the lines of the MWR2MM algorithm are covered.

- **Test 1:**

w =2 n = 8 e = 5 p = 3

X = 10 Y = 20 M = 247 r = 256

$S = XY\ r^{-1} \bmod M = 132$

- **Test 2:**

w =1 n = 8 e = 9 p = 5

X = 10 Y = 20 M = 247 r = 256

$S = XY\ r^{-1} \bmod M = 132$

- **Test 3:**

w =4 n = 16 e = 5 p = 3

X = 10 Y = 20 M = 60001 r = 65536

$S = XY\ r^{-1} \bmod M = 58646$

- **Test 4:**

w =1 n = 16 e = 17 p = 9

X = 10 Y = 20 M = 60001 r = 65536

$S = XY\ r^{-1} \bmod M = 58646$

These results were compute by hand.

Figure 6-7 and 6-8 show the results obtained by test 1. The number of processing elements generate in figure 6-7 matches with the result obtained (p = 3).

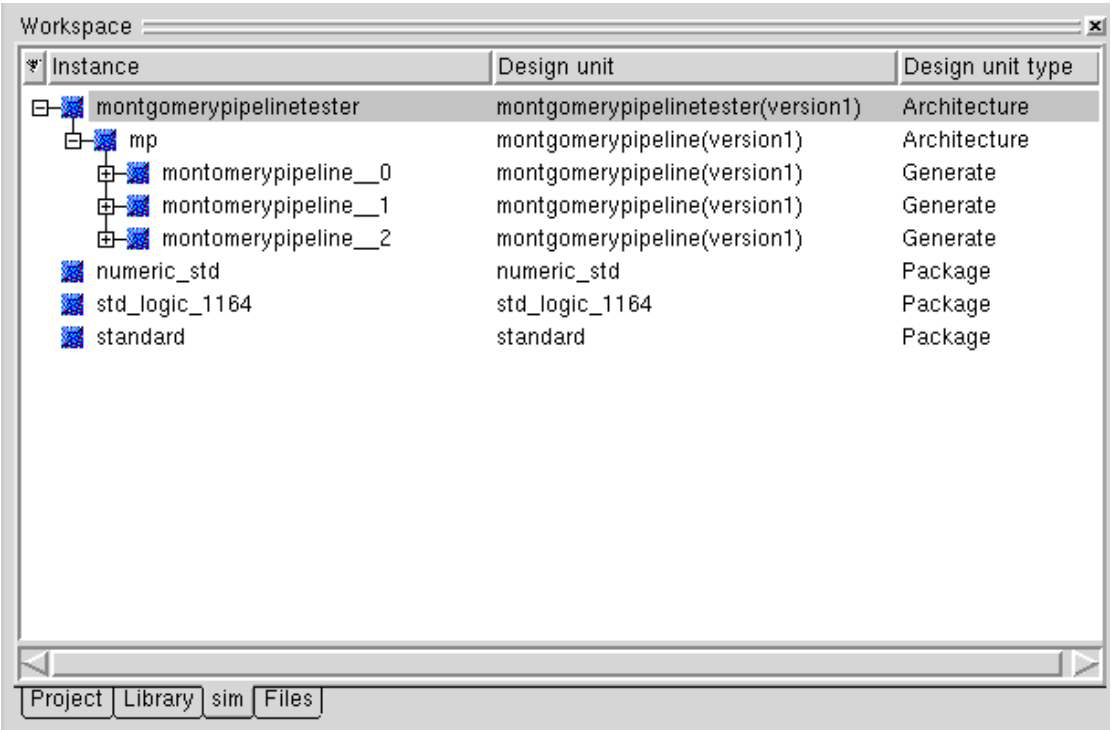


Figure 6-7: Generated architecture for w=2 and n=8 (Test 1)

The final result (*Sout*) is ready only in the last clock cycle of the signal Ready. Since we have a shift register in the output of the pipeline, we need to wait each partial result be transferred to the output register.

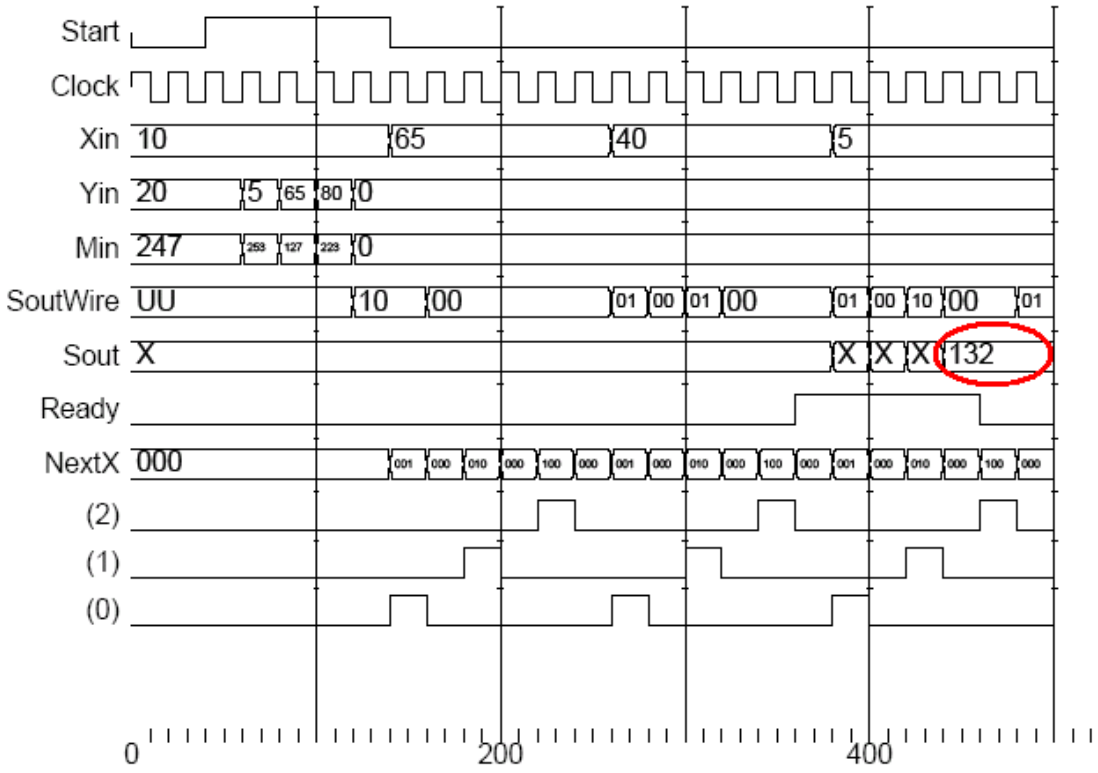


Figure 6-8: Montgomery Multiplicator ModelSim simulation for w=2 and n=8 (Test 1)

Next figures (6-9 and 6-10) are related to test 2. The same output is obtained. However, the word-size now is changed.

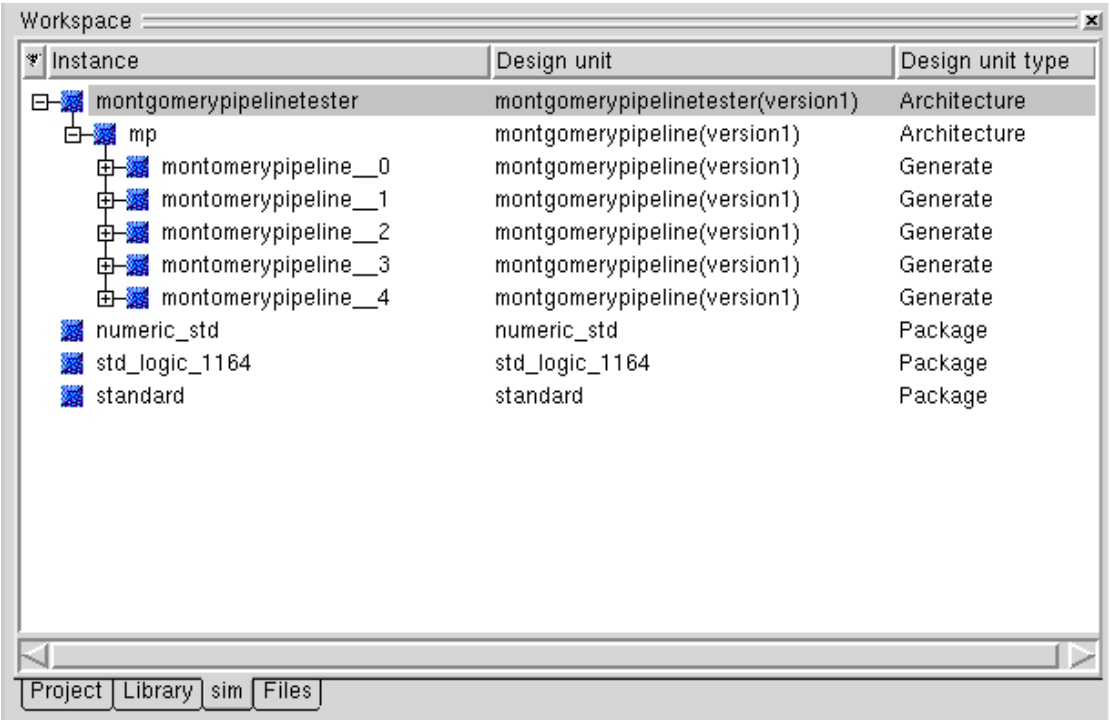


Figure 6-9: Generated architecture for w=1 and n=8 (Test 2)

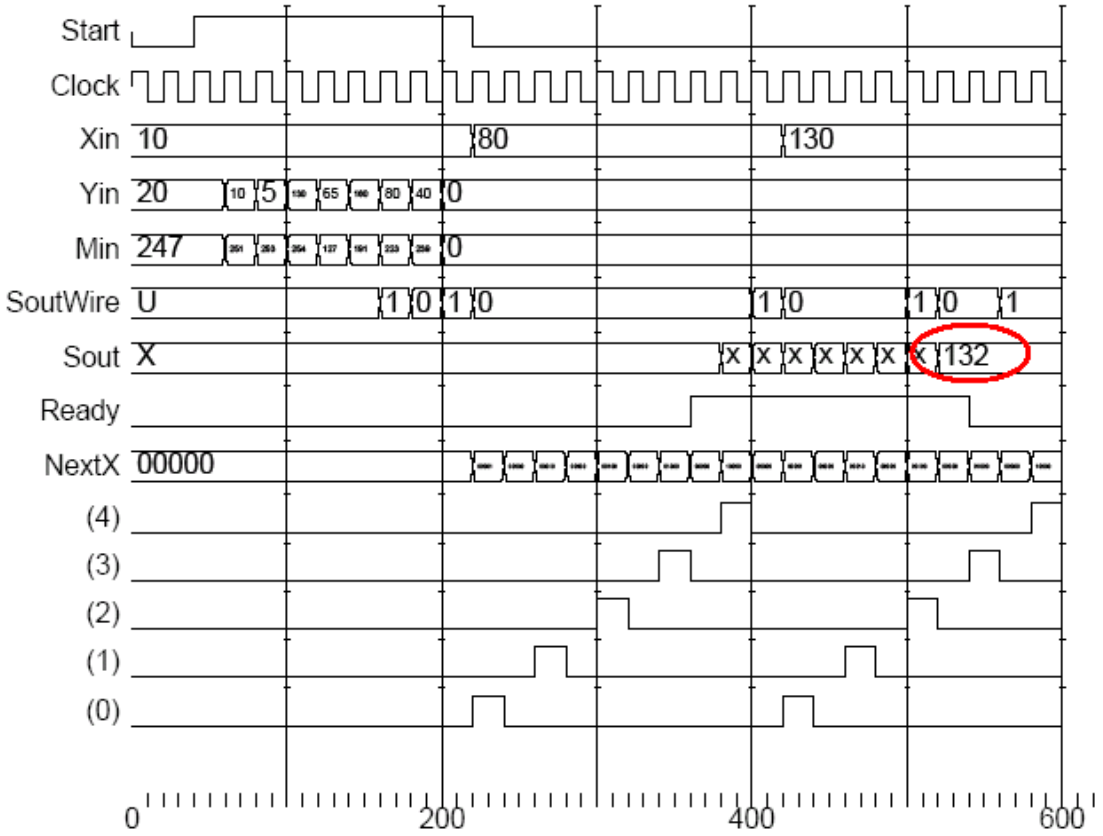


Figure 6-10: Multiplicator ModelSim simulation for w=1 and n=8 (Test 2)

Notice that when smaller word sizes are used more processing elements are necessary. Next examples we will increase the operand sizes of the pipeline (16 bits).

