University of Plymouth

PEARL **https://pearl.plymouth.ac.uk**

04 University of Plymouth Research Theses 01 Research Theses Main Collection

1995

A high-speed integrated circuit with applications to RSA Cryptography

Onions, Paul David

http://hdl.handle.net/10026.1/337

http://dx.doi.org/10.24382/3349
University of Plymouth

All content in PEARL is protected by copyright law. Author manuscripts are made available in accordance with publisher policies. Please cite only the published version using the details provided on the item record or document. In the absence of an open licence (e.g. Creative Commons), permissions for further reuse of content should be sought from the publisher or author.

A High-Speed Integrated Circuit for Applications to RSA Cryptography

by

Paul David Onions

BEng(Hons), MSc

A thesis submitted to the University of Plymouth

in partial fulfilment for the degree of

DOCTOR OF PHILOSOPHY

School of Electronic Communication and Electrical Engineering

Faculty of Technology

July 1995

For Sian,
who wanted to wear the floppy hat.

A High-Speed Integrated Circuit for Applications to RSA Cryptography

Paul David Onions
BEng(Hons), MSc

The rapid growth in the use of computers and networks in government, commercial and private communications systems has led to an increasing need for these systems to be secure against unauthorised access and eavesdropping. To this end, modern computer security systems employ public-key ciphers, of which probably the most well known is the RSA ciphersystem, to provide both secrecy and authentication facilities.

The basic RSA cryptographic operation is a modular exponentiation where the modulus and exponent are integers typically greater than 500 bits long. Therefore, to obtain reasonable encryption rates using the RSA cipher requires that it be implemented in hardware.

This thesis presents the design of a high-performance VLSI device, called the WHiSpER chip, that can perform the modular exponentiations required by the RSA cryptosystem for moduli and exponents up to 506 bits long. The design has an expected throughput in excess of 64kbit/s making it attractive for use both as a general RSA processor within the security function provider of a security system, and for direct use on moderate-speed public communication networks such as ISDN.

The thesis investigates the low-level techniques used for implementing high-speed arithmetic hardware in general, and reviews the methods used by designers of existing modular multiplication/exponentiation circuits with respect to circuit speed and efficiency.

A new modular multiplication algorithm, MMDDAMMM, based on Montgomery arithmetic, together with an efficient multiplier architecture, are proposed that remove the speed bottleneck of previous designs.

Finally, the implementation of the new algorithm and architecture within the WHiSpER chip is detailed, along with a discussion of the application of the chip to ciphering and key generation.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to express my sincere thanks to the following people,

- Peter Sanders, my Director of Studies, for his encouragement and motivation, and for his confidence in my abilities throughout the research program,
- Alan Roberts for his help with the GPS/Mentor ECAD design environment,
- Simon Shepherd for originally suggesting the direction of the research, and
- to all the other members of the Network Research Group and to the technicians of the School of Electronic Communication and Electrical Engineering for their help and support during the last two and a half years.

Declaration

At no time during the registration for the degree of Doctor of Philosophy has the author been registered for any other university award.

Relevant scientific seminars and conferences were regularly attended at which work was often presented; external institutions were visited for consultation purposes, and several papers prepared for publication.

The work presented in this thesis is solely that of the author.

Signed .....

Date 29-9-95

Glossary

$'$	the prime symbol applied to the variable x, as x', is usually used to indicate a modification to x. The exact meaning should be clear from the context in which it is used.
$[x_{k-1}x_{k-2}\ldots x_0]$	is the k-bit vector representation of the integer X, whose value is given by $X = \sum_{i=0}^{k-1} 2^i \cdot x_i$ where $x_i \in \{0,1\}$.
$[X_{l-1}X_{l-2}\ldots X_0]$	is the l-digit vector representation of the integer X, whose value is given by $X = \sum_{i=0}^{l-1} 2^{ib} \cdot X_i$ for some integer $b > 0$.
$x(i)$	the i-th recoded digit of X. See Sections 4.3 and 7.2.1.
$x \mid m$	means x divides into m exactly.
$[a,b]$	integer range a to b inclusive.
(a,b)	integer range a to b exclusive.
$\lfloor x/m \rfloor$	greatest integer not exceeding x/m.
$\lceil x/m \rceil$	least integer not less than x/m.
$\langle x \rangle_m$	least non-negative residue of x modulo m.
$x^{-1} \pmod m$	multiplicative inverse of x modulo m.
$\Phi(m)$	Euler's totient on m.
Δ_{AND}	propagation delay of AND gate.

Δ_{NAND}	propagation delay of NAND gate.
Δ_{HA}	propagation delay of Half Adder.
Δ_{FA}	propagation delay of Full Adder.
Δ_{MUX}	propagation delay of 4-to-1 Multiplexer.
Δ_{FF}	setup and propagation delay of Flip-Flop register element.
Ω_{AND}	gate-count complexity of AND gate.
Ω_{NAND}	gate-count complexity of NAND gate.
Ω_{HA}	gate-count complexity of Half Adder.
Ω_{FA}	gate-count complexity of Full Adder.
Ω_{MUX}	gate-count complexity of 4-to-1 Multiplexer.
Ω_{FF}	gate-count complexity of Flip-Flop register element.
$\mathcal{M}_{R,N}(X,Y)$	fully reduced Montgomery multiplication. See Section 6.2.2.
$\lambda_{R,M}(X,Y)$	partially reduced Montgomery multiplication. See Section 8.3.3.
$\mathcal{T}_M$	time (in nanoseconds) required for one multiplication. See Section 8.2.1.
$\mathcal{R}_M$	multiplication rate.
$\mathcal{G}_M$	multiplier circuit gate-count.
$\mathcal{E}_M$	multiplier circuit efficiency.
$\mathcal{T}_E$	time (in nanoseconds) required for one exponentiation. See Section 8.3.1.
$\mathcal{R}_E$	exponentiation rate.

$\mathcal{G}_E$	exponentiator circuit gate-count.
$\mathcal{E}_E$	exponentiator circuit efficiency.
X^+, X^-	positive and negative component vectors of RSD representation.
AE	Array Element.
AMMM	Additive Montgomery Modular Multiplication.
ASIC	Application Specific Integrated Circuit.
CMOS	Complementary Metal-Oxide Semiconductor.
CPA	Carry-Propagate Adder.
CRT	Chinese Remainder Theorem.
CSA	Carry-Save Adder.
DAMMM	Delayed Additive Montgomery Modular Multiplication.
DCA	Delay-Carry Adder.
DR	Diminished Radix.
FA	Full Adder.
FCPA	Fast Carry-Propagate Adder.
FPGA	Field Programmable Gate Array.
GFA	General Full Adder.
GPS	GEC Plessey Semiconductors.
HA	Half Adder.
L-to-R	Left-to-Right.

MMDAMMM	Modified Modulus Delayed Additive Montgomery Modular Multiplication.
MMDDAMMM	Modified Modulus Double Delayed Additive Montgomery Modular Multiplication.
MUX	Multiplexer.
PAM	Programmable Array Memory.
RAM	Random Access Memory.
RNS	Residue Number System.
ROM	Read Only Memory.
RSA	Rivest, Shamir and Adleman.
RSD	Redundant Signed-Digit.
R-to-L	Right-to-Left.
SRAM	Static RAM.
VLSI	Very Large Scale Integration.
WHiSpER	Wide-word High-Speed Encryption for RSA.

Chapter 1

Introduction

This thesis concerns the design of a high-speed integrated circuit device capable of performing the modular exponentiation operations required of the RSA cryptosystem for moduli of around 500 bits in length.

The thesis is constructed as a progression starting from basic concepts in Chapter 2 through standard arithmetic algorithms and hardware in Chapters 3 and 4, current RSA hardware in Chapter 5, Montgomery arithmetic and optimised Montgomery multipliers in Chapters 6 and 7 and culminating in details of the WHiSpER chip in Chapter 8 followed by conclusions and ideas for further work in Chapter 9.

Chapter 2 serves as a brief introduction to cryptography, covering the definition of a ciphersystem and the difference between secret-key and public-key systems. This is followed by an overview of the main theorems and basic algorithms of modular arithmetic, introducing the notation that will be used throughout this thesis. Finally, the RSA ciphersystem is explained in detail.

Chapter 3 reviews the binary representation of numbers and details the right-to-left and left-to-right binary evaluation techniques. These techniques are then used to derive algorithms for iterative multiplication and exponentiation.

Chapter 4 studies arithmetic hardware. Adder circuits and iterative multipliers are

explained together with a discussion of multiplier recoding techniques and signed-number representations. At the end of this chapter the design of an efficient iterative multiplier is presented, the basic principles of which will be used to construct the optimised multipliers of Chapter 7.

Chapter 5 reviews the current literature concerning the implementation of RSA cryptosystems using standard modular multipliers. In depth desciptions of two particular designs are given showing the trade-offs that have to be made in an effort to create a fast and efficient design. The limitations of current designs are identified in this chapter. In Chapter 7 it will be shown that these limitiations can be removed with optimised Montgomery multipliers.

Chapter 6 introduces Montgomery modular arithmetic. The technique is explained and basic algorithms for Montgomery multiplication and exponentiation are studied. A review of some of the proposed hardware implementation schemes is also given.

In Chapter 7 new, optimised designs for Montgomery multipliers are presented that allow the multiplier to operate at full-speed. The algorithms and architecture of Chapters 6 and 4 serve as the basis from which the new optimised Montgomery multipliers are developed.

Chapter 8 presents the design of the WHiSpER chip. Selection of the multiplier and exponentiator circuit is performed based on the technology issues of the GPS CLA7000 series gate array device used. The architecture of the chip is described, showing how an efficient and high-throughput device can be realised, followed by a description of its operation and an analysis of its expected performance. Finally, the details of a simple WHiSpER based IBM PC card are given.

Chapter 9 concludes the thesis. It discusses the use of the WHiSpER chip for applications with key sizes of up to 1000 bits and shows how key generation can be effected by using WHiSpER to implement a primality testing function.

Chapter 2

Cryptology

The discipline of *cryptology* is that of 'secret-writing' or, in less dramatic terms, the study of systems that can hide the information content of messages. Though the complexity of these systems varies considerably (from the simple letter-substitution ciphers of Julius Caesar to the sophisticated 'information-randomisation' algorithms of the present day), they are all generally known as ciphersystems or cryptosystems. This chapter serves as a brief introduction to the field of cryptology and cryptosystems [1] [2] [3].

2.1 Cryptography and Cryptanalysis

Cryptology can be split into two main areas; *cryptography* and *cryptanalysis*. Broadly speaking, it is the cryptographers job to create new ciphersystems, and the cryptanalysts job to break them. Figure 2.1 shows a generic ciphersystem. The message to be encrypted, known as the *plaintext*, is converted into an encrypted message, the *ciphertext*, by means of an encryption algorithm. The encryption algorithm makes use of an encryption key to determine the exact ciphertext produced. Decryption from the ciphertext to the plaintext follows a similar process using the decryption algorithm and a decryption key.

The goal of the cryptographer is to make these processes simple enough to be executed quickly, yet make them complex enough so that it is generally infeasible to infer properties

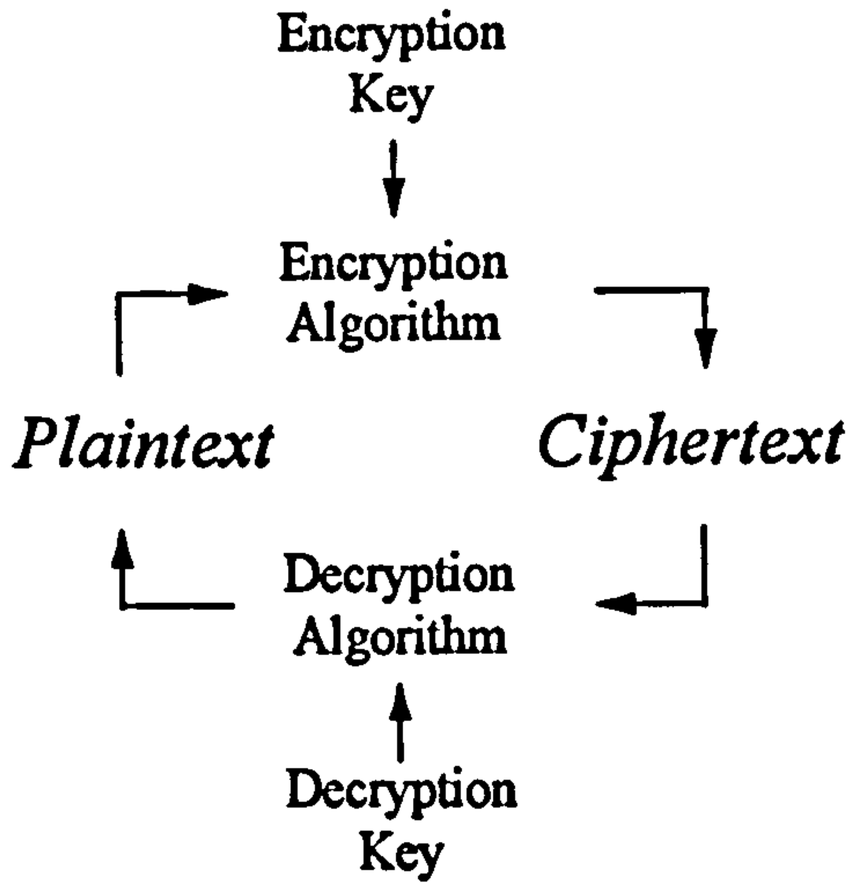


Figure 2.1: A generic ciphersystem.

of one part of the system from properties of another part of the system.

In designing such a system, however, the cryptographer must bear in mind the various different techniques that the cryptanalyst has at his disposal. Basic attack models vary in their assumptions concerning the nature of the information that the cryptanalyst has obtained. They are, in increasing order of significance,

- ciphertext-only; the cryptanalyst has access only to a selection of ciphertexts,
- known-plaintext; the cryptanalyst has access to plaintext-ciphertext pairs,
- chosen-plaintext (-ciphertext); the cryptanalyst is able to obtain the ciphertext (plaintext) corresponding to a plaintext (ciphertext) of his own choosing.

A secure ciphersystem should be resistant to all of these.

This raises the question of what exactly is meant by the term 'secure'. The following two definitions are commonly used,

- perfect security; the ciphersystem is unbreakable even under the assumption that the cryptanalyst has unlimited computing power, and
- practical security; the ciphersystem is considered unbreakable under the assumption that the cryptanalyst has powerful but finite computing resources.

It has been shown by Shannon in [4] that the only perfectly secure ciphersystem is one in which the uncertainty of the encryption/decryption key is at least as large as the uncertainty of the plaintext message, and that this key be used only once and then discarded. Such a system is known as a 'one-time-pad' and obviously has only very limited use.

The majority of ciphersystems in use today are of the practically secure type, where the level of security is often measured in terms of the number of years required to break a ciphersystem using state-of-the-art computing power. However there is no concensus on the level of difficulty of solving certain problems that arise in breaking cryptosystems, and so such estimates are always very approximate.