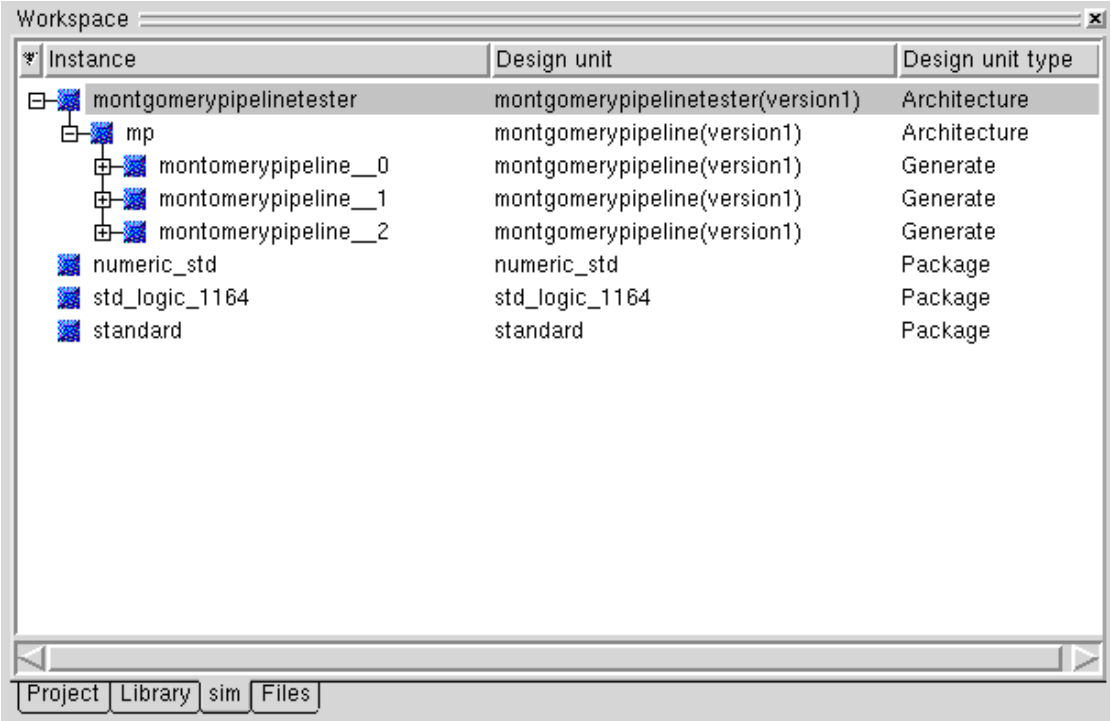


Figure 6-11: Generated architecture for w=4 and n=16 (Test 3)

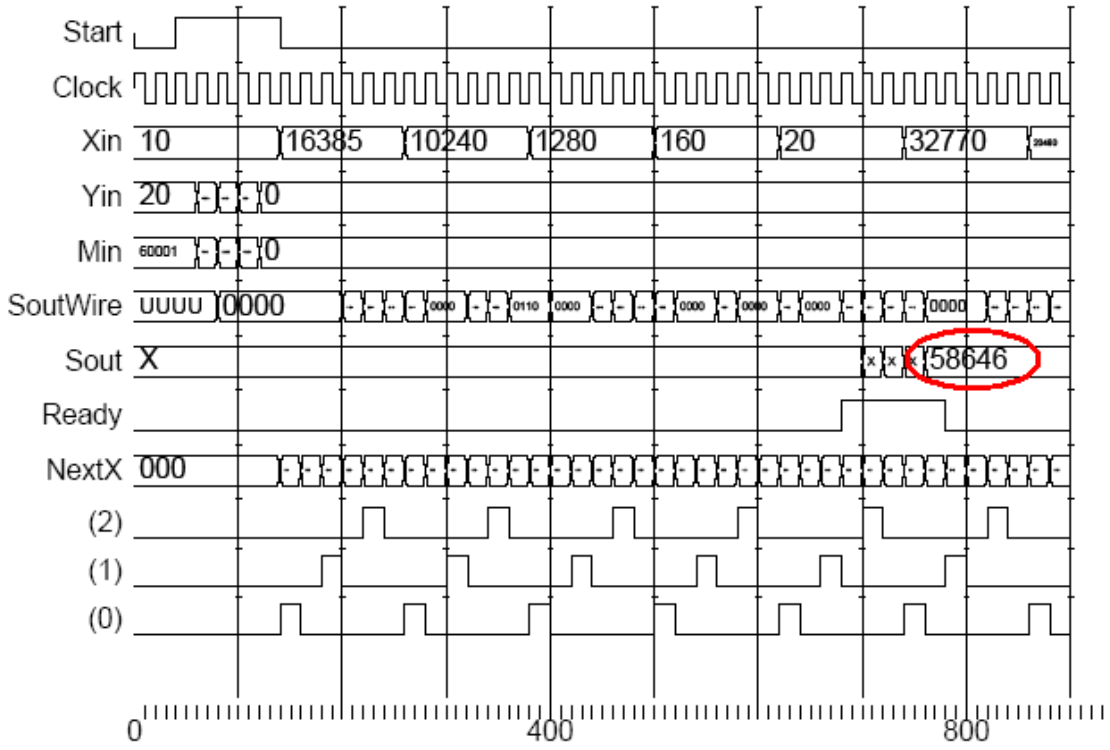


Figure 6-12: Multiplicator ModelSim simulation for w=4 and n=16 (Test 3)

Recall that the algorithm works only when $2^{k-1} < M < 2^k$. This explains why we can't use the previous values of M for result comparison.

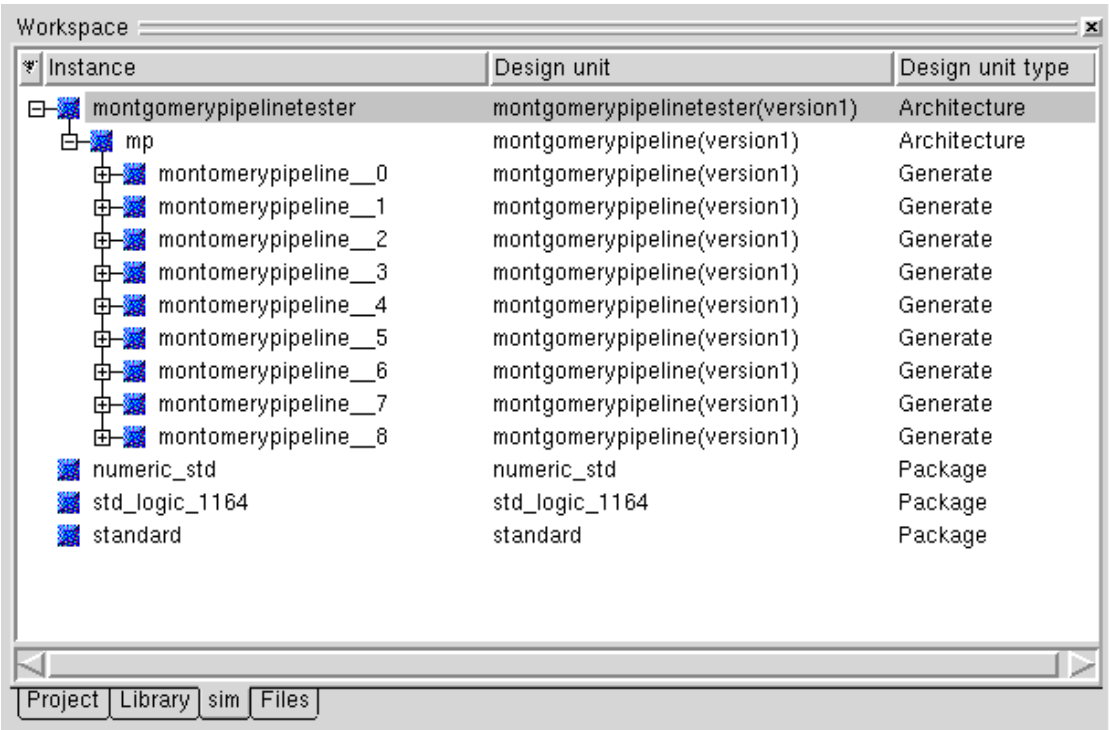


Figure 6-13: Generated architecture for w=4 and n=16 (Test 4)

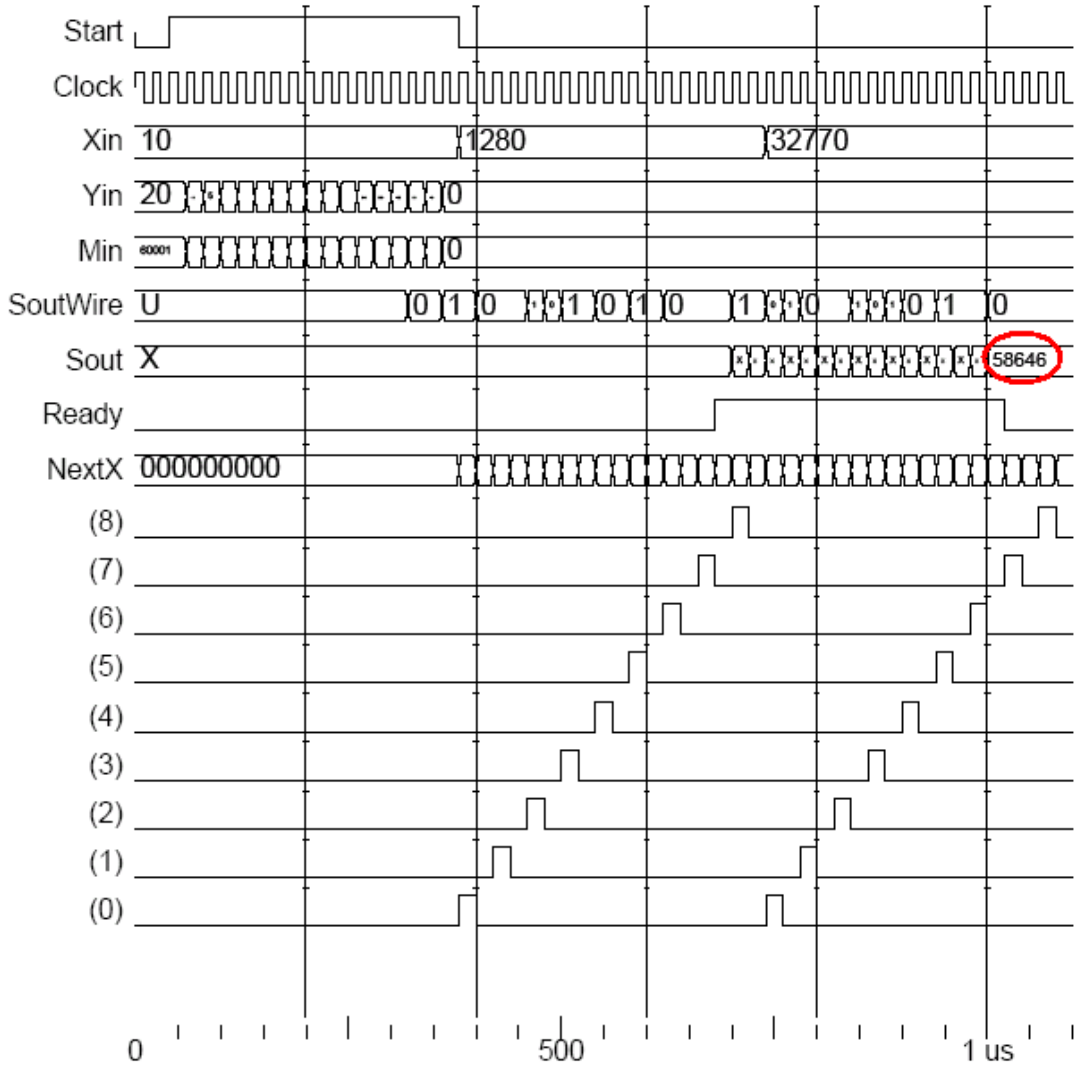


Figure 6-14: Multiplicator ModelSim simulation for w=1 and n=16 (Test 4)