The elements of the ciphersystem that are considered 'secret' vary from system to system. For example in some governmental and most military ciphersystems, the encryption and decryption algorithms themselves are not public knowledge. Whether this makes the ciphersystem more secure is open to debate since although fewer people know the algorithm and thus can perform cryptanalytic attacks on it, this does not prove that the algorithm is a strong one. Indeed, there is currently no method of proving the general level of security of any ciphersystem and the only widely accepted measure of the security of a system is that the cipher algorithm be public knowledge and that it have survived repeated cryptanalytic attacks over many years.

Thus, if a particular ciphersystem is in common use (and so assuming that the algorithm is public knowledge), we see that the security of the system depends entirely on the cipher keys. The criteria for such systems can thus be summarised as follows,

- the resultant ciphertext should be statistically un-correlated with the plaintext,
- it should not be possible to determine a cipher-key given an arbitrary number of plaintext-ciphertext pairs, and
- if there is a large range from which a key can be selected (a large *key-space*), then the security of such a system is increased. e.g. against exhaustive key search attacks.

2.2 Secret- and Public-Key Ciphersystems

Ciphersystems split quite neatly into two distinct types; *secret-key* and *public-key* systems.

2.2.1 Secret-Key Ciphersystems

A secret-key system is, referring back to Figure 2.1, a system in which the encryption key and decryption key are identical. This means that, with two parties wishing to exchange secret information (call them Alice and Bob; Alice wants to send a message to Bob), it is first necessary for each of them to possess the common key. If they are physically remote from each other and wish to communicate via some sort of communications network then some method of distributing this key is needed. Obviously the key cannot be transmitted over a non-encrypted communication channel and so the usual solution to this problem (at least as far as initial key distribution is concerned) is to place the key into some secure physical device and manually transport it to Bob (assuming Alice created the key).

The above solution to the key distribution problem may be acceptable in isolated cases, but consider the situation in which there is a network of n users each of whom wishes to communicate with any one of the $n-1$ other users in a secure way. It can be seen that each of the users must possess $n-1$ keys and so the total number of keys in the system is $\frac{n(n-1)}{2} \approx n^2$ for large n. Thus for any sizeable network of n people there are two major problems associated with the use of a secret-key cryptosystem,

- key distribution; physically transporting the $n-1$ keys to each user, and
- key space; the greater the ratio of keys used to available key space, the greater the probability of 'random' cryptanalytic attacks on all n users succeeding in finding the key for one of the $\frac{n(n-1)}{2}$ keys in use.

Among the many secret-key cryptosystems in use today (such as the IDEA block cipher [5] and the various stream ciphers [1]) the most common is probably the DES (Data Encryption Standard) cipher [3]. Developed in the 1970's as an offshoot of IBM's

Lucifer ciphersystem [1], DES was accepted by the American NBS (National Bureau of Standards) as a U.S. Federal Information Processing Standard in 1977. In that same year the complete specification of DES was also published but IBM's design principles for the cipher were classified by the NSA (National Security Agency). Considerable controversy has been generated by the DES cipher, not least concerning the small key size of 56 bits. Although the resultant key space of $2^{56} \approx 10^{17}$ is considered too large to mount a brute-force cryptanalytic attack on any particular key using current computer hardware, it is not considered impossible that this approach may become feasible in the near future. Indeed, recent developments in cryptanalytic techniques (differential cryptanalysis, see [6] [7]) have shown that DES is susceptible to certain cryptanalytic attacks, and this is why enhanced DES-like algorithms (such as DES double key mode [8]) have been investigated as to their increased level of security.

2.2.2 Public-Key Ciphersystems

A public-key ciphersystem is one in which the encryption and decryption keys of Figure 2.1 are distinct. Together the keys are known as a *key-pair* and, in general, one of the keys is kept secret (known only to the generator of the key-pair) whilst the other is made public.

Returning to the example of the previous section, if Bob creates a key-pair and releases the encryption-key to the public then Alice can send secret information to Bob by encrypting her plaintext with the encryption-key and sending the resultant ciphertext to Bob. Since Bob is the only person with the decryption-key, he is the only one who can read the encrypted message and so the desired goal of keeping the communication secret has been achieved.

If Alice and Bob are now considered to belong to a network of n users, where each user creates their own key-pair and makes public only the encryption-key from this pair, then it is possible for secret communications to take place between any two users on the network. Note that this has been achieved firstly with the use of only n key-pairs and

secondly without any form of secure key-distribution being necessary.

There are a number of public-key cryptosystems known today including ElGamal's [9], the elliptic curve cryptosystems [10] and others [11] [12] [13]. However, perhaps the most widely used is the RSA system named after its inventors Rivest, Shamir and Adleman [14]. Like most other public-key systems, the RSA cipher uses elements from *number theory* to construct the encryption/decryption algorithms, and a complete description of the mathematics of RSA is given in the following sections. For the moment though, it suffices to say that the computation of an RSA ciphertext involves arithmetic calculations on very large numbers (numbers over 150 digits or 500 bits long), and the security of an RSA ciphersystem depends (in the main) on the difficulty of factoring very large integers into their prime components. An interesting consequence of the latter statement is that, since the integer factoring problem has been studied by mathematicians since 'time immemorial' with no generally applicable fast algorithm ever having been found, the security of the RSA cryptosystem has at least a sound historical footing.

More information on the security of RSA can be found in [15] [16] [17] [18] [19].

Another feature of the RSA system is that, since its level of security depends on the difficulty of the general problem of factoring integers of a given size, that level of security can be increased simply by increasing the size of the numbers that the RSA system manipulates. In other words, the RSA system is scalable with respect to security.

However, RSA is not all good news. There are two points which detract from its appeal, and these are

- calculation time; arithmetic operations on very large numbers take time, and
- security; the unknown level of hardness of the integer factorization problem.

The first of the above means that the throughput of an RSA cryptosystem is very slow compared to say the DES system. The second item above acknowledges the fact that mathematicians have never been able to show exactly how hard the integer factorization

problem is. In other words, is there a lower limit on the 'easiness' of being able to factor integers of a given size? Since this is not known then it is not impossible for someone to discover tomorrow a new algorithm to solve the integer factorization problem in very fast time, and thus render useless any cryptosystem based on this problem. In fairness though, such proofs of minimal problem complexity have not been found for other cryptosystems, and so any cipher may be rendered obsolete tomorrow by the discovery of new efficient algorithms (however unlikely).

In summary, secret-key ciphersystems are generally fast but suffer from key distribution problems, whilst public-key systems are slow but allow simpler key-management schemes.

2.3 Modular Arithmetic

Modular arithmetic is a tool that is used in the branch of mathematics known as *number theory* [20] [21] [22] [23]. Number theory, as its name suggests, is all about the properties of numbers, and in the following sections the notation, arithmetical operations, some simple algorithms and a few of the major theorems of modular arithmetic will be described.

The following is an overview of the modular arithmetic notation that will be used throughout this thesis. As a general rule, if a variable, say m, is used in the context of a modulus then it is understood that m is a positive integer. On the other hand, if a variable, say x, is used in a general context then x may be any integer.

2.3.1 Divisibility

The notation $x|m$ means that x divides into m exactly. In other words m is a multiple of x. Conversely, $x\not|m$ means that x does not divide into m.

2.3.2 Integer Range

The notation $x \in [a, b]$ means that x is an integer that ranges over the values a to b such that $a \leq x \leq b$. The notation $x \in [a, b)$ means $a \leq x < b$.

2.3.3 Integer Division

The notation $q = \lfloor x/m \rfloor$ means the largest integer q not exceeding x/m. The notation $q = \lceil x/m \rceil$ means the smallest integer q not less than x/m.

For example, $\lfloor 2.7 \rfloor = 2$ and $\lceil 2.7 \rceil = 3$.

2.3.4 Congruences

The *congruence* is a modulo relationship between numbers. The notation $x \equiv r \pmod{m}$ is pronounced 'x is congruent to r modulo m' and means that $m \mid (x - r)$, or in other words, x and r differ only by multiples of m. Thus x and r are said to belong to the same residue class modulo m.

For example, $25 \equiv 18 \equiv 11 \equiv 4 \pmod{7}$.

2.3.5 Least Non-negative Residue

The *least non-negative residue* is a modulo operator on numbers. The notation $r = \langle x \rangle_m$ means $r = x - qm$ where $q = \lfloor x/m \rfloor$ is the integer part of x/m, and so r is the remainder part of x/m such that $r \in [0, m-1]$.

For example, $\langle 20 \rangle_7 = 6$.

2.4 Modular Arithmetic Operations

The following are definitions of the basic modular arithmetic operations of addition, subtraction, multiplication and 'division'.

2.4.1 Addition

Modular addition is performed as in 'normal' arithmetic addition, with the result usually denoted by the least non-negative residue.

For example, $3 + 5 \equiv 8 \equiv 1 \pmod{7}$. Or, using the alternative notation, $\langle 3 + 5 \rangle_7 = 1$.

2.4.2 Subtraction

Modular subtraction can be performed by noting that $-r \equiv m - r \pmod{m}$.

For example, $-3 \equiv 7 - 3 \equiv 4 \pmod{7}$. Or $\langle -3 \rangle_7 = 4$.

2.4.3 Multiplication

Modular multiplication is again performed as in 'normal' arithmetic multiplication.

For example, $3 \cdot 4 \equiv 12 \equiv 5 \pmod{7}$. Or $\langle 3 \cdot 4 \rangle_7 = 5$.

2.4.4 Division

Although the above three operations are all very straightforward, the operation of division in modular arithmetic is not always defined, and requires special constraints to be met in order for it to be possible.

Consider $x \cdot y \equiv z \pmod{m}$. If we want to find x then, using 'normal' arithmetic principles we would write $x \equiv z \cdot y^{-1} \pmod{m}$. But this implies that, given a y, there must exist an integer y^{-1} (called the *multiplicative inverse* of y modulo m) such that $y \cdot y^{-1} \equiv 1 \pmod{m}$. The problem is that given arbitrary y and m it is not always possible to find such an integer. Thus for 'division' by y to be possible modulo m, the multiplicative inverse of y must exist, and this occurs if and only if y and m are coprime. That is $\gcd(y, m) = 1$, where gcd() is the *greatest common divisor* function.

The gcd of two numbers can be computed by the *Euclidean algorithm*, and the multiplicative inverse of an integer with respect to a coprime modulus can be computed by the *extended Euclidean algorithm.*

2.5 Euclid's Algorithms

2.5.1 The Euclidean Algorithm

The *Euclidean algorithm* can be used to find the greatest common divisor of two integers.

It can be explained as follows (for strict proof see for example [24]).

Given two integers a and b with $a > b$ and $d = \gcd(a, b)$ then we have $d|a$ and $d|b$, and so also $d|(a \pm b)$. In fact $d|(a - b)$, $d|(a - 2b)$, ..., $d|(a - nb)$ for arbitrary integer n.

If now, given the two integers a and b, we want to find $d = \gcd(a, b)$, we can first calculate an integer $q = \lfloor a/b \rfloor$ so that

$$a - qb = r$$

where $r \in [0, b - 1]$. From the above we know that $d|(a - qb)$ and so also $d|r$. This means that we now have two integers b and r, each respectively less than a and b, for which $b > r$ and $d = \gcd(b, r)$. If we substitute these new numbers into a and b so that

$$a \leftarrow b$$

$$b \leftarrow r$$

and perform the calculation of q again, we see that we can keep going in this recursive fashion with a and b getting smaller with each iteration. If we end the process when $r = 0$ (before b is assigned the value of r), we will find that b is the greatest common divisor of the two original numbers.

Using the notation that $r(i)$ means the value of r at the end of the i-th iteration, we can restate Euclid's algorithm as:

Algorithm 1 (Euclidean Algorithm) *Given two integers a and b then to find $d =$* $\gcd(a, b)$ *do the following. Let*

$$r(-2) = a, \quad r(-1) = b$$

Then, starting at $i = 0$ *let*

$$r(i) \quad = \quad \langle r(i-2) \rangle_{r(i-1)}$$

until $r(k) = 0$ *for some* $k \geq 0$. *Then* $d = \gcd(a, b) = r(k-1)$.

Proof: In [24] ■[1].

2.5.2 The Extended Euclidean Algorithm

The *extended Euclidean algorithm* (sometimes known as the *modified Euclidean algorithm*) can be used to find the multiplicative inverse of an integer with respect to a coprime modulus. It will be stated as follows.

Algorithm 2 (Extended Euclidean Algorithm) *Given coprime integers* x *and* m *such that* $x < m$, *then to find the integer* x^{-1} *such that* $x \cdot x^{-1} \equiv 1 \pmod{m}$ *do the following. Let*

$$r(-2) = m, \;\; s(-2) = 0, \;\; r(-1) = x, \;\; s(-1) = 1$$

Then, starting at $i = 0$ *let*

$$\begin{aligned} q(i) &= \lfloor r(i-2)/r(i-1) \rfloor \\ r(i) &= r(i-2) - q(i) \cdot r(i-1) \\ s(i) &= s(i-2) - q(i) \cdot s(i-1) \end{aligned}$$

until $r(k) = 0$ *for some* $k \geq 0$. *Then* $x^{-1} = \langle s(k-1) \rangle_m$.

Proof: Can be found in [24] ■.

Note that, in the above, $\langle s(k-1) \rangle_m$ just means if $s(k-1) < 0$ then add m to get the result into the range $[0, m-1]$.

---

[1] The symbol ■ means 'end of proof'.

2.6 Fundamental Theorems

The theorems presented in this section are fundamental to the study of number theory, and as such their proofs can be found in many text books. The only proof given here is for Fermat's Little Theorem, the sole purpose of which is to give an overall impression of how theorems can be proven within this subject.

2.6.1 The Fundamental Theorem of Arithmetic

The *fundamental theorem of arithmetic* states that any integer can be uniquely expressed as the product of distinct powers of prime numbers. Since the expression is unique, it is known as the prime factorization of the integer.

Theorem 1 (Fundamental Theorem of Arithmetic) *Any positive integer m can be uniquely expressed as*

$$m = \prod_{i=0}^{n-1} (p_i)^{k_i}$$

where p_i are distinct primes, k_i is the power to which each p_i is raised and n is the number of distinct primes in the factorization of m.

Proof: Intuitively 'obvious', but proof can be found in [24] ■.

2.6.2 Fermat's Little Theorem

Before presenting Fermat's Little Theorem (FLT for short), we first need to understand the following two lemmas.

Lemma 2 *For p a prime and $a \not| p$, the numbers $a, 2a, 3a, \ldots, (p-1)a$ are all incongruent modulo p.*

Proof (by contradiction): Suppose $i \cdot a \equiv j \cdot a \pmod{p}$ for $1 \leq j < i \leq p-1$, then $(i-j) \cdot a \equiv 0 \pmod{p}$ and so, since p is prime, either $p|a$ or $p|(i-j)$. But the former is not possible by choice of a, and the latter is not possible since $0 < (i-j) < p$ ■.

Lemma 3 *For p a prime and $a \not| p$, the numbers $a, 2a, 3a, \ldots, (p-1)a$ are all incongruent to 0 modulo p.*

Proof (by contradiction): Suppose $k \cdot a \equiv 0 \pmod{p}$ for $1 \leq k \leq p-1$, then, since p is prime, either $p|a$ or $p|k$. But the former is not possible by choice of a, and the latter is not possible since $0 < k < p$ ■.

Theorem 4 (Fermat's Little Theorem) *For p a prime and a an integer such that $p \not| a$, then*

$$a^{p-1} \equiv 1 \ (mod\ p)$$

Proof: From the above lemmas we know that the numbers $a, 2a, 3a, \ldots, (p-1)a$ are all incongruent modulo p and that none of them are congruent to 0 modulo p. This must mean that the numbers $\{a,2a,\ldots,(p-1)a\}$ must be congruent to the numbers $\{1,2,\ldots,p\text{-}1\}$ in some order. Therefore

$$a \cdot 2a \cdot 3a \cdot \cdots \cdot (p-1)a \equiv 1 \cdot 2 \cdot 3 \cdot \cdots \cdot (p-1) \pmod{p}$$

so

$$a^{p-1} \cdot (p-1)! \equiv (p-1)! \pmod{p}$$

and since $\gcd(p, (p-1)!) = 1$, we can cancel the $(p-1)!$ terms in the above to obtain

$$a^{p-1} \equiv 1 \pmod{p}$$

■.

A generalisation of the above can be obtained for any integer a as follows.

Corollary 5 *For p a prime, and a any integer, then $a^p \equiv a \ (mod\ p)$.*

Proof: If $p \not| a$ then multiply both sides of FLT by a, else if $p|a$ then $a \equiv 0 \pmod{p}$ and the answer is trivial ■.

2.6.3 The Euler Totient Function

Let $\Phi(m)$ be the number of integers in the range $[0, m-1]$ that are coprime with the integer m; this is the *Euler Totient function* (sometimes known as the *Euler Phi function*). It has an intimate relationship with the prime factorization of m.

Theorem 6 (Euler Totient Function) *Given positive integer m, the expression for the Euler Totient function is*

$$\Phi(m) = \prod_{i=0}^{n-1} {p_i}^{k_i-1}(p_i - 1)$$

where the p_i, k_i and n are as in Theorem 1.

Proof: can be found in [24] ■.

For example, if $m = 539 = 7^2 \cdot 11$ then $\Phi(m) = 7 \cdot (7-1) \cdot (11-1) = 420$. That is, there are 420 numbers less than 539 that are coprime with 539. To be precise, the 420 numbers that are not multiples of 7 or 11.

2.6.4 Euler's Theorem

Euler's theorem is an extension of Fermat's Little Theorem to non-prime moduli.

Theorem 7 (Euler's Theorem) *Given coprime integers x and m, then*

$$x^{\Phi(m)} \equiv 1 \pmod{m}$$

Proof: in [24] ■.

A generalisation that considers non-coprime integers x and m can be stated as

Corollary 8 *For positive integers x and m and n, $x^{n \cdot \Phi(m)+1} \equiv x \pmod{m}$*

Proof: in [24] ■.

2.6.5 The Chinese Remainder Theorem

The *Chinese Remainder Theorem* (CRT) allows arithmetic operations modulo m to be performed using coprime divisors of m. For large m this can significantly speed up modulo m operations.

Theorem 9 (Chinese Remainder Theorem) *Given n coprime moduli $m_1, m_2, \ldots, m_n$ and setting*

$$m = \prod_{i=1}^{n} m_i$$

then any integer, $x \in [0, m-1]$ can be uniquely expressed as the n-tuple of residues of x modulo m_i. That is

$$x \equiv (x_1, x_2, \ldots, x_n) \pmod{m}$$

where $x_i = \langle x \rangle_{m_i}$.

Furthermore, x can be recovered from its n-tuple using

$$x = \left\langle \sum_{i=1}^{n} x_i \cdot \left\langle \hat{m}_i^{-1} \right\rangle_{m_i} \cdot \hat{m}_i \right\rangle_m$$

where

$$\hat{m}_i = \frac{m}{m_i}$$

Proof: in [24] ■.

2.7 The RSA Ciphersystem

The RSA ciphersystem is based upon the generalized version of Euler's Theorem. It relies upon the following observation; if e and d are two integers such that

$$e \cdot d \equiv 1 \pmod{\Phi(m)}$$

where m is a positive integer, then we have

$$e \cdot d = n \cdot \Phi(m) + 1$$

for some positive integer n, and so, by Corollary 8

$$(x^e)^d \equiv x \pmod{m}$$

What this equation is saying, is that, given a piece of information, x, this information can first be 'encrypted' (raised to the power e mod m), and then decrypted (raised to the power d mod m), and we will recover the original information, x. Since the integer e is the multiplicative inverse of d modulo $\Phi(m)$, and, in general, $e \neq d$, we have effectively just created a two-key cryptosystem.

The significance of the numbers d, e, $\Phi(m)$ and m with regard to security can be summarised as follows,

- integers d and e can only be derived once $\Phi(m)$ is known, and
- $\Phi(m)$ can only be calculated once the prime factorization of m is known.

Now, if m is a number such that, using current techniques, it is very hard to factor into its prime components, then it will be very difficult to derive integers d and e directly from m. The key to using all this in the context of a cryptosystem then, is for the implementor of the system to create the number m from the product of a few very large prime numbers. If he then creates the numbers $\Phi(m)$, d and e using his knowledge of the prime factors of m, and releases the numbers m and e to the public (but keeps the primes, $\Phi(m)$ and d secret), then it will be an extremely difficult task for a member of the public to recreate either the primes, $\Phi(m)$ or d from just m and e. For suitably large initial primes, the problem can be made to be infeasibly difficult.

2.7.1 Key Generation

Formalizing the above discussion, an RSA ciphersystem requires both *public* and *secret* keys – called the *RSA key-pair* – to encrypt and decrypt plaintext and ciphertext respectively. Each key consists of the modulus and an exponent. The public key consists of the

modulus and the public exponent, while the secret key consists of the modulus and the secret exponent. That is

$$k_p = \{e_p, m\}$$

$$k_s = \{e_s, m\}$$

where k_p and k_s are the public and secret keys, e_p and e_s are the public and secret exponents, and m is the modulus.

An RSA key-pair is typically generated as follows,

1. Generate two large primes, p and q, each greater than 75 digits (250 bits) in size, then calculate the modulus m, such that

$$m = p \cdot q$$

2. Calculate Euler's Totient $\Phi(m)$,

$$\Phi(m) = (p-1) \cdot (q-1)$$

3. Choose the public exponent e_p such that,

$$e_p \in [0, \Phi(m) - 1]$$

$$\gcd(e_p, \Phi(m)) = 1$$

4. Calculate the secret exponent e_s given that,

$$e_p \cdot e_s \equiv 1 \pmod{\Phi(m)}$$

using the extended Euclidean algorithm.

2.7.2 Secret Information Exchange

Assume that we have two people, Alice and Bob, and Alice wants to send some secret information to Bob. Assume also that Bob has previously generated an RSA key-pair

such that his public key, $k_p(B) = \{e_p(B), m(B)\}$, has been placed into some public list of keys that Alice has access to, and his secret key, $k_s(B) = \{e_s(B), m(B)\}$, is known only to Bob himself. Now, if the information that Alice wants to send to Bob is represented by the integer I where $I \in [0, m(B) - 1]$, then

1. Alice calculates encrypted data,

$$x = \left\langle I^{e_p(B)} \right\rangle_{m(B)}$$

2. Alice sends encrypted data, x, to Bob.

3. Bob decrypts data,

$$\begin{aligned} y &= \left\langle x^{e_s(B)} \right\rangle_{m(B)} \\ &= \left\langle I^{e_p(B) \cdot e_s(B)} \right\rangle_{m(B)} \\ &= \left\langle I^{n \cdot \Phi(m)+1} \right\rangle_{m(B)} \\ &= \langle I \rangle_{m(B)} \\ &= I \end{aligned}$$

and Bob now has the information, I.

Note that for an eavesdropper to recover the information I from x he would either have to know the secret exponent $e_s(B)$ or be able to perform the number-theoretic logarithm function on x which is similar in difficulty to factoring m.

2.7.3 Authentic Information Exchange

Assume now that Alice wants to send some information to Bob, but that this time the information is not secret. However, Bob wants to be sure that the information originated from Alice and not from anyone else. This is called authenticity of information and can be achieved as follows.

If Alice has previously generated a key-pair, $k_p(A)$ and $k_s(A)$, and placed $k_p(A)$ into a public registry, then authentic transfer can take place by Alice first 'encrypting' the information, I, with her secret key. Since no-one else knows Alice's secret key, the ciphertext thus produced will be the unique result of Alice and the information. In effect Alice has 'digitally signed' the document represented by I. Thus

1. Alice authenticates data,

$$x = \left\langle I^{e_s(A)} \right\rangle_{m(A)}$$

2. Alice sends authenticated data, x, to Bob.

3. Bob validates the authentication,

$$\begin{aligned} y &= \left\langle x^{e_p(A)} \right\rangle_{m(A)} \\ &= \left\langle I^{e_s(A) \cdot e_p(A)} \right\rangle_{m(A)} \\ &= \langle I \rangle_{m(A)} \\ &= I \end{aligned}$$

2.7.4 Secret and Authentic Information Exchange

The obvious problem with the method for authenticated information exchange presented above, is that, since $e_p(A)$ is public, persons other than Bob will be able to read the message, I, if they have access to the transported data, x. This can be easily prevented by combining the protocols for secret and authentic information exchange.

Assuming that both Alice and Bob have created and distributed their key-pairs, $k_s(A)$, $k_p(A)$, $k_s(B)$ and $k_p(B)$ respectively, then Alice can send secret information to Bob, that Bob knows must have originated from Alice, as follows.

1. Alice authenticates data,

$$x_1 = \left\langle I^{e_s(A)} \right\rangle_{m(A)}$$

2. Alice encrypts data,

$$x_2 = \left\langle x_1^{e_p(B)} \right\rangle_{m(B)}$$

3. Alice sends data, x_2, to Bob.

4. Bob decrypts data,

$$y_2 = \left\langle x_2^{e_s(B)} \right\rangle_{m(B)}$$

5. Bob validates the authentication,

$$y_1 = \left\langle y_2^{e_p(A)} \right\rangle_{m(A)} = I$$

2.7.5 Framing

In practice, the use of RSA to encrypt/authenticate messages of arbitrary length requires a process called *framing*. This process is simply the 'slicing' up of messages so that each slice, when combined with a small amount of protocol information, forms a 'block' which, when viewed as representing a number, has magnitude less than that of the modulus being used. In the case of secret-only or authentication-only information exchange protocols, these 'blocks' are then simply enciphered and sent to their destination. In the case of the secret-authentic protocol there are two moduli in use, and so the information must be re-blocked after the first encipherment to prepare it for the second pass.

In essence, the sender of encrypted messages must perform framing immediately before any RSA exponentiation, whilst the receiver of the messages must perform de-framing immediately after any exponentiation.

2.8 Summary

This chapter served as a brief introduction to cryptography, covering the definition of a ciphersystem and the difference between secret-key and public-key systems. This was followed by an overview of the main theorems and basic algorithms of modular arithmetic,

introducing the notation that will be used throughout this thesis. Finally, the RSA cipher-system was explained including key generation, ciphering for secrecy and authentication and the need to frame the plaintext into suitable blocks before performing the modular exponentiation cipher operation.

Chapter 3

Binary Arithmetic

In this chapter we will look at algorithms that permit the efficient calculation of long-integer arithmetic operations.

3.1 The Binary Representation of Numbers

A positive integer, X, may be represented in the binary number system as a k-bit *bit-vector*

$$X = [x_{k-1}, x_{k-2}, \ldots, x_0]$$

where $x_i \in \{0, 1\}$, and whose value is given by

$$X = \sum_{i=0}^{k-1} 2^i \cdot x_i \tag{3.1}$$

The right-hand side of the above expression can be thought of as a set of instructions for evaluating the integer X given the k-bit vector $[x_{k-1}, x_{k-2}, \ldots, x_0]$. This is therefore a kind of algorithm for finding X given its bit-vector representation.

Two other well-known [25] algorithms for the evaluation of binary integers are discussed in the next sections.

3.1.1 Right-to-Left Binary Evaluation Algorithm

An iterative algorithm can be defined to determine the value of an integer, X, given its bit-vector representation. Using the notation of the previous section, with $s(i)$ and $t(i)$

referring respectively to the value of s and t after the i-th iteration, then

Algorithm 3 *Given a positive binary integer, X, and setting*

$$s(0) = 0, \;\; t(0) = 1$$

then letting

$$\begin{aligned} s(i+1) &= s(i) + x_i \cdot t(i) \\ t(i+1) &= 2 \cdot t(i) \end{aligned}$$

on the k-th iteration we will have $s(k) = X$.

Proof: By noting that $t(i) = 2^i$ and expanding out the right-hand side of Equation 3.1, we have

$$\begin{aligned} X &= x_0 \cdot t(0) + x_1 \cdot t(1) + x_2 \cdot t(2) + \ldots + x_{k-2} \cdot t(k-2) + x_{k-1} \cdot t(k-1) \\ &= [x_0 \cdot t(0) + \ldots + x_{k-2} \cdot t(k-2)] + x_{k-1} \cdot t(k-1) \\ &= s(k-1) + x_{k-1} \cdot t(k-1) \\ &= s(k) \end{aligned}$$

■.

Note that this is called the *right-to-left* algorithm because the indexing, x_i, of the bit-vector starts at 0 and moves up towards $k-1$. That is from the least-significant-bit towards the most-significant-bit which, under normal binary notation, corresponds to starting at the right-hand end of the vector and working left.

3.1.2 Left-to-Right Binary Evaluation Algorithm

Again using the previous notation,

Algorithm 4 *Given a positive binary integer, X, set*

$$s(0) = 0$$

and let

$$s(i+1) \;=\; 2 \cdot s(i) + x_{k-i-1}$$

then on the k-th iteration it can also be shown that $s(k) = X$.

Proof: By applying the above equation k times, and then expanding out the recursion by 3 steps, we get

$$\begin{aligned} s(k) &= 2 \cdot s(k-1) + x_0 \\ &= 2 \cdot (2 \cdot s(k-2) + x_1) + x_0 \\ &= 2 \cdot (2 \cdot (2 \cdot s(k-3) + x_2) + x_1) + x_0 \\ &= 2^3 \cdot s(k-3) + 2^2 \cdot x_2 + 2^1 \cdot x_1 + 2^0 \cdot x_0 \end{aligned}$$

and so we see that, expanding the recursion α steps, gives

$$s(k) = 2^{\alpha} \cdot s(k-\alpha) + \sum_{i=0}^{\alpha-1} 2^i \cdot x_i$$

if we now set $\alpha = k$, then

$$\begin{aligned} s(k) &= 2^k \cdot s(0) + \sum_{i=0}^{k-1} 2^i \cdot x_i \\ &= \sum_{i=0}^{k-1} 2^i \cdot x_i \\ &= X \end{aligned}$$

■.

Similarly, this algorithm is called *left-to-right* because processing starts at the most-significant-bit and proceeds towards the least-significant-bit.

3.2 Binary Arithmetic

Having seen how a positive integer can be represented in binary bit-vector form, we now look at the fundamental algorithms for performing integer arithmetic on these numbers.