Also, the following tests were performed. In all of them we obtained the expected output.

- w =2, n = 8, e = 5, p = 3 and r = 256

X = 11, Y = 20, M = 247 S = XY r^{-1} mod M = 244

X = 12, Y = 20, M = 247 S = XY r^{-1} mod M = 109

X = 13, Y = 20, M = 247 S = XY r^{-1} mod M = 221

X = 10, Y = 21, M = 247 S = XY r^{-1} mod M = 188

X = 10, Y = 22, M = 247 S = XY r^{-1} mod M = 244

X = 10, Y = 23, M = 247 S = XY r^{-1} mod M = 53

X = 10, Y = 20, M = 249 S = XY r^{-1} mod M = 242

X = 1o, Y = 20, M = 251 S = XY r^{-1} mod M = 40

X = 11, Y = 20, M = 253 S = XY r^{-1} mod M = 151

Thus, we prove the Montgomery pipeline is functional.

6.3. Synthesis Results

We synthesized the Montgomery Pipeline in Leonardo. The target FPGA is Xilinx Virtex-II Pro (Device 2VP7fg456). By changing the parameters in the VHDL code we performed the following experimentations:

- Synthesis 1

n = 1024 w = 32 e = 33 p = 17

- Synthesis 2

n = 1024 w = 64 e = 17 p = 9

By doing this we obtained f_1 =80.9 MHz and f_2 =63.2 MHz for synthesis 1 and 2 respectively.

We can also estimate the encryption/decryption rate of the complete circuit (exponentiator). The equation below was extracted from [26]. It means the total computation time T (in clock cycles) of the Montgomery Pipeline.

$$T = \left\lceil \frac{n+1}{p} \right\rceil (e+1) - 1 + 2(p-1)$$

Thus, by solving equation above, we have:

Synthesis 1: $T_1 = 2105$

Synthesis 2: $T_2 = 2067$

Assuming we have an average number of multiplications (section 4.1.1) and an exponent of 1024 bits, we have $T_1 \cdot 3/2\,(k-1) = 3230122{,}5$ and $T_2 \cdot 3/2\,(k-1) = 3171811{,}5$.

These are the total number of cycles require for encrypting 1024 bits. Therefore, we can obtain the encryption rate by

$$R_1 = \frac{f_1}{T_1} \cdot k\,bits = 25{,}05\,kbits/s$$

$$R_2 = \frac{f_2}{T_2} \cdot k\,bits = 19{,}92\,kbits/s\,.$$

7. Conclusion

The present work showed us all the steps to design a Montgomery multiplication unit adjustable to work with large key sizes. Due to its pipelined characteristics there is no limitation on the maximum number of bits manipulated, i.e., the Montgomery multiplication algorithm core is ready for future changes. Also, we have specified a complete RSA system.

The total time to compute a Montgomery multiplication depends on the available area and the pipeline configuration. Our final tests with Leonardo showed us that a multi-stage pipeline is faster than a single unit working with a large word length. This interesting result demostrates that more processing elements units decrease the amount of time to encrypt data. So, the desired performance is made according to the available chip area.

Because of time constraints, only the Montgomery multiplication unit was finished. This is the core of the modular exponentiation unit, and the core of the RSA coprocessor. Since good RSA architectures are difficult to design most of the time was spent in research. It should be also kept in mind that is not an easy task to implement a 1024-bit architecture or higher. However, we assure that anyone with a background of algorithms, digital logic design and computer architecture will be able to read this document and understand it quickly. Thus, by virtue of its contents anyone who wants to continue this project has good references and information to study.

8. Bibliography

[1] **RSA Laboratories**, RSA Laboratories' Frequently Asked Questions About Today's Cryptography, Version 4.1, RSA Security Inc., 2000.

[2] **Bauer F.L.**, Decrypted Secrets: Methods and Maxims of Cryptology, third edition, Springer, 2002.

[3] **Diffie W, Hellman M. E.**, New directions in cryptography (pp 644-654), IEEE Transactions on Information Theory 22, 1976.

[4] **R.L. Rivest, A. Shamir, L.M. Adleman**, A method for obtaining digital signatures and public-key cryptosystems (pp 120-126), Communications of the ACM (2) 21, 1978.

[5] http://mathworld.wolfram.com, concepts of math, March, 2004.

[6] **Ç. K. Koç**, RSA Hardware Implementation, RSA Laboratories Technical Report TR-801, 1996.

[7] **Ç. K. Koç**, High-Speed RSA Implementation, RSA Laboratories, 1994.

[8] http://www.rsasecurity.com/rsalabs/node.asp?id=2004, RSA secures key-sizes, March, 2004.

[9] http://www.rsascurity.com/rsalabas/challenges/, RSA factorization challenge, March, 2004.

[10] **T. Matthews**, RSA Laboratories Bulletin, RSA Laboratories, 1996.

[11] http://smartrust.com, history of cryptography, April, 2004.

[12] http://cybercrimes.net/Cryptography/Articles/Hebert.html, history of cryptography, April, 2004.

[13] http://www.rsasecurity.com/, cryptosystems, commercial solutions, applications, March of 2004.

[14] **D. E. Knuth**, The Art of Computer Programming: Seminumerical Algorithms, volume 2, third edition, Addison-Wesley, 1997.

[15] **C. H Wu, J. H. Hong, C. W. Wu**, RSA cryptosystem design based on the Chinese remainder theorem, Proceedings of the 2001 conference on Asia South Pacific design automation, January, 2001.

[16] **A. Daly, W. Marnane**, Efficient architectures for implementing montgomery modular multiplication and RSA modular exponentiation on reconfigurable logic, Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays, February, 2002.

[17] **H. Nozaki, M. Motiyama, A. Shimbo, S. Kawamura**, Implementation of RSA Algorithm Based on RNS Montgomery Multiplication, Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg, Volume 2162, 2001.

[18] **P. L. Montgomery**. Modular multiplication without trial division, Mathematics of Computation, 44(170):519-521, April 1985.

[19] **C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, J.-P. Seifert,** Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures, Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg, Volume 2523, 2003.

[20] **S.-M. Yen, S. Kim, S. Lim, S. Moon.,** RSA Speedup with Residue Number System Immune against Hardware Fault Cryptanalysis, Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg, Volume 2523, 2002.

[21] **S.-M. Yen, S. Moon, J. Ha,** Permanent Fault Attack on the Parameters of RSA with CRT, Lecture Notes in Computer Science, Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg, Volume 2523, 2003.

[22] **S.-M. Yen, S. Moon, J. Ha,** Hardware Fault Attack on RSA with CRT Revisited*, Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg, Volume 2587, 2003.

[23] **S.-M. Yen, S. Kim, S. Lim, S.-J. Moon,** RSA Speedup with Chinese Remainder Theorem Immune against Hardware Fault Cryptanalysis, Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg, Volume 2228, 2002.

[24] **J. Blömer, M. Otto, J.-P. Seifert,** A New CRT-RSA Algorithm Secure Against Bellcore Attacks,Conference on Computer and Communications Security, pages 311-320, 2003.

[25] http://people.atips.ca/~rahmanc/619.88/, RNS tutorial, May, 2004.

[26] **Ç. K. Koç, C. Paar,** A Scalable Architecture for Montgomery Multiplication, CHES'99, LNCS 1717, pp. 94-108, 1999.

[27] **O. Goldreich, S. Goldwasser, S. Micali,** How to construct random functions, Journal of the ACM (JACM), Volume 33, Issue 4, August, 1986.

[28] **A. Shamir,** On the generation of cryptographically strong pseudorandom sequences , ACM Transactions on Computer Systems (TOCS), Volume 1, Issue 1, February, 1983.

[29] **J. P. Hayes,** Computer Architecture and Organization, third edition, McGraw-Hill, 1998.

[30] **S. Brown, Z. Vranesic,** Fundamentals of Digital Logic with VHDL Design, McGraw-Hill, 2000.

UFRGS
UNIVERSIDADE FEDERAL
DO RIO GRANDE DO SUL

Design of a scalable RSA coprocessor

Alcides Silveira Costa
André Inácio Reis
Renato Perez Ribas

Introduction

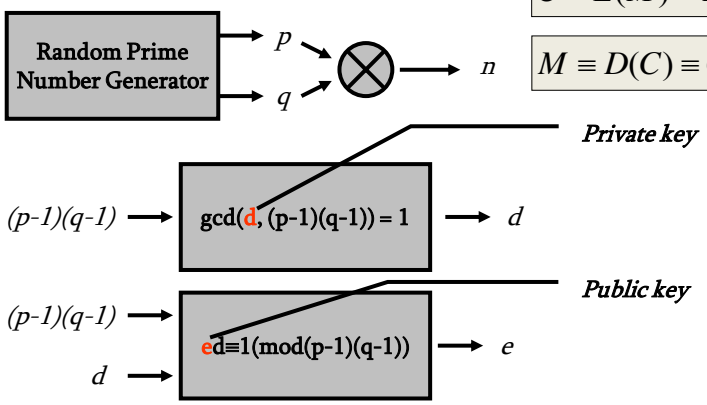
- The RSA Cryptosystem
- Design Methods
- Coprocessor Design
- Simulation Results
- Synthesis Results
- New applications & Future Works
- Conclusion