3.2.1 Addition

Given two positive k-bit integers, X and Y, whose values are

$$X = \sum_{i=0}^{k-1} 2^i \cdot x_i, \quad Y = \sum_{i=0}^{k-1} 2^i \cdot y_i$$

where $x_i, y_i \in \{0, 1\}$.

Then their sum, Z, can be expressed by

$$Z = X + Y = \sum_{i=0}^{k} 2^i \cdot z_i$$

where $z_i \in \{0, 1\}$ and so Z is a $(k+1)$-bit vector.

The value of each z_i must be calculated from $i = 0$ up to $i = k$ with

$$z_i = \langle x_i + y_i + c_{i-1} \rangle_2$$

where

$$c_i = \left\lfloor \frac{x_i + y_i + c_{i-1}}{2} \right\rfloor$$

and $x_k = y_k = c_{-1} = 0$.

If the above addition mechanism is viewed from a hardware perspective, then its implementation would take the form of a *ripple adder* as shown in Figure 3.1. To add two

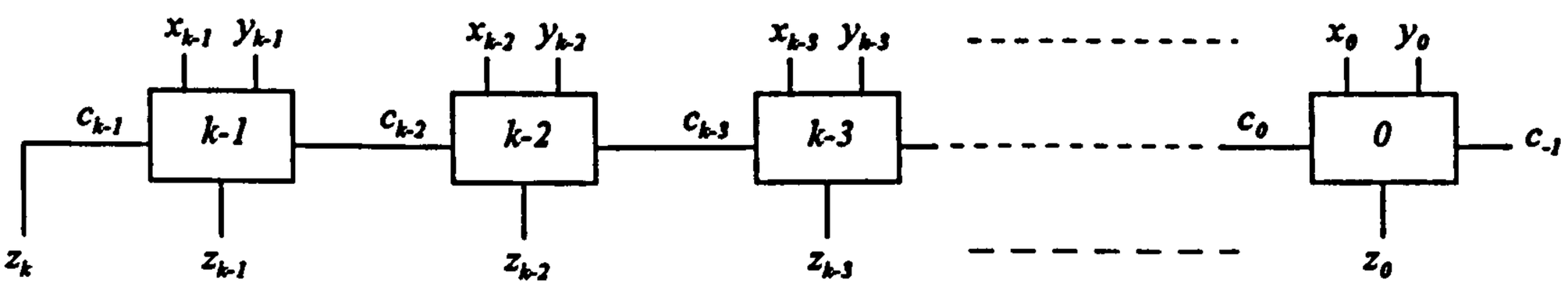


Figure 3.1: A k-bit ripple adder.

k-bit binary numbers the adder employs k 1-bit adders, and it can be seen that the z_i and c_i of the above equations correspond to the 'sum-out' and 'carry-out' signals respectively of the i-th adder unit.

3.2.2 Multiplication

Given two k-bit integers, X and Y, as in the previous section, their product, Z, can be expressed by

$$Z = X \cdot Y = \sum_{i=0}^{2k-1} 2^i \cdot z_i$$

where $z_i \in \{0,1\}$ and so Z is a $(2k)$-bit vector.

The product can be calculated by adding together all of the *partial products* formed by multiplying Y by each weighted bit of X. That is

$$Z = \sum_{i=0}^{k-1} 2^i \cdot x_i \cdot Y$$

and since each x_i is either 0 or 1, this is, in effect, adding together the left-shifted versions of Y that correspond to the bit positions where $x_i = 1$. When the multiplication is expressed in this way, X is considered to be the *multiplier* and Y the *multiplicand.* This is shown, for a 4x4-bit multiplication, in Figure 3.2.

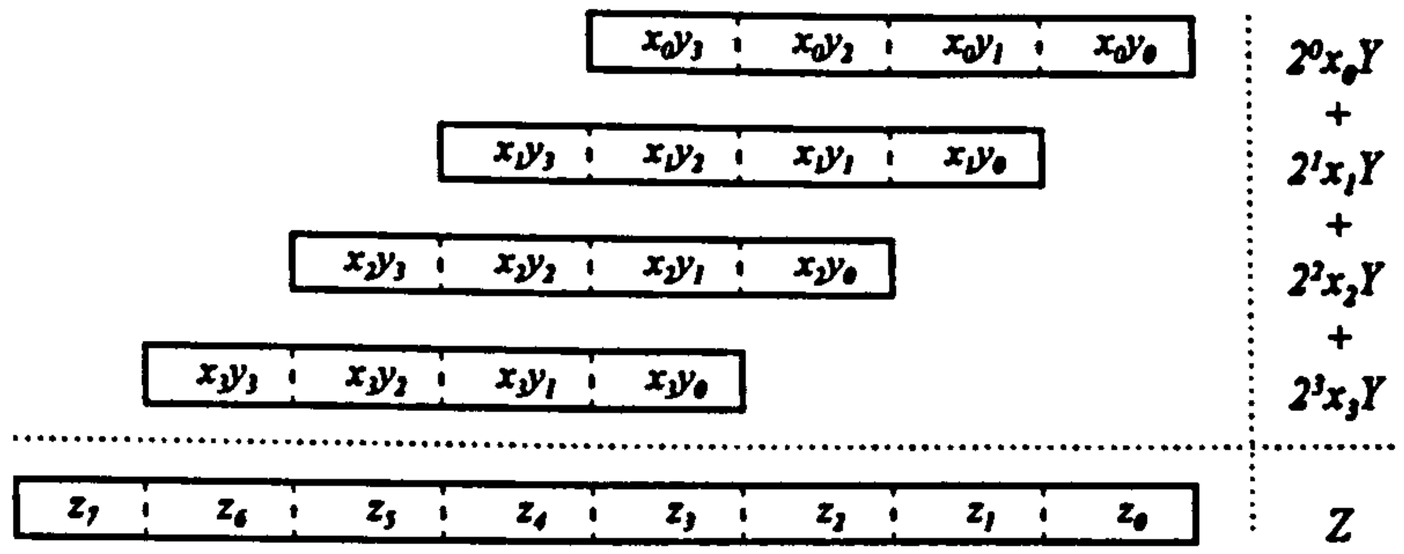


Figure 3.2: Multiplier partial products.

On comparison with the right-to-left and left-to-right binary evaluation algorithms of Section 3.1, we see that similar iterative algorithms can be found for multiplication.

Algorithm 5 (Right-to-Left Serial Multiplication) *If X and Y are two k-bit binary integers and $s(i)$ refers to the value of s after the i-th iteration, then setting*

$$s(0) = 0$$

and letting

$$s(i+1) = s(i) + 2^i \cdot x_i \cdot Y$$

then on the k-th iteration we will have $s(k) = X \cdot Y$.

Proof: Essentially the same as the proof of Algorithm 3.

$$\begin{aligned} X \cdot Y &= \left(2^0 \cdot x_0 + 2^1 \cdot x_1 + 2^2 \cdot x_2 + \ldots + 2^{k-2} \cdot x_{k-2} + 2^{k-1} \cdot x_{k-1}\right) \cdot Y \\ &= \left(2^0 \cdot x_0 \cdot Y + \ldots + 2^{k-2} \cdot x_{k-2} \cdot Y\right) + 2^{k-1} \cdot x_{k-1} \cdot Y \\ &= s(k-1) + 2^{k-1} \cdot x_{k-1} \cdot Y \\ &= s(k) \end{aligned}$$

■.

Algorithm 6 (Left-to-Right Serial Multiplication) *With* X *and* Y *as above, then setting*

$$s(0) = 0$$

and letting

$$s(i+1) = 2 \cdot s(i) + x_{k-i-1} \cdot Y$$

will give $s(k) = X \cdot Y$.

Proof: Similar to Algorithm 4. For $0 \leq \alpha \leq k$, then

$$s(k) = 2^{\alpha} \cdot s(k-\alpha) + \sum_{i=0}^{\alpha-1} 2^i \cdot x_i \cdot Y$$

which, for $\alpha = k$ gives

$$s(k) = \sum_{i=0}^{k-1} 2^i \cdot x_i \cdot Y = X \cdot Y$$

■.

3.2.3 Exponentiation

Consider the exponentiation of A to the power $E > 0$, so that

$$D = A^E$$

now if E is expressed as a k-bit bit-vector, then its value is given by

$$E = \sum_{i=0}^{k-1} 2^i \cdot e_i$$

and therefore

$$A^E = A^{2^{k-1} \cdot e_{k-1} + 2^{k-2} \cdot e_{k-2} + \cdots + 2^0 \cdot e_0} \tag{3.2}$$

$$= A^{2^{k-1} \cdot e_{k-1}} \cdot A^{2^{k-2} \cdot e_{k-2}} \cdots A^{2^0 \cdot e_0} \tag{3.3}$$

and the Right-to-Left and Left-to-Right algorithms can be modified to perform exponentiation as follows.

Algorithm 7 (Right-to-Left Exponentiation) *Given an integer A and a positive k-bit binary integer E, then setting*

$$s(0) = 1, \quad t(0) = A$$

and letting

$$s(i+1) = s(i) \cdot (t(i))^{e_i}$$

$$t(i+1) = (t(i))^2$$

results in

$$s(k) = A^E$$

Proof: Noting that $t(i) = A^{2^i}$, we have

$$s(i+1) = s(i) \cdot A^{2^i \cdot e_i}$$

which, when multiplied over $i = 0 \ldots k-1$, gives

$$s(k) = 1 \cdot A^{2^0 \cdot e_0} \cdot A^{2^1 \cdot e_1} \cdots A^{2^{k-1} \cdot e_{k-1}}$$

and so, on comparison with the right-hand-side of Equation 3.3

$$s(k) = A^E$$

■.

Algorithm 8 (Left-to-Right Exponentiation) *Given an integer A and a positive k-bit integer E, then setting*

$$s(0) = 1$$

and letting

$$s(i+1) = (s(i))^2 \cdot A^{e_{k-i-1}}$$

results in

$$s(k) = A^E$$

Proof: Using a similar technique to the proof of Algorithm 4, we have

$$\begin{aligned} s(k) &= (s(k-1))^2 \cdot A^{e_0} \\ &= \left((s(k-2))^2 \cdot A^{e_1}\right)^2 \cdot A^{e_0} \\ &= \left(\left((s(k-3))^2 \cdot A^{e_2}\right)^2 \cdot A^{e_1}\right)^2 \cdot A^{e_0} \\ &= (s(k-3))^8 \cdot A^{4 \cdot e_2} \cdot A^{2 \cdot e_1} \cdot A^{1 \cdot e_0} \end{aligned}$$

thus in general

$$s(k) = (s(k-\alpha))^{2^\alpha} \cdot \prod_{i=0}^{\alpha-1} A^{2^i \cdot e_i}$$

and setting $\alpha = k$ gives

$$\begin{aligned} s(k) &= (s(0))^{2^k} \cdot \prod_{i=0}^{k-1} A^{2^i \cdot e_i} \\ &= \prod_{i=0}^{k-1} A^{2^i \cdot e_i} \\ &= A^E \end{aligned}$$

■.

Note that in the above algorithms, the term A^{e_i} is evaluated simply as

$$A^{e_i} = \begin{cases} 1 & \text{if } e_i = 0 \\ A & \text{if } e_i = 1 \end{cases}$$

and so, assuming uniformly distributed k-bit exponents, the average number of multiplications required by the algorithms is $\frac{3k}{2}$.

3.3 Modular Binary Arithmetic

Modular arithmetic algorithms for addition, multiplication and exponentiation can be created by modifying the previous algorithms so that the results are reduced to their least non-negative residue modulo N.

i.e. Addition of two numbers X and Y modulo N, where $X, Y \in [0, N-1]$, can be performed by adding the X and Y and then subtracting N if the result is greater than or equal to N.

3.3.1 Modular Multiplication

There are two distinct methods of performing the modular multiplication of two numbers X and Y modulo N.

The first is to compute the product $T = X \cdot Y$ and then reduce T by dividing it by N and keeping the remainder. The disadvantage with this technique is that the product, T, is twice the size of the arguments X and Y, and thus requires extra storage and manipulation space. This is especially critical in hardware with the large word-sizes used in RSA calculations.

The second method involves modifying the previously stated iterative multiplication algorithms so that the product $\langle X \cdot Y \rangle_N$ is computed directly with all intermediate results in the range $[0, N-1]$.

Modifying the Right-to-Left algorithm leads to the following.

Algorithm 9 (Right-to-Left Modular Multiplication) *If X and Y are two positive k-bit integers less than N, and $s(i)$ refers to the value of s after the i-th iteration, then setting*

$$s(0) = 0$$

and letting

$$s(i+1) = \left\langle s(i) + 2^i \cdot x_i \cdot Y \right\rangle_N$$

then on the k-th iteration we will have $s(k) = \langle X \cdot Y \rangle_N$.

Examining the right-hand-side of the $s(i+1)$ calculation however, reveals the need to perform a modular reduction of $2^i \cdot Y$ where $i = 0 \ldots k-1$. Since $2^i \cdot Y$ can be a very large integer for high values of i, this calculation involves a division by N, and so therefore is no better than the separate multiplication/division approach.

A much better solution is obtained by modifying the Left-to-Right algorithm.

Algorithm 10 (Left-to-Right Modular Multiplication) *With X, Y and N as above, then setting*

$$s(0) = 0$$

and letting

$$s(i+1) = \langle 2 \cdot s(i) + x_{k-i-1} \cdot Y \rangle_N$$

will give $s(k) = \langle X \cdot Y \rangle_N$.

Here we see that the calculation of $s(i+1)$ involves the reduction of a number with upper bound less than $3N$ (since $s(i), Y < N$) for any i. Therefore only N or $2N$ need be subtracted during each iteration of the algorithm, and hence this algorithm lends itself most easily to hardware implementations. Indeed, most existing long-word modular multipliers use this algorithm as a basis from which to develop more efficient, faster algorithms.

3.3.2 Modular Exponentiation

Modular exponentiation algorithms can be derived from non-modular exponentiation algorithms simply by replacing the non-modular multiplications with modular ones. For completeness, they are stated below.

Algorithm 11 (Right-to-Left Modular Exponentiation) *Given an integer A, a positive k-bit integer E and modulus N, then setting*

$$s(0) = 1, \quad t(0) = A$$

and letting

$$\begin{aligned} s(i+1) &= \langle s(i) \cdot (t(i))^{e_i} \rangle_N \\ t(i+1) &= \left\langle (t(i))^2 \right\rangle_N \end{aligned}$$

results in

$$s(k) = \left\langle A^E \right\rangle_N$$

Algorithm 12 (Left-to-Right Modular Exponentiation) *Given an integer A, a positive k-bit integer E and modulus N, then setting*

$$s(0) = 1$$

and letting

$$s(i+1) = \left\langle \left\langle (s(i))^2 \right\rangle_N \cdot A^{e_{k-i-1}} \right\rangle_N$$

results in

$$s(k) = \left\langle A^E \right\rangle_N$$

3.4 Summary

This chapter reviewed the binary representation of positive integers and explored fundamental right-to-left and left-to-right binary evaluation techniques. These techniques were subsequently used to derive algorithms for iterative multiplication and exponentiation. These algorithms were then modified to perform standard modular operations, in particular showing that standard modular multiplication is inherently a left-to-right process.

Chapter 4

Arithmetic Hardware

In this chapter we will look at digital hardware techniques that allow for the efficient implemention of long-integer arithmetic operations.