The RSA Cryptosystem

- The RSA algorithm

$$C \equiv E(M) \equiv M^e \pmod{n}$$

$$M \equiv D(C) \equiv C^d \pmod{n}$$



Design Methods

- CRT-based, RNS-based and Pipelined-based architectures

Protection Lifetime of Data	*Present 2010*	*Present 2030*	*Present 2031 and beyond*
Minimum RSA key size	1024 bits	2048 bits	3072 bits

http://www.rsasecurity.com/rsalabs/technotes/twirl.html

Design Methods

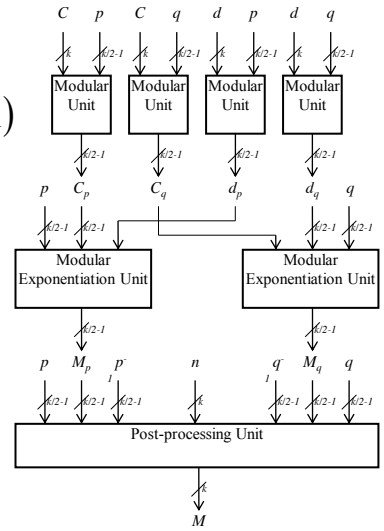
- CRT based Architecture

$$C_p = C(\mathrm{mod}\, p), d_p = d(\mathrm{mod}\, p-1)$$

$$C_q = C(\mathrm{mod}\, q), d_q = d(\mathrm{mod}\, q-1)$$

$$M_p = C_p^{d_p}(\mathrm{mod}\, p) \quad M_q = C_q^{d_q}(\mathrm{mod}\, q)$$

$$M = \begin{pmatrix} M_p\left(q^{-1}(\mathrm{mod}\, p)\right)q \\ +M_q\left(p^{-1}(\mathrm{mod}\, q)\right)p \end{pmatrix}(\mathrm{mod}\, n)$$

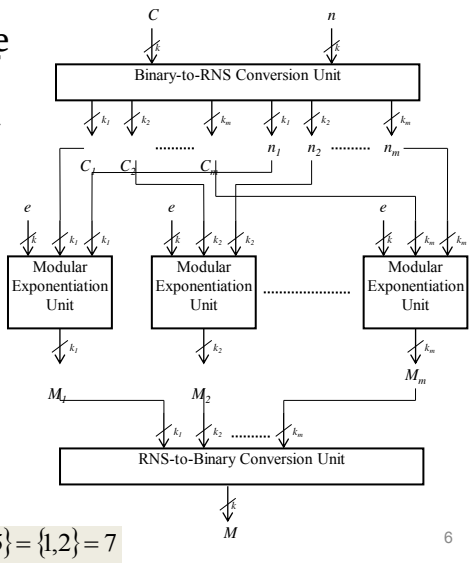


Design Methods

- RNS based architecture

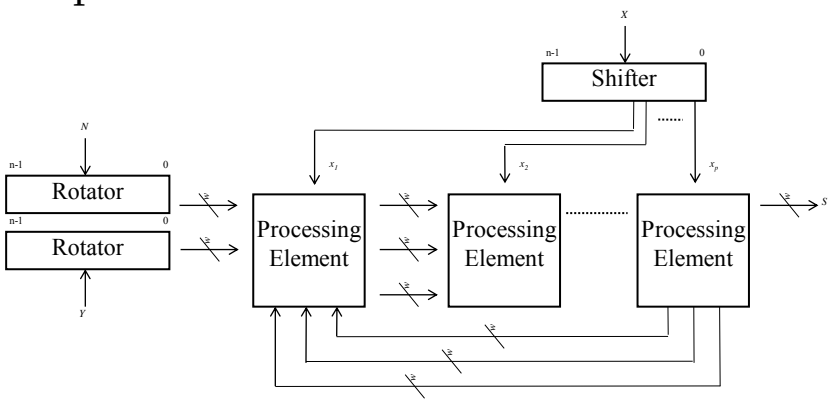
Signed Integer Numbers	*Unsigned Integer Numbers*	*Mod 3*	*Mod 5*
0	**0**	0	0
1	**1**	1	1
2	**2**	2	2
3	**3**	0	3
4	**4**	1	4
5	**5**	2	0
6	**6**	0	1
7	**7**	1	2
-7	**8**	2	3
-6	**9**	0	4
-5	**10**	1	0
-4	**11**	2	1
-3	**12**	0	2
-2	**13**	1	3

$$4+3=\{1,4\}+\{0,3\}=\{1+0\,\mathrm{mod}\,3, 4+3\,\mathrm{mod}\,5\}=\{1,2\}=7$$



Design Methods

- Pipelined based architecture



7

Design Methods

- CRT-based architectures
  - Vulnerable (Fault Attacks)
  - Fast

$$\gcd\left(\left(C-(M')^{e}\right)\bmod n, n\right)=q$$

- RNS-based architectures
  - Suitable when operand sizes are large
  - Complex
- Pipelined-based architectures
  - Secure
  - Achieve good results

8

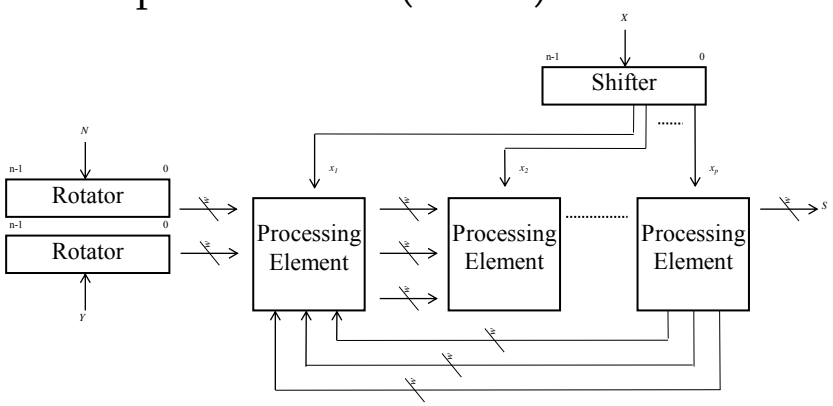
Design Methods

- Montgomery Exponentiation
  Entradas: M, e, n e k
  Saída: $C = M^e \bmod n$ $\quad k = 2^{2m} \bmod n$

```
1.  M' = MonPro(M, k)
2.  x' = MonPro(1, k)
3.  for i = k-1 down to 0 do
4.          x' = MonPro(x', x')
5.          if e_i = 1 then x' = MonPro(M', x')
6.  C = MonPro(x', 1)
7.  return M
```

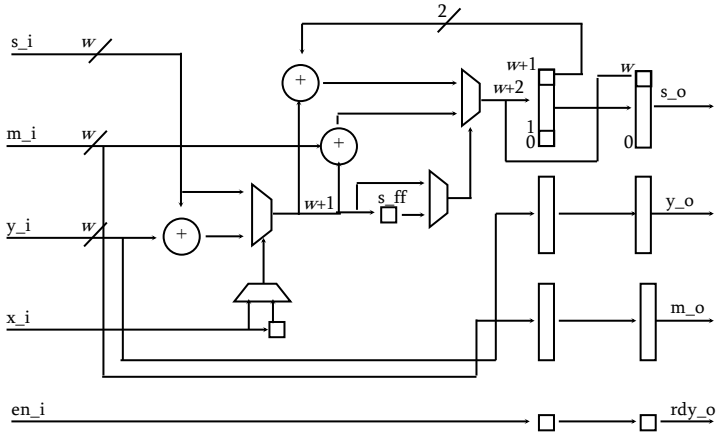

Coprocessor Design

- Multiplication Core (France)



Coprocessor Design

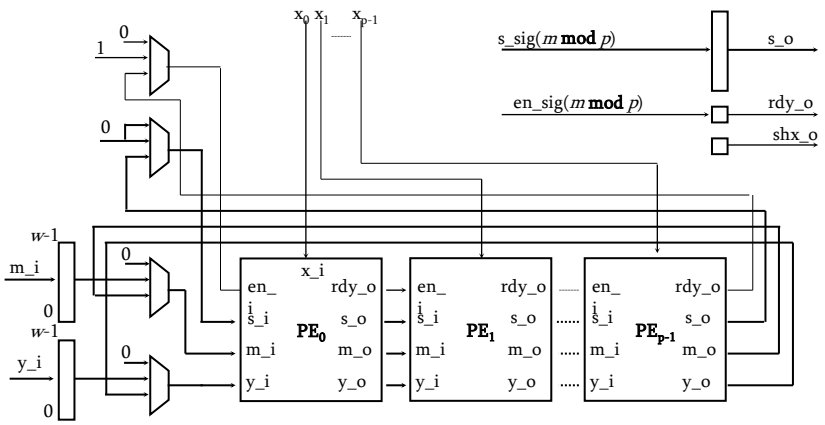
- Processing Element (PE)



11

Coprocessor Design

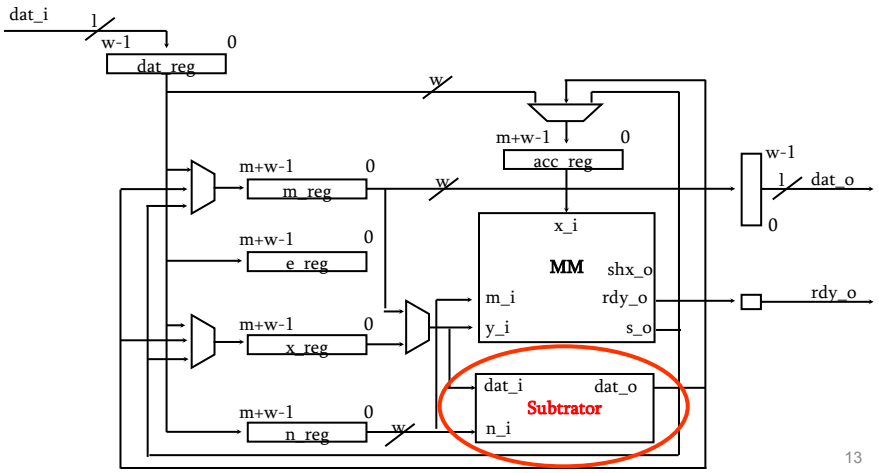
- Montgomery Multiplication Unit (MM)



12

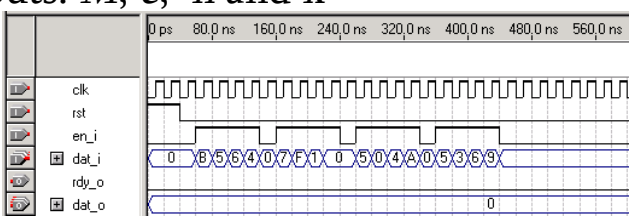
Coprocessor Design

- Modular Exponentiation Unit (Brazil)