4.1 Adders

There are numerous ways in which the addition of two positive k-bit numbers can be performed in hardware. In this section we will look at three of the most common methods applicable to long-integer addition, and review their design trade-offs in terms of circuit complexity and operational speed.

4.1.1 Ripple Adder

The *ripple adder* is the simplest of adder logic circuits and, as may be expected, also the slowest. Two k-bit positive integers can be added with k one-bit full adders as was shown in Figure 3.1. The circuit diagram for a one-bit full adder is shown in Figure 4.1.

Letting Δ_{FA} represent the time required for a full adder to generate both the sum-out and carry-out signals, then the time needed to add two k-bit numbers will be $k \cdot \Delta_{FA}$.

In most technologies, if the adder is implemented as in Figure 4.1, the time required to generate the carry-out signal will be slightly less than that required to generate the

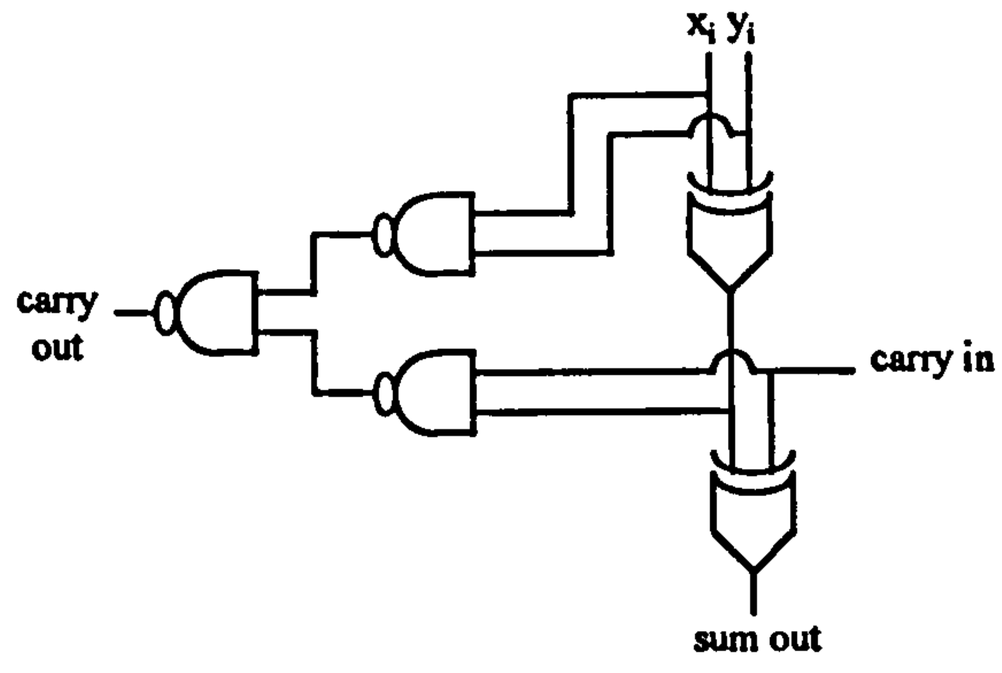


Figure 4.1: One-bit full adder (FA).

sum-out signal. This is because, usually, $\Delta_{XOR} \approx 3 \cdot \Delta_{NAND}$. However, in trying to allow for both differing technologies and differing adder implementations, it will be assumed throughout this thesis that Δ_{FA} applies to both sum-out and carry-out signals and that $\Delta_{FA} = 2 \cdot \Delta_{XOR}$.

Furthermore, it will be assumed that in any reasonably complex design primitive – such as a full adder or flip-flop – there is no penalty involved in taking the inverted value of a signal as opposed to its true value. The justification for this approach is that, since the implementation of any particular design primitive will vary from one manufacturer's technology to another, it can easily happen that a particular signal is generated true in one technology and inverted in another. In practice, there is usually a way to speedily generate the desired true or inverted signal in any primitive of a few gates or more. This is true particulary in CMOS where the basic transistor unit is the p-type/n-type complementary pair.

4.1.2 Carry-Completion Adder

Although the ripple adder is simple and uses relatively little circuit area, it is very slow. Its main drawback is that it must always wait for the worst-case carry propagation from the 0-th FA to the $(k-1)$-th FA, irrespective of whether any such carry is actually generated. Analysis has shown [25] that the average length of propagated carries when adding two randomly chosen k-bit integers is of the order of $\log_2 k$. The *carry-completion* adder

takes advantage of this fact by incorporating extra circuitry within the adder to detect when all carries have fully propagated. A simple block diagram is shown in Figure 4.2. With reference to the diagram, the two numbers to be added, X and Y, are placed onto

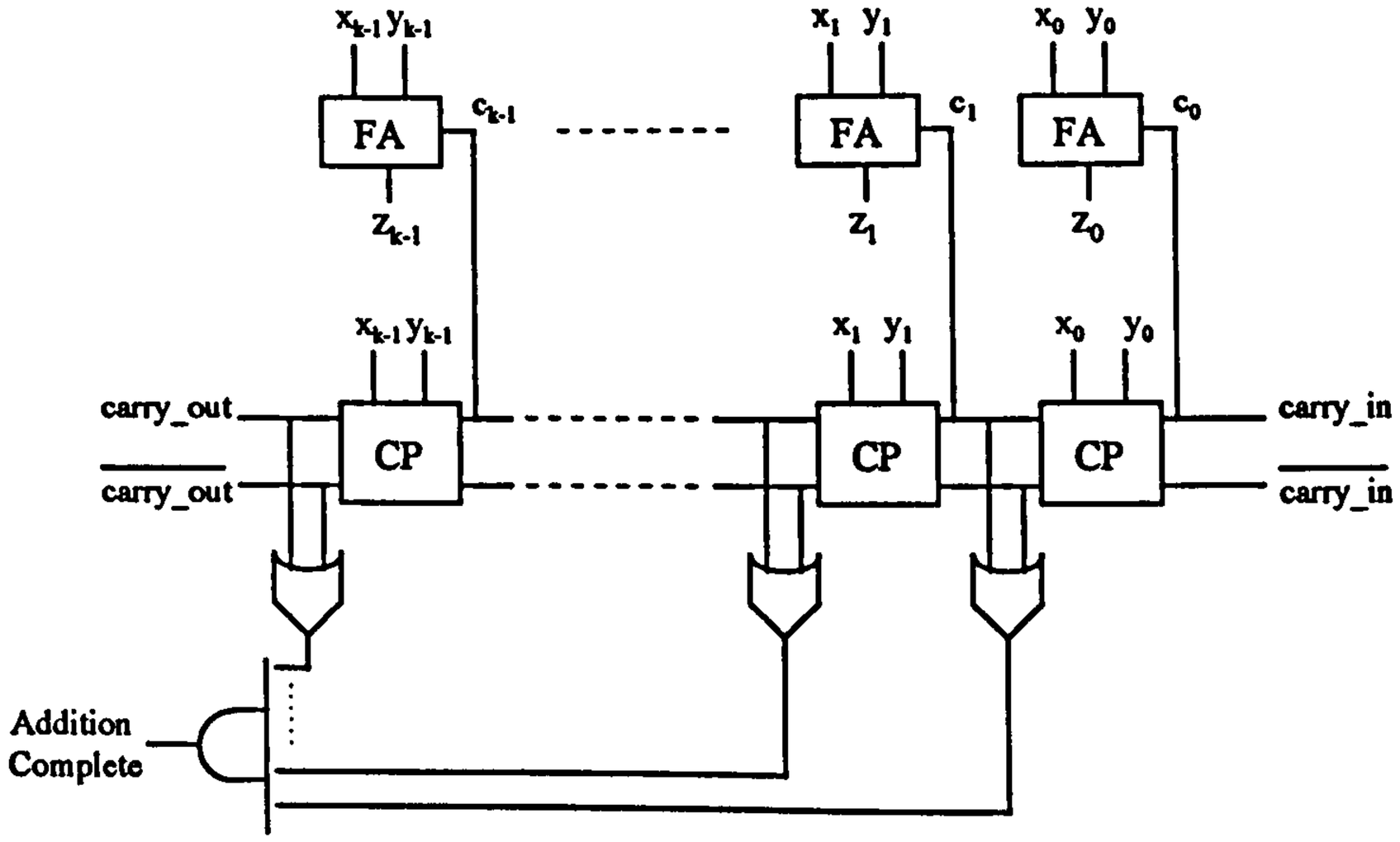


Figure 4.2: Carry-completion adder.

the inputs of the CP (Carry Propagate) cells where an intermediate carry vector, C, is generated. A CP cell generates its true and inverted carry-out signals [26] such that the signals will be mutual inverses only when all of the carries from the preceeding CP cells have propagated past the current cell. Carry propagation is thus monitored by the k OR gates and, once propagation has ceased, the k-input AND gate goes active indicating that the first stage of the addition (the generation of C) is complete. The second stage may now commence which is simply the bitwise addition (modulo 2) of the vectors X, Y and C.

In practice, the two stages are allowed to complete simultaneously, and the result becomes stable at the same time as the AND gate goes active. Assuming the delay of a CP cell is the same as that of an FA, we have the *average* time required to add two k-bit numbers as $\Delta_{FA} \cdot \log_2 k$. Note however that the complexity of this adder is roughly twice that of the ripple adder.

4.1.3 Carry-Select Adder

Although the carry-completion adder is, on average, faster than the ripple adder, it suffers from the disadvantage of a variable-length addition time. This makes the control circuitry surrounding the adder more complex than it would be for a fixed-time addition. The *carry-select adder*, on the other hand, is a fast, fixed-time adder circuit.

The carry-select adder works by partitioning the k-bit bit-vectors X and Y into vectors composed of b-bit sub-blocks. i.e. if $k = l \cdot b$, then using vector notation we have

$$X = [X_{l-1}, X_{l-2}, \ldots, X_0]$$

where now $X_i \in [0, 2^b - 1]$ is a b-bit sub-block. The value of X is given by

$$X = \sum_{i=0}^{l-1} 2^{ib} \cdot X_i$$

Initially, the sub-blocks of each vector are treated as individual sections - ignoring the relationship that each sub-block has with its neighbours. Each sub-block of the X vector is then involved in two simultaneous additions with its corresponding sub-block in the Y vector. One addition is performed with the carry-in of the sub-block adder active, and the other with it inactive. Since these additions are performed in parallel for all sub-blocks, it is not known at the time of the addition whether the carry input of any particular block (except for the lowest block) should be high or low. Thus we have to select, after the addition and for each sub-block, which of the two additions is the correct one. This selection is based on the carry-out of the previous sub-block addition. Figure 4.3 shows a simplified diagram of one sub-block of a carry-select adder. Assuming that each Sub-Block Adder of Figure 4.3 is implemented as a b-bit ripple adder, then the time needed to add two $k = l \cdot b$ numbers is approximately $b \cdot \Delta_{FA} + l \cdot \Delta_{MUX2TO1}$. The complexity of the carry-select adder is typically around twice that of the ripple adder [27].

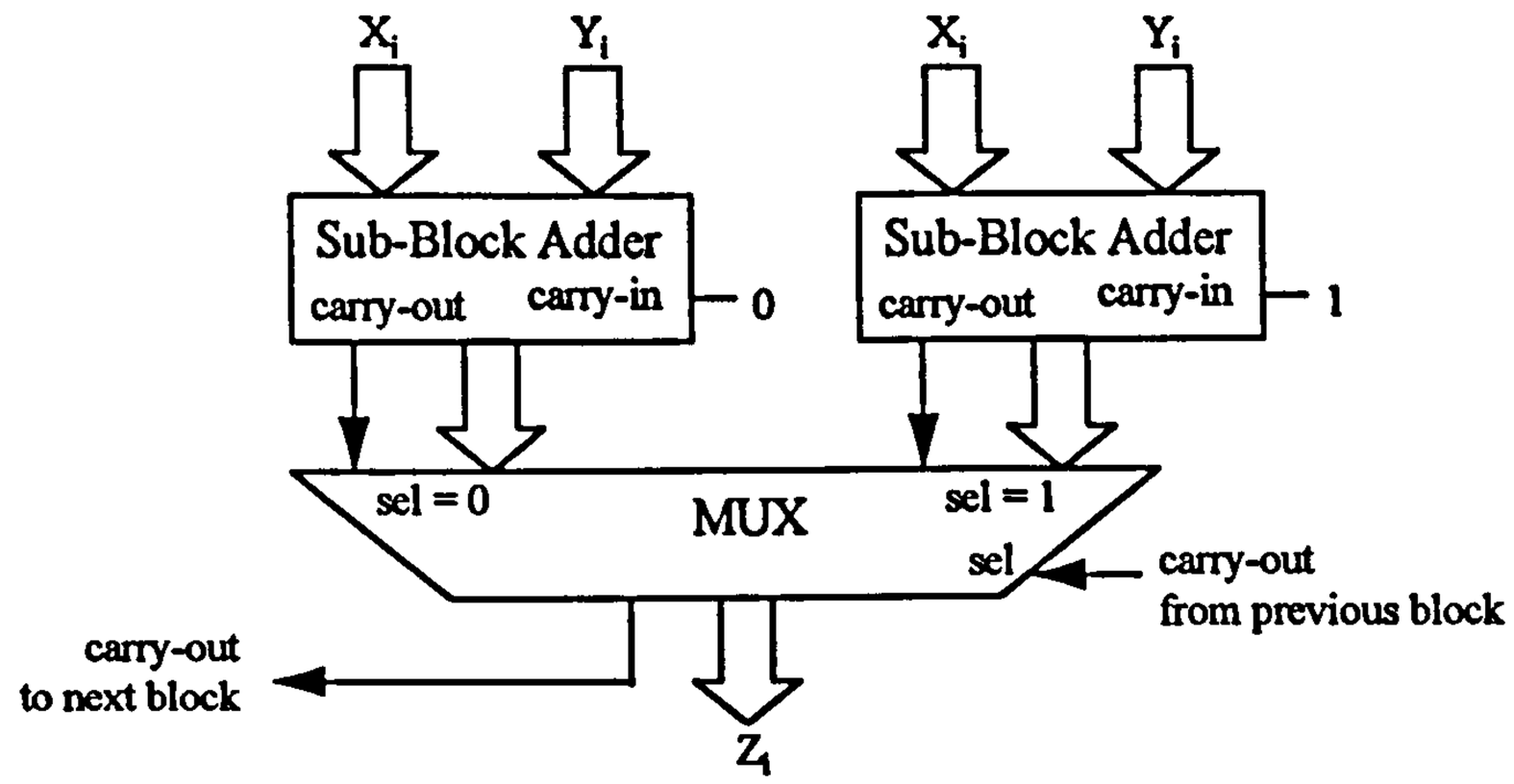


Figure 4.3: Carry-select adder sub-block.

4.2 Iterative Multipliers

With reference to the iterative multiplication algorithms of the previous chapter (Algorithms 5 and 6) we see that multiplication can be implemented as repeated addition. This leads to the generic hardware implementation of a multiplier in Figure 4.4. Here we have a simple adder/accumulator circuit, where the product of two positive numbers,

$$X = \sum_{i=0}^{k-1} 2^i \cdot x_i, \quad Y = \sum_{i=0}^{k-1} 2^i \cdot y_i$$

is calculated by the accumulated summation of the partial products $x_i \cdot Y$ for $i = 0 \ldots k-1$.

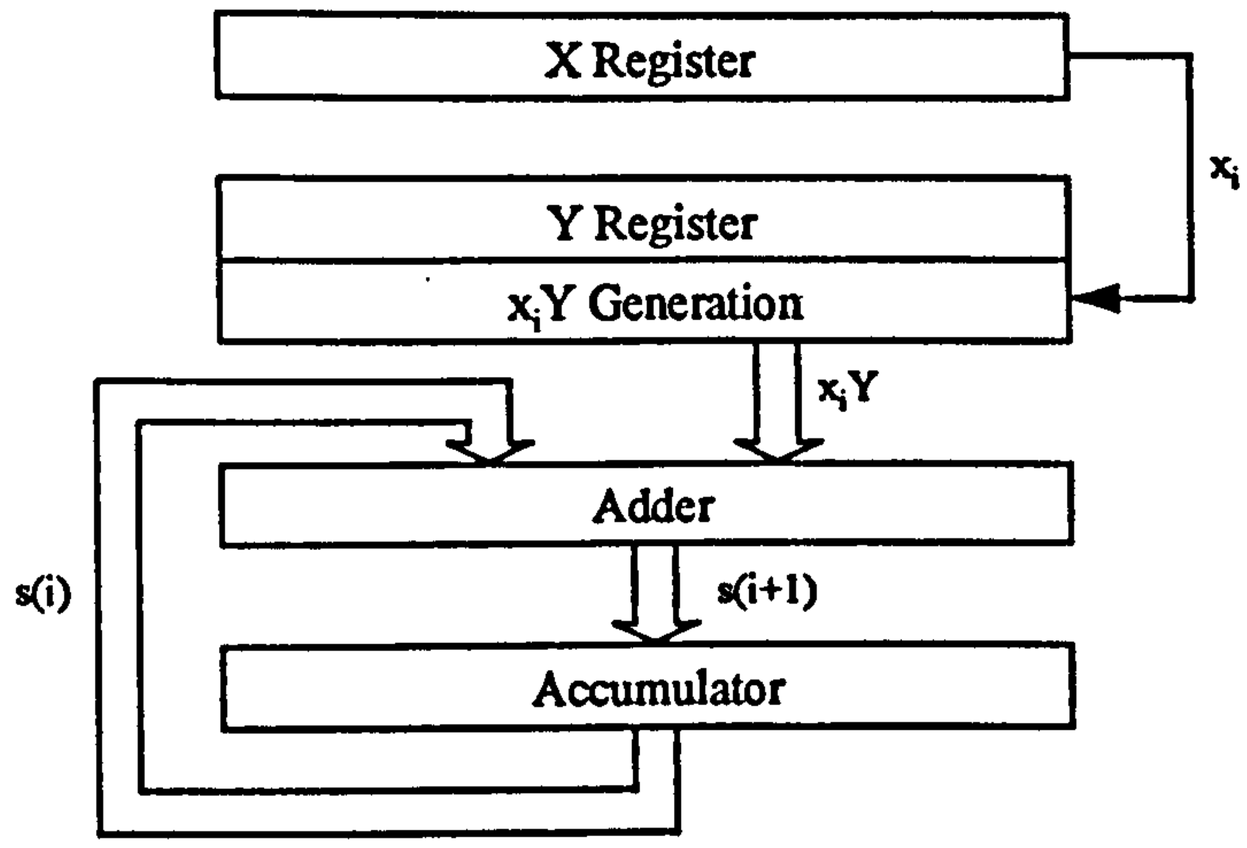


Figure 4.4: Generic iterative multiplier.

An implementation of the Right-to-Left multiplication algorithm (Algorithm 5) for example, would require an adder/accumulator configuration that, initially resets the ac-

cumulator to zero, then for each clock cycle $i = 0 \ldots k-1$, adds $x_i \cdot Y$ to the contents of the accumulator right-shifted by one bit position. The lower k-bits of the product are the k bits that were shifted out of the accumulator during cycles 0 to $k-1$. The upper k bits are those that are left in the accumulator.

The main problem with this circuit is the time taken to add the partial products $x_i \cdot Y$ during each cycle. Using any of the adders of Section 4.1 would lead to multiplication times proportional to k^α where $1 < \alpha \leq 2$. This is because the addition times of each of the above methods depend on the length, k, of the operands. To remove this power relationship on multiplication speed, and make it linear with k, a special multi-operand adder called a *carry-save adder* is used.

4.2.1 Carry-Save Adder

The carry-save adder is a fast, constant-time multi-operand adder suitable for implementing iterative multipliers. As its name suggests, the carries generated when adding numbers are not propagated but are 'saved'. In other words, all the partial products of a multiplication can be added in carry-save form, with carry-propagation performed only at the end of processing.

The circuit diagram for a carry-save adder (CSA) is shown in Figure 4.5. The circular

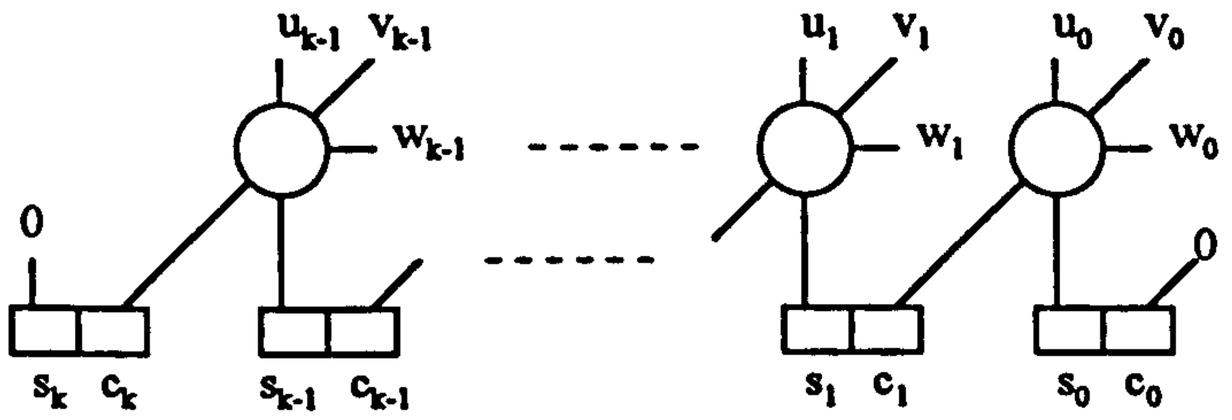


Figure 4.5: Carry-save adder.

components in this diagram are one-bit full adders and the rectangular components at the bottom of the diagram represent clocked flip-flops. Thus three k-bit numbers, U, V and

W,

$$U = \sum_{i=0}^{k-1} 2^i \cdot u_i, \quad V = \sum_{i=0}^{k-1} 2^i \cdot v_i, \quad W = \sum_{i=0}^{k-1} 2^i \cdot w_i$$

can be added together to form a $(k+1)$-digit result, where each digit of the result consists of the two-bit combination $s_i + c_i$. In other words,

$$U + V + W = \sum_{i=0}^{k} 2^i \cdot (s_i + c_i)$$

Another way of looking at the result is as the sum of two distinct bit-vectors; the sum vector S, and the carry vector C. Thus

$$U + V + W = S + C$$

where

$$S = \sum_{i=0}^{k} 2^i \cdot s_i, \quad C = \sum_{i=0}^{k} 2^i \cdot c_i$$

The big advantage of the CSA is that the addition of $U + V + W$ to produce S and C takes a constant time equal to the delay of a single full adder. In other words, addition time is unrelated to operand size.

The CSA can be used iteratively to add multiple numbers by feeding back the S and C outputs to the U and V inputs and adding successive bit-vectors via the W inputs. Thus on successive clock cycles the accumulated partial result held in the flip-flops at the bottom of the diagram is added to a new input operand W and the sum again stored in this register. At the end of this multi-operand addition process the S and C vectors can be added together, using one of the carry-propagation adders of Section 4.1, so that the result will be in conventional binary bit-vector form.

Using the CSA to implement the Right-to-Left multiplication algorithm, for example, would require, as stated above, that the partial products $x_i \cdot Y$ be added to the right-shifted contents of the accumulator. The partial products are generated by the AND gate circuitry of Figure 4.6. The right-shift can be accomplished efficiently by 'hardwiring' it into the feedback of the S and C vectors. Thus the following equations would describe

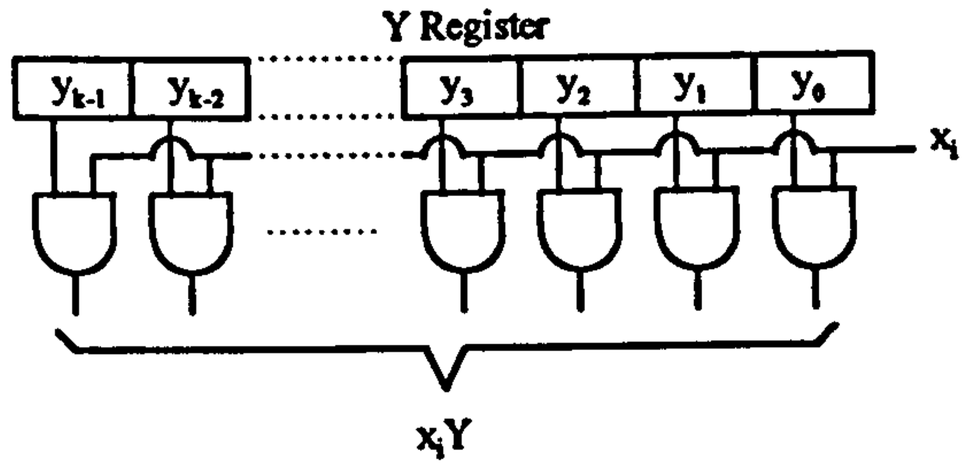


Figure 4.6: Generating $x_i \cdot Y$.

the connections from S and C back to U and V,

$$u_i \leftarrow s_{i+1}$$

$$v_i \leftarrow c_{i+1}$$

Note that bits s_0 and c_0 are effectively shifted out of the accumulator on each iteration of the algorithm. Since these bits form the least-significant k bits of the result they must be saved into a k bit shift register operating along-side the CSA as the calculation progresses. Moreover, examining the circuit diagram of the CSA shows us that the c_0 bit is always '0'. Therefore only the shifted out s_0's need be saved and these directly form the lower k bits of the result. The upper k bits of the result are formed from the carry-propagated addition of the S and C vectors.

Using the property that $c_0 = 0$ always, we can perform a k-bit multiplication obtaining the result in standard binary bit-vector form without performing a post-processing carry-propagate addition simply by allowing the CSA to cycle through another k iterations without adding any partial products into the accumulator. This works because the k extra cycles will produce a $2k$-bit vector from the shifted out s_0's. Since the product of two k-bit vectors can be expressed in at most $2k$-bits, then the shifted out $2k$-bit vector must be this result. Of course, the disadvantage of using this technique is that twice as many iterations are required to complete the multiplication, but it does have the advantage of not requiring a carry-propagate adder, and so uses less hardware.

The time required to perform a multiplication using CSA hardware is the sum of the times required for the iterative partial product summation and the time required for the

carry-propagated addition of the CSA S and C vectors. The latter is obviously dependent upon the type of adder used for the carry-propagated addition, but the former can be approximated by the product of the number of iterations required and the time required for each iteration. They are, respectively,

$$\text{Number of iterations} = k$$

$$\text{Iteration time} = \Delta_{AND} + \Delta_{FA} + \Delta_{FF}$$

where, in this context, Δ_{FF} corresponds to the 'setup' and delay time required by the flip-flops that make up the accumulator register.

The circuit complexity can be approximated by

$$\text{Number of bitslices} = k + 1$$

$$\text{Bitslice complexity} = \Omega_{AND} + \Omega_{FA} + 4 \cdot \Omega_{FF}$$

where the Ω notation measures the gate-count of the design primitives. Note that in the above bitslice complexity measure, the $4 \cdot \Omega_{FF}$ term reflects the fact that four flip-flops are required per bitslice for the X, Y, S and C vectors.

4.2.2 High-Radix Iterative Multiplication

The multipliers of the previous section have been of the radix-2 type. That is simple binary multipliers where the creation of partial products has been performed for one bit of the multiplier operand, X, at a time. High-radix multipliers generate partial products by looking at b-bits of X at a time, and so are radix-2^b multipliers.

A positive k-bit integer X can be viewed as a radix-2^b vector consisting of l blocks of b bits each. That is an l-digit vector

$$X = [X_{l-1}, X_{l-2}, \ldots, X_0]$$

where each digit $X_i \in [0, 2^b - 1]$ and $l = \lceil k/b \rceil$. The value of X is given by

$$X = \sum_{i=0}^{l-1} 2^{ib} \cdot X_i$$

Each radix-2^b digit of X can be expressed as

$$X_i = \sum_{j=0}^{b-1} 2^j \cdot X_{i,(j)}$$

where $X_{i,(j)}$ is the j-th bit of the i-th digit and corresponds to the $(ib+j)$-th bit of X, that is

$$X_{i,(j)} = x_{ib+j}$$

The product of two radix-2^b vectors, X and Y, can be expressed as

$$X \cdot Y = \sum_{i=0}^{l-1} 2^{ib} \cdot X_i \cdot Y$$

For example, the Right-to-Left multiplication algorithm becomes

$$\begin{aligned} s(i+1) &= 2^{ib} \cdot X_i \cdot Y \\ &= 2^{ib} \left(2^{b-1} \cdot X_{i,(b-1)} \cdot Y + 2^{b-2} \cdot X_{i,(b-2)} \cdot Y + \cdots + 2^0 \cdot X_{i,(0)} \cdot Y \right) \end{aligned}$$

where, when implemented in CSA hardware, the 2^{ib} term on the right-hand side of the above equation corresponds to a right-shift of the accumulator by b bits, and each partial product of the form $2^j \cdot X_{i,(j)} \cdot Y$ corresponds to a horizontal line of full adders so that this partial product may be added to the accumulated product.