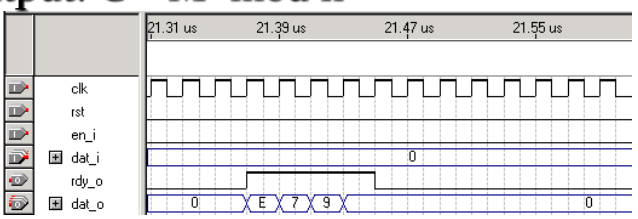


Simulation Results

- Inputs: *M*, *e*, *n* and *k*



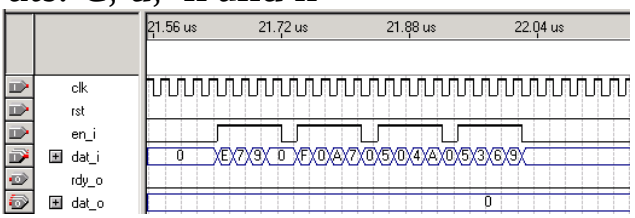
- **Output: $C = M^e \bmod n$**



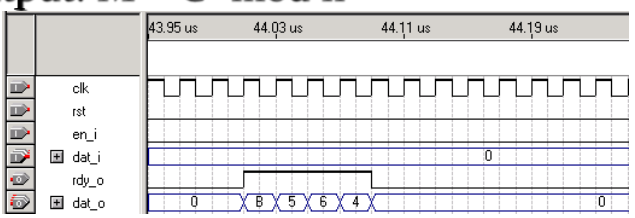
$C = 18011^{503} \bmod 41989 = 2430$

Simulation Results

- Inputs: C, d, n and k



- **Output: $M = C^d \bmod n$**



$M = 2430^{31247} \bmod 41989 = 18011$

15

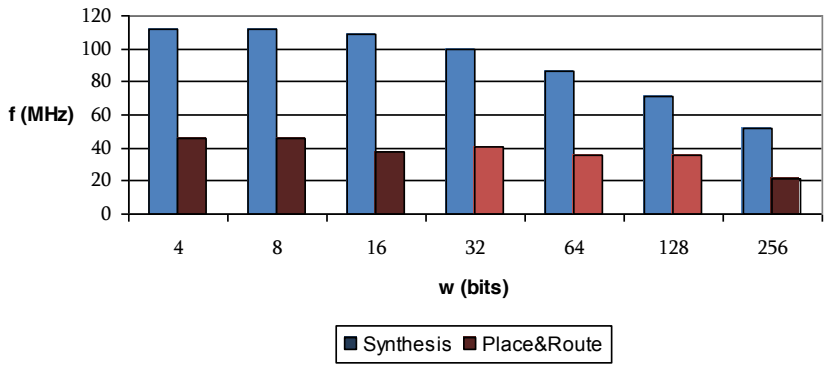
Synthesis Results

- Synthesis Tool
  - Xilinx XST (ISE)
- Target device
  - Xilinx Virtex Pro II (xc2vp100-ff1696-6)
- One-hot enconding

16

Synthesis Results

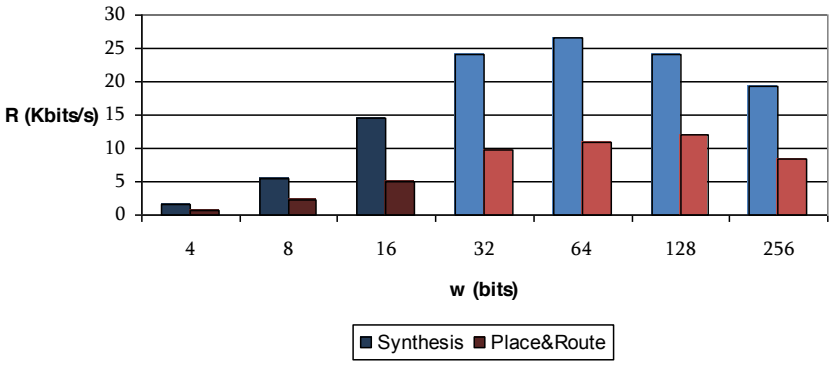
Data bus size: 1 bit
Key size: 1024 bits



17

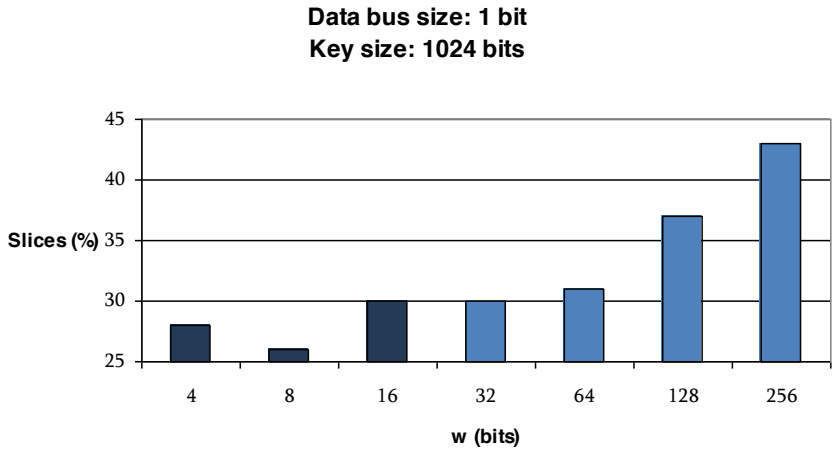
Synthesis Results

Data bus size: 1 bit
Key size: 1024 bits



18

Synthesis Results



19

New applications & Future work

- New applications
  - Digital TV
  - Cell phones
  - Smart Cards
- Future work
  - Best-case Architecture
  - Usage of memories can improve performance
  - Outside data bus < internal data bus

20

Conclusions

- A Scalable RSA coprocessor
- Fast RSA systems are a challenge
- Increasing research field

UFRGS
UNIVERSIDADE FEDERAL
DO RIO GRANDE DO SUL

Design of a scalable RSA cooprocessor

Alcides Silveira Costa
cids@inf.ufrgs.br

```
--------------------------------------------------------------------------------
-- Autor   : Alcides Silveira Costa
-- Bloco   : RSA (Topo + Controle)
--------------------------------------------------------------------------------
library ieee;
   use ieee.std_logic_1164.all;
   use ieee.std_logic_unsigned.all;
   use ieee.std_logic_arith.all;

entity rsa is
   generic(
      n  : positive := 1024;
      w  : positive := 2;
      l  : positive := 1;
      e  : positive := 513;
      p  : positive := 257);
   port(
      -- Entradas
      clk   : in std_logic;
      rst   : in std_logic;
      en_i  : in std_logic;
      dat_i : in std_logic_vector(l-1 downto 0);

      -- Saidas
      dat_o : out std_logic_vector(l-1 downto 0);
      rdy_o : out std_logic);
end rsa;

architecture rtl of rsa is
-- Tipos definidos
type shift_reg1 is array (w/l-1 downto 0) of std_logic_vector(l-1 downto 0);
type shift_reg2 is array ((n+w)/w-1 downto 0) of std_logic_vector(w-1 downto 0);

-- Componentes
component mp
   generic(
      n  : positive := 16;
      w  : positive := 4;
      e  : positive := 5;
      p  : positive := 3);
   port(
      clk   : in std_logic;
      rst   : in std_logic;
      en_i  : in std_logic;
      m_i   : in std_logic_vector(w-1 downto 0);
      y_i   : in std_logic_vector(w-1 downto 0);
      x_i   : in std_logic_vector(p-1 downto 0);
      rdy_o : out std_logic;
      shx_o : out std_logic;
      s_o   : out std_logic_vector(w-1 downto 0));
end component;

component sub
   generic(w  : positive := 1);
   port(
      clk   : in std_logic;
      rst   : in std_logic;
      en_i  : in std_logic;
      a_i   : in std_logic_vector(w-1 downto 0);
      n_i   : in std_logic_vector(w-1 downto 0);
      s_o   : out std_logic_vector(w-1 downto 0));
end component;

-- Funções
```

```
function log2(x: integer) return integer is
   -- Funciona apenas para numero base 2
   variable result : integer := 1;
   variable aux  : integer;
   begin
      if x /= 1 then
         aux := x;
         while aux /= 2 loop
            aux := aux/2;
            result := result + 1;
         end loop;
      else
         result := 0;
      end if;
   return result;
end log2;

function array2vector(x: shift_reg2) return std_logic_vector is
   variable result : std_logic_vector((x'left+1)*(x(0)'left+1)-1 downto 0);
   begin
      for i in x'left downto 0 loop
         result((x(0)'left+1)*(i+1)-1 downto (x(0)'left+1)*i) := x(i);
      end loop;
   return result;
end array2vector;

function vector2array(x: std_logic_vector) return shift_reg2 is
   variable result : shift_reg2;
   begin
      for i in result'left downto 0 loop
         result(i) := x((result(0)'left+1)*(i+1)-1 downto (result(0)'left+1)*i);
      end loop;
   return result;
end vector2array;

function set_array(y, z: positive) return shift_reg2 is
   variable result : shift_reg2;
   variable aux  : std_logic_vector(w-1 downto 0);

   begin
      aux := (others => '0');
      for i in y/z-1 downto 0 loop
         result(i) := aux;
      end loop;
   result(0)(0) := '1';
   return result;
end set_array;

function special_shift(
   y, z: positive;
   a  : shift_reg2) return shift_reg2 is

   variable aux  : std_logic_vector(y-1 downto 0);
   variable aux2 : std_logic_vector(z-1 downto 0);
   variable aux3 : std_logic_vector(z-1 downto 0);
   variable result  : shift_reg2;

begin
   aux   := array2vector(a);
   aux3:= (others => '0');

   if y mod z = 0 then
      -- Registrador homemgeneo
      for i in 1 to n/p-1 loop
```

```
         aux(z*i-1 downto z*(i-1)):= aux(z*(i+1)-1 downto z*i);
      end loop;
   else
      -- Registrador hetereogeneo
      if y/z = 1 then
         -- Apenas um shift
         aux(z-1 downto 0):= aux3&aux(y-1 downto z);
      else
         -- Varios shift
         for i in 1 to y/z loop
            if i < y/z then
               aux(z*i-1 downto z*(i-1)):= aux(z*(i+1)-1 downto z*i);
            else
               aux2(((y-1) mod z) downto 0) := aux(y-1 downto z*i);
--             aux2(p-1 downto (n mod p)) := (others => '0');
               aux2(z-1 downto (y mod z)) := aux3(z-1 downto (y mod z));
               aux(z*i-1 downto z*(i-1)):= aux2;
            end if;
         end loop;
      end if;
   end if;
   result := vector2array(aux);
   return result;
end special_shift;

function shift_byte(
   y, z: positive;
   a  : std_logic_vector;
   b  : shift_reg1) return shift_reg1 is

   variable result  : shift_reg1;
begin
   result(y/z-1):=a;
   for i in y/z-1 downto 1 loop
      result(i-1):= b(i);
   end loop;
   return result;
end shift_byte;

function shift_byte(
   y, z: positive;
   a  : shift_reg1;
   b  : shift_reg2) return shift_reg2 is

   variable result  : shift_reg2;
   variable aux  : std_logic_vector(z-1 downto 0);
   variable l    : positive := a(0)'left + 1;

begin
   for i in w/l downto 1 loop
      aux(l*i-1 downto l*(i-1)):= a(i-1);
   end loop;

   result(y/z-1):= aux;
   for i in y/z-1 downto 1 loop
      result(i-1):= b(i);
   end loop;

   return result;
end shift_byte;

function shift_byte(
      y: positive;
      x: shift_reg2) return shift_reg2 is
```

```
   variable aux  : std_logic_vector((x'left+1)*(x(0)'left+1)-1 downto 0);
   variable zero : std_logic_vector(y-1 downto 0);
   variable result  : shift_reg2;
begin
   zero  := (others => '0');
   aux   := array2vector(x);
   aux      := zero & aux(aux'left downto y);
   result   := vector2array(aux);

   return result;
end shift_byte;

function shift_byte(
   y, z: positive;
   a  : std_logic_vector;
   b  : shift_reg2) return shift_reg2 is

   variable result  : shift_reg2;

begin
   result(y/z-1):= a;
   for i in y/z-1 downto 1 loop
      result(i-1):= b(i);
   end loop;

   return result;
end shift_byte;


function rotate_byte(
   y, z: positive;
   a  : shift_reg2) return shift_reg2 is

   variable result  : shift_reg2;
begin
   result(y/z-1):= a(0);

   for i in y/z-1 downto 1 loop
      result(i-1):= a(i);
   end loop;

   return result;
end rotate_byte;

-- FSM
type states is (m1_st, m2_st, e1_st, e2_st, n1_st, n2_st, k1_st, k2_st,
            op10_st, op11_st, op12_st, op13_st, op14_st,
            op21_st, op22_st, op23_st, op24_st, op25_st,
            op31_st, op32_st, op33_st, op34_st,
            op41_st, op42_st, op43_st, op44_st, op45_st,
            op51_st, op52_st, op53_st, op54_st, op55_st,
            op61_st);
signal current_state, next_state: states;

signal en_mp     : std_logic;
signal en_sub    : std_logic;
signal s_sub     : std_logic_vector(w-1 downto 0);
signal sel_fsm   : std_logic;

-- Registradores de deslocamento
signal dat_reg: shift_reg1;
signal m_reg  : shift_reg2;
signal e_reg  : shift_reg2;
signal n_reg  : shift_reg2;
```

```
signal acc_reg: shift_reg2;
signal x_reg  : shift_reg2;

signal s_reg  : std_logic_vector(l-1 downto 0);

-- Flip-flops
signal rdy_ff : std_logic;

-- Counters
signal a_cnt  : std_logic_vector(log2(n/l) downto 0);
signal b_cnt  : std_logic_vector(log2(n)-1 downto 0);

-- Sinais intertos
signal y_sig  : std_logic_vector(w-1 downto 0);
signal s_sig  : std_logic_vector(w-1 downto 0);
signal x_sig  : std_logic_vector(p-1 downto 0);
signal zero_sig  : std_logic_vector(w-1 downto 0);
signal shx_sig: std_logic;
signal rdy_sig: std_logic;
signal e_bit  : std_logic;
signal e_sig  : std_logic_vector(n+w-1 downto 0);

begin
e_sig <= array2vector(e_reg);
e_bit <= e_sig(conv_integer(unsigned(b_cnt)));

process (current_state, rst, en_i, a_cnt, b_cnt, rdy_sig, e_reg, m_reg, x_reg)
begin
   if (rst = '1') then
      sel_fsm  <= '0';
      en_mp    <= '0';
      en_sub     <= '0';
      next_state  <= m1_st;
   else
      case current_state is
         when m1_st =>
            sel_fsm    <= '0';
            en_mp      <= '0';
            en_sub        <= '0';
            if (en_i = '0') then
               next_state <= m1_st;
            else
               next_state <= m2_st;
            end if;

         when m2_st =>
            sel_fsm    <= '0';
            en_mp      <= '0';
            en_sub        <= '0';
            if (en_i = '1') then
               next_state <= m2_st;
            else
               next_state <= e1_st;
            end if;

         when e1_st =>
            sel_fsm    <= '0';
            en_mp      <= '0';
            en_sub        <= '0';
            if (en_i = '0') then
               next_state <= e1_st;
            else
               next_state <= e2_st;
            end if;
```

```
when e2_st =>
   sel_fsm    <= '0';
   en_mp      <= '0';
   en_sub        <= '0';
   if (en_i = '1') then
      next_state <= e2_st;
   else
      next_state <= n1_st;
   end if;


when n1_st =>
   sel_fsm    <= '0';
   en_mp      <= '0';
   en_sub        <= '0';
   if (en_i = '0') then
      next_state <= n1_st;
   else
      next_state <= n2_st;
   end if;

when n2_st =>
   sel_fsm    <= '0';
   en_mp      <= '0';
   en_sub        <= '0';
   if (en_i = '1') then
      next_state <= n2_st;
   else
      next_state <= k1_st;
   end if;

when k1_st =>
   sel_fsm    <= '0';
   en_mp      <= '0';
   en_sub        <= '0';
   if (en_i = '0') then
      next_state <= k1_st;
   else
      next_state <= k2_st;
   end if;

when k2_st =>
   sel_fsm    <= '0';
   en_mp      <= '0';
   en_sub        <= '0';
   if (en_i = '1') then
      next_state <= k2_st;
   else
      next_state <= op10_st;
   end if;

when op10_st =>
   sel_fsm    <= '0';
   en_mp      <= '0';
   en_sub        <= '0';
   next_state <= op11_st;

when op11_st =>
   sel_fsm    <= '1';
   en_mp      <= '1';
   en_sub        <= '0';
   if a_cnt /= n/w then
      next_state <= op11_st;
```

```
   else
      next_state <= op12_st;
   end if;

when op12_st =>
   sel_fsm    <= '1';
   en_mp      <= '0';
   en_sub        <= '0';
   if rdy_sig = '0' then
      next_state <= op12_st;
   else
      next_state <= op13_st;
   end if;

when op13_st =>
   sel_fsm    <= '1';
   en_mp      <= '0';
   en_sub        <= '0';
   if rdy_sig = '1' then
      next_state <= op13_st;
   else
      if m_reg(n/w) = 0 then
         next_state <= op21_st;
      else
         -- Correcao
         next_state <= op14_st;
      end if;
   end if;

when op14_st =>
   sel_fsm    <= '0';
   en_mp      <= '0';
   if a_cnt < n/w + 1 then
      en_sub     <= '1';
      next_state <= op14_st;
   else
      en_sub     <= '0';
      if a_cnt < n/w + 2 then
         next_state <= op14_st;
      else
         next_state <= op21_st;
      end if;
   end if;

when op21_st =>
   sel_fsm    <= '1';
   en_mp      <= '0';
   en_sub        <= '0';
   next_state <= op22_st;

when op22_st =>
   sel_fsm    <= '1';
   en_mp      <= '1';
   en_sub        <= '0';
   if a_cnt /= n/w then
      next_state <= op22_st;
   else
      next_state <= op23_st;
   end if;

when op23_st =>
   sel_fsm    <= '1';
   en_mp      <= '0';
   en_sub        <= '0';
```

```
   if rdy_sig = '0' then
      next_state <= op23_st;
   else
      next_state <= op24_st;
   end if;

when op24_st =>
   sel_fsm    <= '1';
   en_mp      <= '0';
   en_sub        <= '0';
   if rdy_sig = '1' then
      next_state <= op24_st;
   else
      if x_reg(n/w) = 0 then
         next_state <= op31_st;
      else
         -- Correcao
         next_state <= op25_st;
      end if;
   end if;

when op25_st =>
   sel_fsm    <= '1';
   en_mp      <= '0';
   if a_cnt < n/w + 1 then
      en_sub     <= '1';
      next_state  <= op25_st;
   else
      en_sub     <= '0';
      if a_cnt < n/w + 2 then
         next_state <= op25_st;
      else
         next_state <= op31_st;
      end if;
   end if;

when op31_st =>
   sel_fsm    <= '1';
   en_mp      <= '1';
   en_sub        <= '0';
   if a_cnt /= n/w then
      next_state <= op31_st;
   else
      next_state <= op32_st;
   end if;

when op32_st =>
   sel_fsm    <= '1';
   en_mp      <= '0';
   en_sub        <= '0';
   if rdy_sig = '0' then
      next_state <= op32_st;
   else
      next_state <= op33_st;
   end if;

when op33_st =>
   sel_fsm    <= '1';
   en_mp      <= '0';
   en_sub        <= '0';
   if rdy_sig = '1' then
      next_state <= op33_st;
   else
      if x_reg(n/w) = 0 then
```

```
         if e_bit = '1' then
            next_state <= op41_st;
         else
            next_state <= op45_st;
         end if;
      else
         -- Correcao
         next_state <= op34_st;
      end if;
   end if;

when op34_st =>
   sel_fsm    <= '1';
   en_mp      <= '0';
   if a_cnt < n/w + 1 then
      en_sub     <= '1';
      next_state <= op34_st;