For example, with $b = 2$, the partial products are generated two at a time and a 2-level CSA network is required to sum these products. This is shown in Figure 4.7 where the

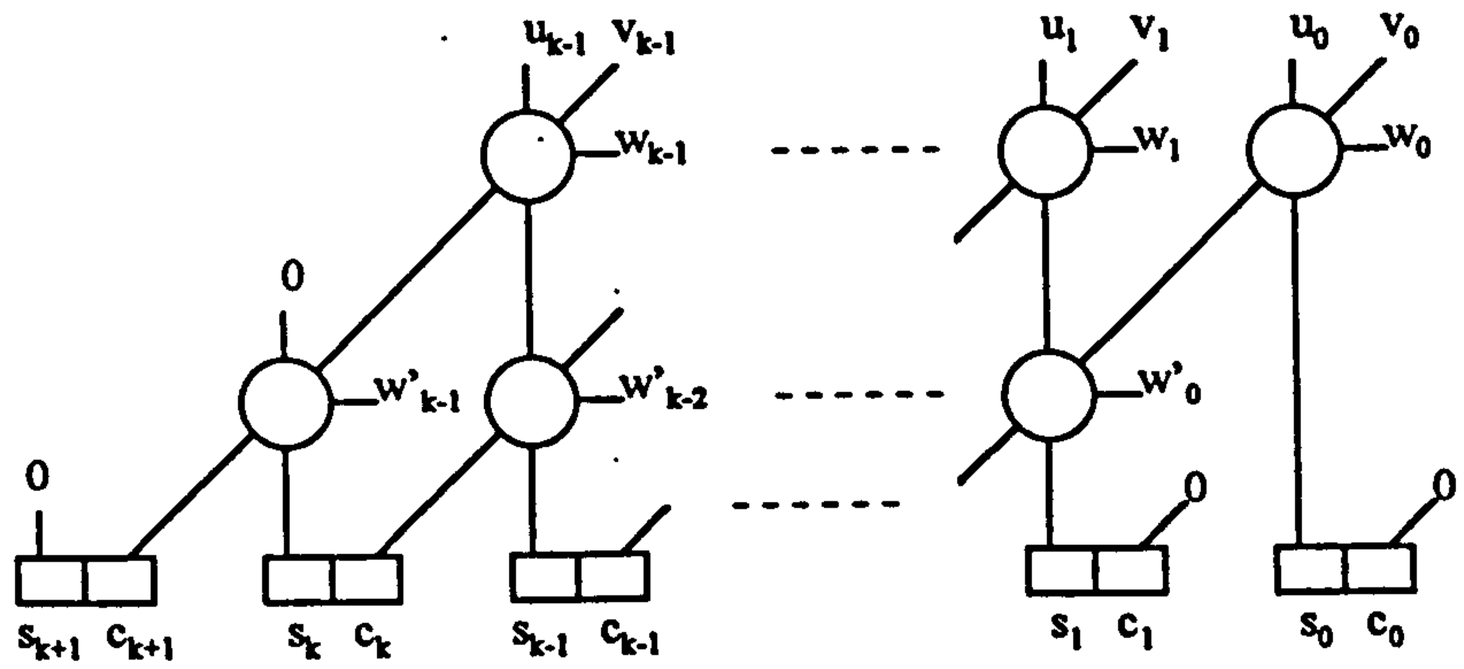


Figure 4.7: A 2-level CSA multiplier.

input vectors W and W' correspond to

$$W = X_{i,(0)} \cdot Y, \quad W' = X_{i,(1)} \cdot Y$$

and the feedback from the accumulator outputs to the adder inputs, incorporating a right-shift by 2 bits is defined by

$$u_i \leftarrow s_{i+2}$$

$$v_i \leftarrow c_{i+2}$$

The time required for the summation of the partial products, assuming all $X_{i,(j)}$ bits are available immediately following the accumulator's active clock edge, is the product of

$$\text{Number of iterations} = \lceil k/b \rceil$$

$$\text{Iteration time} = \Delta_{AND} + b \cdot \Delta_{FA} + \Delta_{FF}$$

The complexity is

$$\text{Number of bitslices} = k + b$$

$$\text{Bitslice complexity} = b \cdot \Omega_{AND} + b \cdot \Omega_{FA} + 4 \cdot \Omega_{FF}$$

Adder Network Optimisation

For multi-level CSA adder networks there are various methods that can be employed to speed up the propagation of signals through the adder levels.

One method involves the 'collapsing' of successive adder levels into a single, larger adder, and then performing logic optimization of this larger adder. For example, in a 2-level CSA design, the 'upper' and 'lower' full adders can be collapsed into a single larger adder. To see how this works we must think of a one-bit full adder as a special case of a more general class of adders. The one-bit full adder can be called a 3:2 adder because it 'compresses' three input signals of equal weight down to two output signals of differing weights. That is, with the input signals denoted as u_i, v_i and w_i, and the outputs as s_i and c_{i+1}, then the adder obeys the following equation

$$u_i + v_i + w_i = 2c_{i+1} + s_i$$

Two 3:2 adders connected as in a 2-level CSA tree can be collapsed down into a single 5:3 adder (also known as a 4:2 compressor or a 4:2 counter) as shown in Figure 4.8. Here

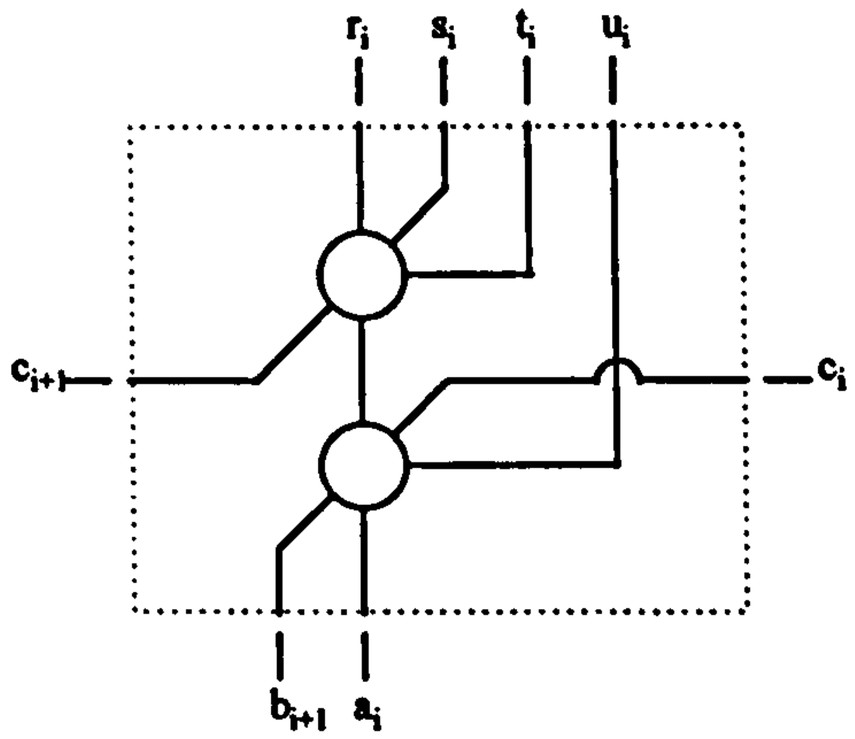


Figure 4.8: A 5:3 adder cell.

the inputs and outputs obey the equation

$$r_i + s_i + t_i + u_i + c_i = 2c_{i+1} + 2b_{i+1} + a_i$$

In the diagram, the 5:3 adder is constructed directly from two 3:2 adders and so obviously has the same function, but in practice the internals of the 5:3 adder can be redesigned so that they retain the same function but offer a faster implementation. Various designs are possible (see [28] and [29]) but the advantages they give in operational speed are generally offset by the added complexity of the design. e.g. a 50% increase in speed accompanied by a similar increase in circuit area. Higher order adders can be created in this way (i.e. 7:4, 9:5 etc.) but to be advantageous in a VLSI device would require careful full-custom design.

Another method for adder network optimization that is useful for networks of 4 levels and upwards is the optimization of adder interconnections. For example, with a 4-level network, Figure 4.9 shows how the interconnections between 3:2 adders can be modified so that the delay through the network is only $3\Delta_{FA}$ instead of $4\Delta_{FA}$. This type of optimization can also be applied to networks of 5:3 adders, 7:4 adders etc. when a high number of levels is to be implemented.

The final method of adder network optimization to be examined here is that of a

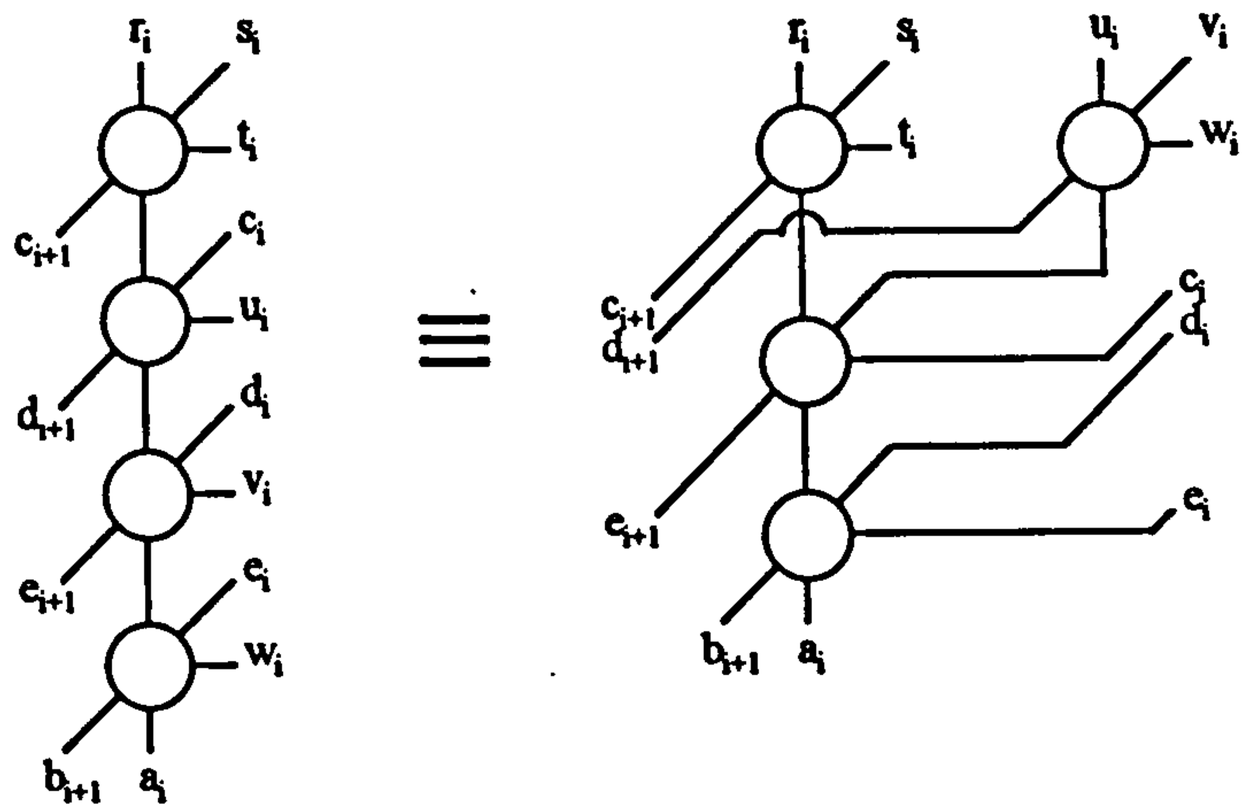


Figure 4.9: Adder interconnect optimization.

pipelined implementation of the separate levels. The basic strategy is to break up multi-level adder networks by placing latches between the levels. This has the effect of dramatically reducing the time required for each iteration of a multiplication but at the expense of increasing the hardware complexity and increasing the total number of iterations required.

A conceptual diagram of a pipelined, iterative multiplier is shown in Figure 4.10. In

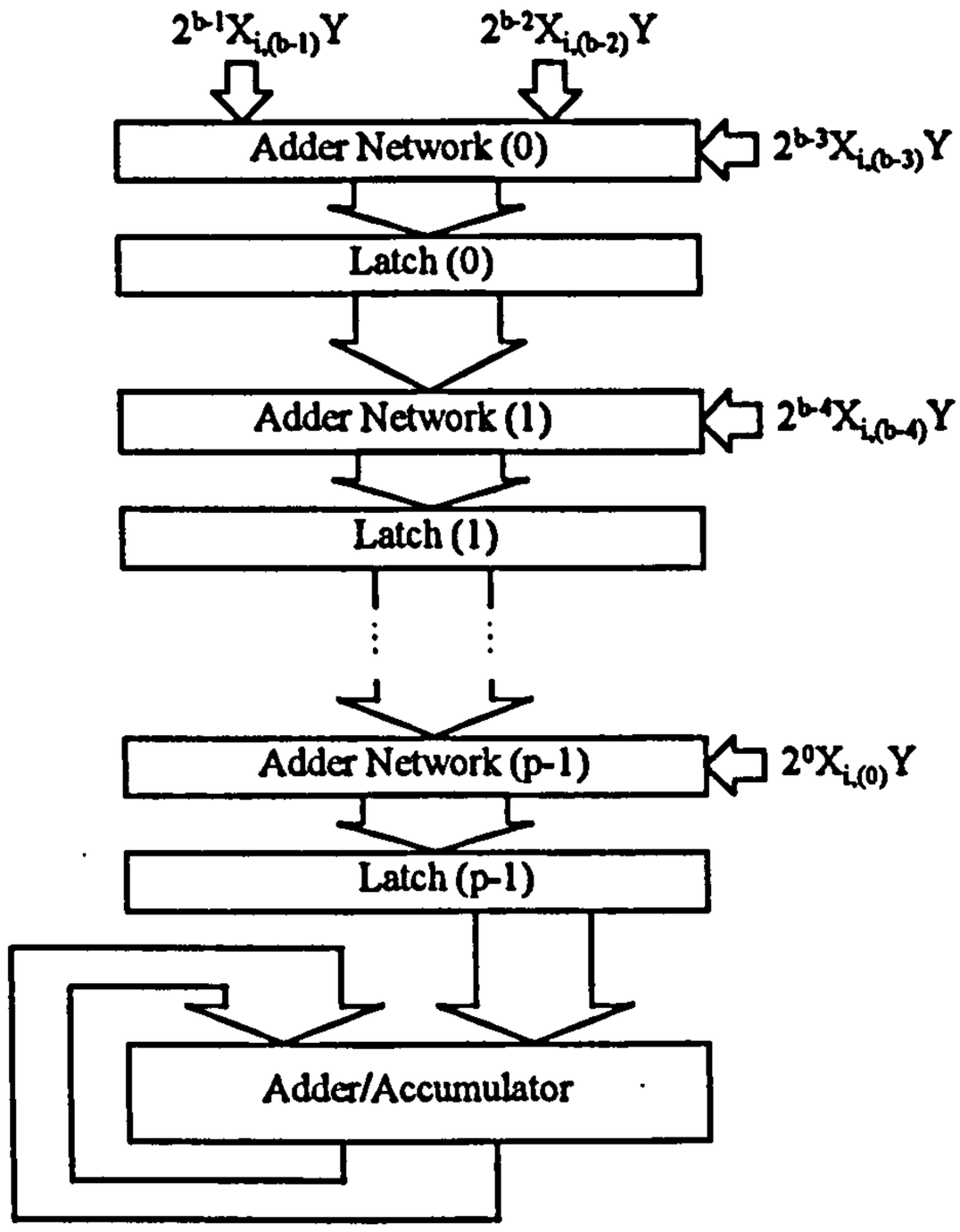


Figure 4.10: A pipelined iterative multiplier.

this diagram the adder networks, as implied by the labelling, consist of simple 1-level 3:2 adders but this need not necessarily be the case. Each network could consist of 2-level

3:2 adders or 1-level 5:3 adders etc. The p pipeline stages are used to accumulate partial sums of partial products as they proceed from the top to the bottom of the diagram. At the bottom is an adder/accumulator that is used to sum the partial sums as they arrive. The adder/accumulator has a built-in hardwired shift as it feeds back its outputs to its inputs as was the case in the simple CSA multiplier.