   else
      en_sub     <= '0';
      if a_cnt < n/w + 2 then
         next_state <= op34_st;
      else
         if e_bit = '1' then
            next_state <= op41_st;
         else
            next_state <= op45_st;
         end if;
      end if;
   end if;

when op41_st =>
   sel_fsm    <= '0';
   en_mp      <= '1';
   en_sub        <= '0';
   if a_cnt /= n/w then
      next_state <= op41_st;
   else
      next_state <= op42_st;
   end if;

when op42_st =>
   sel_fsm    <= '0';
   en_mp      <= '0';
   en_sub        <= '0';
   if rdy_sig = '0' then
      next_state <= op42_st;
   else
      next_state <= op43_st;
   end if;

when op43_st =>
   sel_fsm    <= '0';
   en_mp      <= '0';
   en_sub        <= '0';
   if rdy_sig = '1' then
      next_state <= op43_st;
   else
      if x_reg(n/w) = 0 then
         next_state <= op45_st;
      else
         -- Correcao
         next_state <= op44_st;
      end if;
   end if;
```

```
when op44_st =>
   sel_fsm    <= '1';
   en_mp      <= '0';
   if a_cnt < n/w + 1 then
      en_sub     <= '1';
      next_state <= op44_st;
   else
      en_sub     <= '0';
      if a_cnt < n/w + 2 then
         next_state <= op44_st;
      else
         next_state <= op45_st;
      end if;
   end if;

when op45_st =>
   sel_fsm    <= '0';
   en_mp      <= '0';
   en_sub        <= '0';
   if b_cnt /= 0 then
      next_state <= op31_st;
   else
      next_state <= op51_st;
   end if;

when op51_st =>
   sel_fsm    <= '1';
   en_mp      <= '0';
   en_sub        <= '0';
   next_state    <= op52_st;

when op52_st =>
   sel_fsm    <= '1';
   en_mp      <= '1';
   en_sub        <= '0';
   if a_cnt /= n/w then
      next_state <= op52_st;
   else
      next_state <= op53_st;
   end if;

when op53_st =>
   sel_fsm    <= '1';
   en_mp      <= '0';
   en_sub        <= '0';
   if rdy_sig = '0' then
      next_state <= op53_st;
   else
      next_state <= op54_st;
   end if;

when op54_st =>
   sel_fsm    <= '1';
   en_mp      <= '0';
   en_sub        <= '0';
   if rdy_sig = '1' then
      next_state <= op54_st;
   else
      if m_reg(n/w) = 0 then
         next_state <= op61_st;
      else
         -- Correcao
         next_state <= op55_st;
```

```
            end if;
         end if;

      when op55_st =>
         sel_fsm    <= '0';
         en_mp      <= '0';
         if a_cnt < n/w + 1 then
            en_sub     <= '1';
            next_state <= op55_st;
         else
            en_sub     <= '0';
            if a_cnt < n/w + 2 then
               next_state <= op55_st;
            else
               next_state <= op61_st;
            end if;
         end if;

      when op61_st =>
         sel_fsm    <= '0';
         en_mp      <= '0';
         en_sub        <= '0';
         if a_cnt /= n/l-1 then
            next_state <= op61_st;
         else
            next_state <= m1_st;
         end if;

      when others =>
         sel_fsm    <= '0';
         en_mp      <= '0';
         en_sub        <= '0';
         next_state <= m1_st;
   end case;
 end if;
end process;

process (clk)
begin
 if clk'event and clk = '1' then
   case current_state is
      when m1_st =>
         rdy_ff        <= '0';
         dat_reg(w/l-1)<= dat_i;
         a_cnt      <= (others => '0');
         s_reg      <= (others => '0');

      when m2_st =>
         dat_reg       <= shift_byte(w, l, dat_i, dat_reg);
         if a_cnt =w/l - 1 then
            a_cnt    <= (others => '0');
            acc_reg     <= shift_byte(n+w, w,  dat_reg, acc_reg);
         else
            a_cnt    <= a_cnt + 1;
         end if;

      when e1_st =>
         dat_reg(w/l-1)<= dat_i;

      when e2_st =>
         dat_reg       <= shift_byte(w, l, dat_i, dat_reg);
         if a_cnt =w/l - 1 then
            a_cnt    <= (others => '0');
            e_reg    <= shift_byte(n+w, w,  dat_reg, e_reg);
```

```
            else
               a_cnt    <= a_cnt + 1;
            end if;

         when n1_st =>
            dat_reg(w/l-1)<= dat_i;

         when n2_st =>
            dat_reg  <= shift_byte(w, l, dat_i, dat_reg);
            if a_cnt =w/l - 1 then
               a_cnt <= (others => '0');
               n_reg <= shift_byte(n+w, w,  dat_reg, n_reg);
            else
               a_cnt <= a_cnt + 1;
            end if;

         when k1_st =>
            dat_reg(w/l-1)<= dat_i;

         when k2_st =>
            dat_reg  <= shift_byte(w, l, dat_i, dat_reg);
            if a_cnt = w/l - 1 then
               a_cnt <= (others => '0');
               x_reg <= shift_byte(n+w, w,  dat_reg, x_reg);
            else
               a_cnt <= a_cnt + 1;
            end if;

-- Até aqui la legal!!!

         when op10_st =>
            m_reg    <= shift_byte(n+w, w, zero_sig , m_reg);
            e_reg    <= shift_byte(n+w, w, zero_sig , e_reg);
            n_reg    <= shift_byte(n+w, w, zero_sig , n_reg);
            x_reg    <= shift_byte(n+w, w, zero_sig , x_reg);
            acc_reg  <= shift_byte(n+w, w, zero_sig , acc_reg);

         when op11_st =>
            a_cnt    <= a_cnt + 1;
            n_reg    <= rotate_byte(n+w, w, n_reg);
            x_reg    <= rotate_byte(n+w, w, x_reg);

         when op12_st =>
            if rdy_sig = '1' then
               m_reg <= shift_byte(n+w, w,  s_sig, m_reg);
            else
               if shx_sig = '1' then
                  acc_reg  <= special_shift(n+w, p, acc_reg);
               end if;
            end if;

         when op13_st =>
            if rdy_sig = '1' then
               m_reg <= shift_byte(n+w, w,  s_sig, m_reg);
            else
               a_cnt <= (others => '0');
            end if;

         when op14_st =>
            a_cnt    <= a_cnt + 1;
            m_reg    <= shift_byte(n+w, w, s_sub, m_reg);
            if a_cnt < n/w + 1 then
               n_reg <= rotate_byte(n+w, w, n_reg);
            elsif a_cnt = n/w + 2 then
```

```
      a_cnt <= (others => '0');
   end if;

when op21_st =>
   acc_reg  <= set_array(n+w, w);

when op22_st =>
   a_cnt    <= a_cnt + 1;
   n_reg    <= rotate_byte(n+w, w, n_reg);
   x_reg    <= rotate_byte(n+w, w, x_reg);

when op23_st =>
   if rdy_sig = '1' then
      x_reg    <= shift_byte(n+w, w,  s_sig, x_reg);
      acc_reg  <= shift_byte(n+w, w,  s_sig, acc_reg);
   else
      if shx_sig = '1' then
         acc_reg  <= special_shift(n+w, p, acc_reg);
      end if;
   end if;

when op24_st =>
   if rdy_sig = '1' then
      x_reg    <= shift_byte(n+w, w,  s_sig, x_reg);
      acc_reg     <= shift_byte(n+w, w,  s_sig, acc_reg);
   else
      a_cnt       <= (others => '0');
      b_cnt       <= (others => '1');
   end if;

when op25_st =>
   a_cnt       <= a_cnt + 1;
   x_reg       <= shift_byte(n+w, w, s_sub, x_reg);
   acc_reg        <= shift_byte(n+w, w, s_sub, acc_reg);
   if a_cnt < n/w + 1 then
      n_reg    <= rotate_byte(n+w, w, n_reg);
   elsif a_cnt = n/w + 2 then
      a_cnt    <= (others => '0');
   end if;

when op31_st =>
   a_cnt       <= a_cnt + 1;
   n_reg       <= rotate_byte(n+w, w, n_reg);
   x_reg       <= rotate_byte(n+w, w, x_reg);

when op32_st =>
   if rdy_sig = '1' then
      x_reg    <= shift_byte(n+w, w,  s_sig, x_reg);
      acc_reg     <= shift_byte(n+w, w,  s_sig, acc_reg);
   else
      if shx_sig = '1' then
         acc_reg  <= special_shift(n+w, p, acc_reg);
      end if;
   end if;

when op33_st =>
   if rdy_sig = '1' then
      x_reg    <= shift_byte(n+w, w,  s_sig, x_reg);
      acc_reg     <= shift_byte(n+w, w,  s_sig, acc_reg);
   else
      a_cnt    <= (others => '0');
   end if;

when op34_st =>
```

```
      a_cnt      <= a_cnt + 1;
      x_reg      <= shift_byte(n+w, w, s_sub, x_reg);
      acc_reg       <= shift_byte(n+w, w, s_sub, acc_reg);
      if a_cnt < n/w + 1 then
         n_reg   <= rotate_byte(n+w, w, n_reg);
      elsif a_cnt = n/w + 2 then
         a_cnt   <= (others => '0');
      end if;

   when op41_st =>
      a_cnt      <= a_cnt + 1;
      n_reg      <= rotate_byte(n+w, w, n_reg);
      m_reg      <= rotate_byte(n+w, w, m_reg);

   when op42_st =>
      if rdy_sig = '1' then
         x_reg   <= shift_byte(n+w, w,  s_sig, x_reg);
         acc_reg    <= shift_byte(n+w, w,  s_sig, acc_reg);
      else
         if shx_sig = '1' then
            acc_reg  <= special_shift(n+w, p, acc_reg);
         end if;
      end if;

   when op43_st =>
      if rdy_sig = '1' then
         x_reg   <= shift_byte(n+w, w,  s_sig, x_reg);
         acc_reg    <= shift_byte(n+w, w,  s_sig, acc_reg);
      else
         a_cnt   <= (others => '0');
      end if;

   when op44_st =>
      a_cnt      <= a_cnt + 1;
      x_reg      <= shift_byte(n+w, w, s_sub, x_reg);
      acc_reg       <= shift_byte(n+w, w, s_sub, acc_reg);
      if a_cnt < n/w + 1 then
         n_reg   <= rotate_byte(n+w, w, n_reg);
      elsif a_cnt = n/w + 2 then
         a_cnt   <= (others => '0');
      end if;

   when op45_st =>
      b_cnt   <= b_cnt - 1;

   when op51_st =>
      acc_reg       <= set_array(n+w, w);

   when op52_st =>
      a_cnt      <= a_cnt + 1;
      n_reg      <= rotate_byte(n+w, w, n_reg);
      x_reg      <= rotate_byte(n+w, w, x_reg);

   when op53_st =>
      if rdy_sig = '1' then
         m_reg   <= shift_byte(n+w, w,  s_sig, m_reg);
      else
         if shx_sig = '1' then
            acc_reg  <= special_shift(n+w, p, acc_reg);
         end if;
      end if;

   when op54_st =>
      if rdy_sig = '1' then
```

```
            m_reg    <= shift_byte(n+w, w,  s_sig, m_reg);
         else
            a_cnt    <= (others => '0');
         end if;

      when op55_st =>
         a_cnt       <= a_cnt + 1;
         m_reg       <= shift_byte(n+w, w, s_sub, x_reg);
         if a_cnt < n/w + 1 then
            n_reg    <= rotate_byte(n+w, w, n_reg);
         elsif a_cnt = n/w + 2 then
            a_cnt    <= (others => '0');
         end if;

      when op61_st =>
         a_cnt <= a_cnt + 1;
         m_reg    <= shift_byte(1, m_reg);
         s_reg <= m_reg(0)(l-1 downto 0);
         rdy_ff   <= '1';

      when others =>
         null;
   end case;
   current_state <= next_state;
end if;
end process;

zero_sig<= (others => '0');
x_sig <= array2vector(acc_reg)(p-1 downto 0);

mp_block: mp
   generic map(
      n  => n,
      w  => w,
      e  => e,
      p  => p)
   port map(
      clk      => clk,
      rst      => rst,
      en_i  => en_mp,
      m_i      => n_reg(0),
      y_i      => y_sig,
      x_i      => x_sig,
      rdy_o => rdy_sig,
      shx_o => shx_sig,
      s_o      => s_sig);

sub_blk: sub
   generic map(w => w)
   port map(
      clk      => clk,
      rst      => rst,
      en_i  => en_sub,
      a_i      => y_sig,
      n_i      => n_reg(0),
      s_o      => s_sub);

y_sig <= m_reg(0) when (sel_fsm = '0') else
            x_reg(0);

rdy_o <= rdy_ff;
dat_o <= s_reg;
end rtl;
```

```
--------------------------------------------------------------------------------
-- Autor   : Alcides Silveira Costa
-- Bloco   : Montgomery Pipeline
--------------------------------------------------------------------------------
library ieee;
   use ieee.std_logic_1164.all;
   use ieee.std_logic_unsigned.all;

entity mp is
   generic(
      n  : positive := 16;
      w  : positive := 8;
      e  : positive := 3;
      p  : positive := 2);
   port(
      -- Entradas
      clk   : in std_logic;
      rst   : in std_logic;
      en_i  : in std_logic;
      m_i   : in std_logic_vector(w-1 downto 0);
      y_i   : in std_logic_vector(w-1 downto 0);
      x_i   : in std_logic_vector(p-1 downto 0);

      -- Saidas
      rdy_o : out std_logic;
      shx_o : out std_logic;
      s_o   : out std_logic_vector(w-1 downto 0));
end mp;

architecture rtl of mp is
-- Componentes
component pe is
   generic(w  : positive := 1);
   port(
      clk   : in std_logic;
      rst   : in std_logic;
      en_i  : in std_logic;
      s_i   : in std_logic_vector(w-1 downto 0);
      m_i   : in std_logic_vector(w-1 downto 0);
      y_i   : in std_logic_vector(w-1 downto 0);
      x_i   : in std_logic;
      rdy_o : out std_logic;
      s_o   : out std_logic_vector(w-1 downto 0);
      m_o   : out std_logic_vector(w-1 downto 0);
      y_o   : out std_logic_vector(w-1 downto 0));
end component;

-- Funções
function log2(x: integer) return integer is
   -- Funciona apenas para numero base 2
   variable result : integer := 1;
   variable aux  : integer;
   begin
      aux   := x;
      if aux /= 1 then
         while aux /= 2 loop
            aux := aux/2;
            result := result + 1;
         end loop;
      else
         result := 0;
      end if;
   return result;
end log2;
```

```
-- FSM
type states is (a_st, b_st, d_st, f_st, g_st);
signal current_state, next_state: states;

-- Registradores de amostragem, retardo e saida
signal m_reg  : std_logic_vector(w-1 downto 0);
signal y_reg  : std_logic_vector(w-1 downto 0);
signal s_reg  : std_logic_vector(w-1 downto 0);

-- Registradores internos
signal a_cnt  : std_logic_vector(log2(n)-log2(p-1)-1 downto 0);
signal b_cnt  : std_logic_vector(log2(e-1) downto 0);

-- Fios internos
type array_of_wires is array (p downto 0) of std_logic_vector(w-1 downto 0);
signal s_int  : array_of_wires;
signal m_int  : array_of_wires;
signal y_int  : array_of_wires;
signal en_int : std_logic_vector(p downto 0);

-- Flip-flops
signal rdy_ff : std_logic;
signal shx_ff : std_logic;

-- Sinais de selecao de multiplexadores
signal sel_fsm   : std_logic_vector(1 downto 0);
signal zero_sig  : std_logic_vector(w-1 downto 0);

begin
zero_sig <= (others => '0');

type1_fsm: if n/w > 2 generate
   process (current_state, rst, en_i, a_cnt, en_int(n mod p))
   begin
      if (rst = '1') then
         sel_fsm <= "00";
         next_state <= a_st;
      else
         case current_state is
            when a_st =>
               sel_fsm <= "00";
               if (en_i = '0') then
                  next_state <= a_st;
               else
                  next_state <= b_st;
               end if;

            when b_st =>
               sel_fsm <= "01";
               if (en_i = '1') then
                  next_state <= b_st;
               else
                  if ((n/p - 1)/= 0) then
                     next_state <= d_st;
                  else
                     next_state <= f_st;
                  end if;
               end if;

            when d_st =>
               sel_fsm <= "10";
               if (a_cnt /= (n/p)- 1) then
                  next_state <= d_st;
```

```
            else
               next_state <= f_st;
            end if;

         when f_st =>
            sel_fsm <= "10";
            if (en_int(n mod p) = '1') then
               next_state <= f_st;
            else
               next_state <= g_st;
            end if;

         when g_st =>
            sel_fsm <= "10";
            if (en_int(n mod p) = '1') then
               next_state <= g_st;
            else
               next_state <= a_st;
            end if;

         when others =>
            sel_fsm <= "00";
            next_state <= a_st;
      end case;
   end if;
end process;
end generate;

-------------------------------------------------------------------------------

type2_fsm: if n/w = 2 generate
   process (current_state, rst, en_i, a_cnt, en_int(n mod p))
   begin
      if (rst = '1') then
         sel_fsm <= "00";
         next_state <= a_st;
      else
         case current_state is
            when a_st =>
               sel_fsm <= "00";
               if (en_i = '0') then
                  next_state <= a_st;
               else
                  next_state <= b_st;
               end if;

            when b_st =>
               sel_fsm <= "01";
               if (en_i = '1') then
                  next_state <= b_st;
               else
                  if ((n/p - 1)/= 0) then
                     next_state <= d_st;
                  else
                     next_state <= f_st;
                  end if;
               end if;

            when d_st =>
               sel_fsm <= "10";
               if (a_cnt /= (n/p)- 2) then
                  next_state <= d_st;
               else
                  next_state <= f_st;
```

```
            end if;

         when f_st =>
            sel_fsm  <= "10";
            if (en_int(n mod p) = '1') then
               next_state <= f_st;
            else
               next_state <= g_st;
            end if;

         when g_st =>
            sel_fsm  <= "10";
            if (en_int(n mod p) = '1') then
               next_state <= g_st;
            else
               next_state <= a_st;
            end if;

         when others =>
            sel_fsm  <= "00";
            next_state <= a_st;
      end case;
   end if;
 end process;
end generate;

-------------------------------------------------------------------------------

   process (clk)
   begin
      if clk'event and clk = '1' then
         case current_state is
            when a_st =>
               m_reg    <= m_i;
               y_reg    <= y_i;
               a_cnt       <= (others => '0');
               b_cnt       <= (others => '0');
               s_reg    <= (others => '0');
               shx_ff      <= '0';
               rdy_ff      <= '0';

            when b_st =>
               m_reg <= m_i;
               y_reg <= y_i;
               if (en_i = '0') then
                  shx_ff   <= '1';
               end if;

            when d_st =>
               if (b_cnt /= e) then
                  b_cnt    <= b_cnt + 1;
                  shx_ff   <= '0';
               else
                  a_cnt    <= a_cnt + 1;
                  b_cnt    <= (others => '0');
                  shx_ff   <= '1';
               end if;

            when f_st =>
               a_cnt    <= (others => '0');
               b_cnt    <= (others => '0');
               shx_ff   <= '0';

            when g_st =>
```

```
               m_reg <= m_i;
               y_reg <= y_i;
               s_reg <= s_int(n mod p);
               rdy_ff  <= en_int(n mod p);

            when others =>
               null;
         end case;
         current_state <= next_state;
      end if;
   end process;

MontgomeryPipeline: for k in 0 to p-1 generate
   MP: pe
   generic map(w)
   port map(
      clk   => clk,
      rst   => rst,
      en_i  => en_int(k),
      s_i   => s_int(k),
      m_i   => m_int(k),
      y_i   => y_int(k),
      x_i   => x_i(k),
      rdy_o => en_int(k+1),
      s_o   => s_int(k+1),
      m_o   => m_int(k+1),
      y_o   => y_int(k+1));
end generate;

   en_int(0)  <= '0'      when (sel_fsm = "00") else
            '1'      when (sel_fsm = "01") else
                 en_int(p);

   s_int(0) <= zero_sig when (sel_fsm = "00") else
                 zero_sig when (sel_fsm = "01") else
                 s_int(p);

   m_int(0) <= zero_sig when (sel_fsm = "00") else
                 m_reg    when (sel_fsm = "01") else
                 m_int(p);

   y_int(0) <= zero_sig when (sel_fsm = "00") else
                 y_reg    when (sel_fsm = "01") else
                 y_int(p);

shx_o <= shx_ff;
rdy_o <= rdy_ff;
s_o   <= s_reg;
end rtl;

--------------------------------------------------------------------------------
-- Autor    : Alcides Silveira Costa
-- Bloco    : Processing Element
--------------------------------------------------------------------------------
library ieee;
   use ieee.std_logic_1164.all;
   use ieee.std_logic_unsigned.all;

entity pe is
   generic(w   : positive := 4);
   port(
      -- Entradas
      clk   : in std_logic;
      rst   : in std_logic;
```

```
      en_i  : in std_logic;
      s_i   : in std_logic_vector(w-1 downto 0);
      m_i   : in std_logic_vector(w-1 downto 0);
      y_i   : in std_logic_vector(w-1 downto 0);
      x_i   : in std_logic;

      -- Saidas
      rdy_o : out std_logic;
      s_o   : out std_logic_vector(w-1 downto 0);
      m_o   : out std_logic_vector(w-1 downto 0);
      y_o   : out std_logic_vector(w-1 downto 0));
end pe;

architecture rtl of pe is
-- FSM
type states is (A, B);
signal current_state, next_state: states;

-- Registradores de amostragem, retardo e/ou saida
signal m_reg1 : std_logic_vector(w-1 downto 0);
signal y_reg1 : std_logic_vector(w-1 downto 0);

signal m_reg2 : std_logic_vector(w-1 downto 0);
signal y_reg2 : std_logic_vector(w-1 downto 0);

-- Registradores internos
signal cs_reg1: std_logic_vector(w+1 downto 1);
signal cs_reg2: std_logic_vector(w-1 downto 0);

-- Fios internos
signal wire_int1: std_logic_vector(w+1 downto 0);
signal wire_int2: std_logic_vector(w+1 downto 0);

-- Flip-flops
signal s_ff: std_logic;
signal x_ff: std_logic;
signal en_ff1 : std_logic;
signal en_ff2 : std_logic;

-- Sinais de selecao de multiplexadores
signal sel_a  : std_logic;
signal sel_b  : std_logic;
signal sel_c  : std_logic;
signal sel_x  : std_logic;

begin

process (current_state, rst, en_i)
begin
   if (rst = '1') then
      sel_b <= '0';
      sel_x <= '0';
      next_state <= A;
   else
      case current_state is
         when A =>
            sel_b <= '0';
            sel_x <= '0';
            if (en_i = '0') then
               next_state <= A;
            else
               next_state <= B;
            end if;
```

```
         when B =>
            sel_b <= '1';
            sel_x <= '1';
            if (en_i = '1') then
               next_state <= B;
            else
               next_state <= A;
            end if;

         when others =>
            sel_b <= '0';
            sel_x <= '0';
            next_state <= A;
      end case;
   end if;
end process;

process (clk)
begin
   if clk'event and clk = '1' then
      cs_reg1  <= wire_int2(w+1 downto 1);
      m_reg1   <= m_i;
      y_reg1   <= y_i;
      en_ff1   <= en_i;

      case current_state is
         when A =>
            x_ff  <= x_i;
            s_ff  <= wire_int1(0);
            cs_reg2 <= (others => '0');
            m_reg2 <= (others => '0');
            y_reg2 <= (others => '0');
            en_ff2   <= '0';

         when B =>
            cs_reg2 <= wire_int2(0) & cs_reg1(w-1 downto 1);
            m_reg2 <= m_reg1;
            y_reg2 <= y_reg1;
            en_ff2   <= en_ff1;

         when others =>
            null;

         end case;
      current_state <= next_state;
   end if;
end process;

Mux_X:
with sel_x select
   sel_a <=    x_i   when '0',
               x_ff  when '1',
               x_i   when others;

Mux_A:
with sel_a select
   wire_int1<=   "00"&s_i                  when '0',
                 ("00"&s_i) + ("00"&y_i)   when '1',
                 "00"&s_i                  when others;

Mux_B:
with sel_b select
   sel_c <=    wire_int1(0 when '0',
               s_ff    when '1',
```

```
              wire_int1(0)  when others;

Mux_C:
with sel_c select
   wire_int2<= wire_int1 + cs_reg1(w+1 downto w)               when '0',
               wire_int1 + cs_reg1(w+1 downto w) + ("00"&m_i)  when '1',
               wire_int1 + cs_reg1(w+1 downto w)               when others;


rdy_o <= en_ff2;
s_o     <= cs_reg2;
m_o     <= m_reg2;
y_o     <= y_reg2;
end rtl;

--------------------------------------------------------------------------------
-- Autor    : Alcides Silveira Costa
-- Bloco    : Subtrator em pipeline
--------------------------------------------------------------------------------
library ieee;
   use ieee.std_logic_1164.all;
   use ieee.std_logic_unsigned.all;

entity sub is
   generic(w  : positive := 8);
   port(
      -- Entradas
      clk   : in std_logic;
      rst   : in std_logic;
      en_i  : in std_logic;
      a_i   : in std_logic_vector(w-1 downto 0);
      n_i   : in std_logic_vector(w-1 downto 0);

      -- Saidas
      s_o   : out std_logic_vector(w-1 downto 0));
end sub;

architecture rtl of sub is
-- FSM
type states is (a_st, b_st);
signal current_state, next_state: states;

-- Registradores internos
signal a_reg  : std_logic_vector(w-1 downto 0);
signal n_reg  : std_logic_vector(w-1 downto 0);
signal s_reg  : std_logic_vector(w-1 downto 0);
signal carry_ff  : std_logic;

-- Sinais internos
signal s_sig  : std_logic_vector(w downto 0);
signal carry_sig: std_logic_vector(w-1 downto 1);

begin

process (current_state, rst, en_i)
begin
   if (rst = '1') then
      next_state <= a_st;
   else
      case current_state is
         when a_st =>
            if (en_i = '0') then
               next_state <= a_st;
            else
```

```
               next_state <= b_st;
            end if;

         when b_st =>
            if (en_i = '1') then
               next_state <= b_st;
            else
               next_state <= a_st;
            end if;

         when others =>
            next_state <= a_st;
      end case;
   end if;
end process;

process (clk)
begin
   if clk'event and clk = '1' then
      a_reg    <= a_i;
      n_reg    <= n_i;
      case current_state is
         when a_st =>
            carry_ff <= '0';
            s_reg    <= (others => '0');

         when b_st =>
            carry_ff <= s_sig(w);
            s_reg      <= s_sig(w-1 downto 0);

         when others =>
            null;

         end case;
      current_state <= next_state;
   end if;
end process;

carry_sig  <=  (others => '0');
s_sig      <=  ('0'&a_reg) - ('0'&n_reg) - ('0'&carry_sig&carry_ff);
s_o        <= s_reg;
end rtl;
```