The time required for each iteration of a multiplication is the sum of the times required for a single adder network and the latch setup/hold times. The total number of iterations required is increased from the simple multi-level adder CSA design because of the need to fill/empty the pipeline at the beginning/end of a multiplication.

For example, consider the case with radix-16 ($b = 4$) multiplication using a 2-stage pipeline plus adder/accumulator. The Right-to-Left algorithm will be used, thus the adder/accumulator has a built-in right-shift of 4 bits and the shifted out bits are assumed to be saved in a 4-bit wide shift register attached to the accumulator. If we limit the size of the operands to just $l = 5$ digits then we can follow the process of multiplication as follows.

$$
\begin{aligned}
\text{Reset:} \quad & \text{Latch}(0) = 0 \\
& \text{Latch}(1) = 0 \\
& \text{Accum/SR} = 0 \\
\text{Cycle 0:} \quad & \text{Latch}(0) = 2^3 \cdot X_{0,(3)} \cdot Y + 2^2 \cdot X_{0,(2)} \cdot Y + 2^1 \cdot X_{0,(1)} \cdot Y \\
& \text{Latch}(1) = 0 \\
& \text{Accum/SR} = 0 \\
\text{Cycle 1:} \quad & \text{Latch}(0) = 2^3 \cdot X_{1,(3)} \cdot Y + 2^2 \cdot X_{1,(2)} \cdot Y + 2^1 \cdot X_{1,(1)} \cdot Y \\
& \text{Latch}(1) = 2^3 \cdot X_{0,(3)} \cdot Y + 2^2 \cdot X_{0,(2)} \cdot Y + 2^1 \cdot X_{0,(1)} \cdot Y + 2^0 \cdot X_{0,(0)} \cdot Y = X_0 \cdot Y \\
& \text{Accum/SR} = 0
\end{aligned}
$$

$$\begin{aligned}
\text{Cycle 2:}\quad \text{Latch}(0) &= 2^3 \cdot X_{2,(3)} \cdot Y + 2^2 \cdot X_{2,(2)} \cdot Y + 2^1 \cdot X_{2,(1)} \cdot Y \\
\text{Latch}(1) &= 2^3 \cdot X_{1,(3)} \cdot Y + 2^2 \cdot X_{1,(2)} \cdot Y + 2^1 \cdot X_{1,(1)} \cdot Y + 2^0 \cdot X_{1,(0)} \cdot Y = X_1 \cdot Y \\
\text{Accum/SR} &= 2^{16} \cdot X_0 \cdot Y \\
\\
\text{Cycle 3:}\quad \text{Latch}(0) &= 2^3 \cdot X_{3,(3)} \cdot Y + 2^2 \cdot X_{3,(2)} \cdot Y + 2^1 \cdot X_{3,(1)} \cdot Y \\
\text{Latch}(1) &= 2^3 \cdot X_{2,(3)} \cdot Y + 2^2 \cdot X_{2,(2)} \cdot Y + 2^1 \cdot X_{2,(1)} \cdot Y + 2^0 \cdot X_{2,(0)} \cdot Y = X_2 \cdot Y \\
\text{Accum/SR} &= 2^{16} \cdot X_1 \cdot Y + 2^{12} \cdot X_0 \cdot Y \\
\\
\text{Cycle 4:}\quad \text{Latch}(0) &= 2^3 \cdot X_{4,(3)} \cdot Y + 2^2 \cdot X_{4,(2)} \cdot Y + 2^1 \cdot X_{4,(1)} \cdot Y \\
\text{Latch}(1) &= 2^3 \cdot X_{3,(3)} \cdot Y + 2^2 \cdot X_{3,(2)} \cdot Y + 2^1 \cdot X_{3,(1)} \cdot Y + 2^0 \cdot X_{3,(0)} \cdot Y = X_3 \cdot Y \\
\text{Accum/SR} &= 2^{16} \cdot X_2 \cdot Y + 2^{12} \cdot X_1 \cdot Y + 2^8 \cdot X_0 \cdot Y \\
\\
\text{Cycle 5:}\quad \text{Latch}(0) &= 0 \\
\text{Latch}(1) &= 2^3 \cdot X_{4,(3)} \cdot Y + 2^2 \cdot X_{4,(2)} \cdot Y + 2^1 \cdot X_{4,(1)} \cdot Y + 2^0 \cdot X_{4,(0)} \cdot Y = X_4 \cdot Y \\
\text{Accum/SR} &= 2^{16} \cdot X_3 \cdot Y + 2^{12} \cdot X_2 \cdot Y + 2^8 \cdot X_1 \cdot Y + 2^4 \cdot X_0 \cdot Y \\
\\
\text{Cycle 6:}\quad \text{Latch}(0) &= 0 \\
\text{Latch}(1) &= 0 \\
\text{Accum/SR} &= 2^{16} \cdot X_4 \cdot Y + 2^{12} \cdot X_3 \cdot Y + 2^8 \cdot X_2 \cdot Y + 2^4 \cdot X_1 \cdot Y + 2^0 \cdot X_0 \cdot Y
\end{aligned}$$

and so at the end of processing the accumulator/shift-register contains

$$\sum_{i=0}^{l-1} 2^{ib} \cdot X_i \cdot Y = X \cdot Y$$

In this example with each adder network being 1-level 3:2 adders, then

$$\begin{aligned}
\text{Number of iterations} &= \lceil k/b \rceil + b - 3 \\
\text{Iteration time} &= \Delta_{AND} + \Delta_{FA} + \Delta_{FF}
\end{aligned}$$

4.3 Multiplier Recoding

In the previous section multipliers have been implemented by adding partial products generated by terms of the form $X_{i,(j)} \cdot Y$ where $X_{i,(j)} \in \{0, 1\}$ as was shown in Figure 4.6.

But in digital hardware it is relatively easy to generate numbers of the form $2^\beta \cdot Y$ for $\beta \geq 0$ by a simple left-shift of Y by β bits. This fact can be taken advantage of in *string recoding* theory [30].

First-order string recoding is the transformation of the bit-vector

$$X = [x_{k-1}, x_{k-2}, \ldots, x_0]$$

where $x_i \in \{0,1\}$, to the representation

$$X' = [x'_k, x'_{k-1}, \ldots, x'_0]$$

where $x'_i \in \{-1,0,1\}$, such that the values of X and X' are the same, i.e.

$$\sum_{i=0}^{k-1} 2^i \cdot x_i = \sum_{i=0}^{k} 2^i \cdot x'_i \tag{4.1}$$

This can be accomplished by a mapping of the bits of X to the digits of X' governed by the following equation,

$$x'_i = -x_i + x_{i-1}$$

for $i = 0 \ldots k$ with $x_{-1} = x_k = 0$. That this produces a vector X' of the same value as X can be seen by substituting the definition of x'_i into the right-hand side of Equation 4.1 and simplifying.

Since each bit of X is in $\{0,1\}$, the mapping is equivalent to

x'_i	$\Leftarrow$	x_i	x_{i-1}	Reason
0		0	0	No string
1		0	1	End of string
−1		1	0	Beginning of string
0		1	1	Center of string

with the assumption that $x_{-1} = x_k = 0$.

The comments on the right-hand side refer to 'strings of 1's appearing in the bit-vector representation of X. If the above mapping is applied while scanning X from right-to-left,

then the first 1-bit in a sequence of 1's will reveal itself with $x_i = 1$ and $x_{i-1} = 0$, i.e. the beginning of a string of 1's. (Note that scanning starts with $i = 0$ and $x_{-1} = 0$).

For example, with $k = 8$ and an arbitrary bit-vector X, then

	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{-1}
X:	0	1	0	1	1	1	0	0	1	0
X':	1	-1	1	0	0	-1	0	1	-1	

and to check we have

$$X = 2^7 + 2^5 + 2^4 + 2^3 + 2^0 = 185$$

$$X' = 2^8 - 2^7 + 2^6 - 2^3 + 2^1 - 2^0 = 185$$

Note that the mapping does not have to be performed serially from right-to-left. Since each digit x'_i depends only on bits x_i and x_{i-1}, the mapping can equally well be applied while scanning from left-to-right or even applied to all digits in parallel.

So, we now have a $(k+1)$-digit vector with each digit $\in \{-1, 0, 1\}$. At first sight this does not seem too promising, it looks as though we have just over-complicated things, but the vector does have one interesting point, and this is that no two adjacent digits have the same sign. This can be seen to be true directly from the mapping table. This property is used to advantage when second-order string recoding is performed.

A second-order recoding can be performed by examining the first-order recoded vector X' to generate the second-order recoded vector X''. It should be noted that X'' is a radix-4 vector of approximately half the length of the radix-2 vector X'.

Explicitly, each digit x''_i of X'' is generated by the equation

$$x''_i = 2 \cdot x'_{2i+1} + x'_{2i}$$

for $i = 0 \ldots \lceil (k+1)/2 \rceil - 1$. Since no two adjacent digits of X' are of the same sign, then

the possible combinations of x'_{2i+1} and x'_{2i} that generate x''_i are as follows.

x''_i	$\Leftarrow$	x'_{2i+1}	x'_{2i}
0		0	0
1		0	1
2		1	0
1		1	−1
−2		−1	0
−1		−1	1

and so we see that $x''_i \in \{-2,-1,0,1,2\}$.

That X'' has equivalent value to X' can be seen directly from the equation governing its construction. Thus we have created a second-order recoded vector X'' of the same value as X but of approximately half the length of X with digits in the extended range of $\{-2,-1,0,1,2\}$. It is also possible to construct X'' from X in a parallel fashion since

$$\begin{aligned} x''_i &= 2 \cdot x'_{2i+1} + x'_{2i} \\ &= 2(-x_{2i+1} + x_{2i}) + (-x_{2i} + x_{2i-1}) \\ &= -2 \cdot x_{2i+1} + x_{2i} + x_{2i-1} \end{aligned}$$

and so we see that each digit of X'' depends only on 3 adjacent digits of X.

To multiply two positive numbers X and Y using the second-order recoded version of X, we can write

$$X \cdot Y = \sum_{i=0}^{l-1} 2^{ib} \cdot x(i) \cdot Y$$

where $b = 2$ and $l = \lceil (k+1)/b \rceil$ and $x(i)$ is the i-th element of the recoded X vector in the range $\{-2,-1,0,1,2\}$.

On comparison with the un-recoded ($b = 2$) multiplication method where we had

$$X \cdot Y = \sum_{i=0}^{l-1} 2^{ib} \cdot X_i \cdot Y$$

the main difference is that $X_i \in \{0,1,2,3\}$, and we can see that it is the generation of $3 \cdot Y$ that necessitates the use of a 2-level adder network because this multiple must be generated by the separate sub-multiples of $1 \cdot Y$ plus $2 \cdot Y$. But if we use the recoded X vector, then since we know that multiples of 2 can be generated by a simple left-shift, it should be possible to implement a $b = 2$ multiplier with only a single level adder network. All that is needed to do this is some means of implementing signed numbers in hardware. This is the subject of the next section.

For the moment, assuming such a signed number system is available, we can see how high-radix multipliers using recoding techniques can be constructed using only half the number of adder levels that were required in the previous section.

A radix-2^b (b a multiple of 2) multiplier can be constructed by again writing

$$X \cdot Y = \sum_{i=0}^{l-1} 2^{ib} \cdot x(i) \cdot Y$$

but this time with

$$x(i) = \sum_{j=0}^{b/2-1} 2^{2j} \cdot x_j(i)$$

where $x_j(i)$ denotes the $(i\frac{b}{2} + j)$-th digit of the second-order recoded X vector with each digit in the range $\{-2,-1,0,1,2\}$. Then there are $b/2$ adder levels, each level adding the partial product $x_j(i) \cdot Y$ to the accumulated result.

For example, with $b = 4$,

$$X \cdot Y = \sum_{i=0}^{l-1} 2^{4i}(4 \cdot x_1(i) \cdot Y + x_0(i) \cdot Y)$$

is constructed with a 2-level adder. The first level adds multiples 0, $\pm Y$, $\pm 2Y$, and the second level adds multiples 0, $\pm 4Y$, $\pm 8Y$.

4.4 Signed Number Representations

As was stated in the previous section, to enable multiplier recoding techniques to be used for the construction of efficient iterative multipliers requires a means of representing both

positive and negative numbers in hardware. In this section we will examine various signed number representations.

The three most common ways of representing signed numbers in digital hardware are the *2's complement*, *1's complement* and *sign-magnitude* methods. In the following subsections X is assumed to be a k-bit bit-vector, $X = [x_{k-1}, x_{k-2}, \ldots, x_0]$ where $x_i \in \{0, 1\}$. The procedures for calculating the value of X based on this bit-vector representation are given for each of the above signed number encoding schemes.

4.4.1 2's Complement Representation

The value of X is given by

$$X = -2^{k-1} \cdot x_{k-1} + \sum_{i=0}^{k-2} 2^i \cdot x_i$$

and thus the range of values that X may take on is

$$-2^{k-1} \leq X \leq 2^{k-1} - 1$$

To negate a 2's complement bit-vector (form the negative of the number), we use the well-known 'invert and add one' technique - ignoring any carry from the $(k-1)$-th bit. A proof of this technique is as follows.

Using the bit-inversion relation $\overline{x_i} = 1 - x_i$ and if X' is the result of inverting X and adding one, i.e. $X' = [\overline{x_{k-1}}, \overline{x_{k-2}}, \ldots, \overline{x_0}] + 1$, then we have

$$\begin{aligned}
X' &= -2^{k-1} \cdot (1 - x_{k-1}) + \sum_{i=0}^{k-2} 2^i \cdot (1 - x_i) + 1 \\
&= (-2^{k-1} + \sum_{i=0}^{k-2} 2^i + 1) + (2^{k-1} \cdot x_{k-1} - \sum_{i=0}^{k-2} 2^i \cdot x_i) \\
&= (-2^{k-1} + (2^{k-1} - 1) + 1) + -1 \cdot (-2^{k-1} \cdot x_{k-1} + \sum_{i=0}^{k-2} 2^i \cdot x_i) \\
&= (0) + -1 \cdot (X) \\
&= -X
\end{aligned}$$

and thus $[\overline{x_{k-1}}, \overline{x_{k-2}}, \ldots, \overline{x_0}] + 1 = -X$.

To sign-extend a 2's complement bit-vector (convert from a k-bit bit-vector to an n-bit bit-vector where $n > k$), we can use the following mapping. Given $X = [x_{k-1}, x_{k-2}, \ldots, x_0]$ and $X' = [x'_{n-1}, x'_{n-2}, \ldots, x'_0]$, $n > k$ then

$$x'_i = \begin{cases} x_i & \text{for } i = 0 \ldots k-2 \\ x_{k-1} & \text{for } i = k-1 \ldots n-1 \end{cases}$$

This obviously works when $x_{k-1} = 0$. To see how it works when $x_{k-1} = 1$, we note that what we are trying to do is to make a contribution of -2^{k-1} to the X' vector. To see how this happens we simply write

$$\begin{aligned} -2^{k-1} &= -2^k + 2^{k-1} \\ &= -2^{k+1} + 2^k + 2^{k-1} \\ &\vdots \\ &= -2^{n-1} + 2^{n-2} + \ldots + 2^{k-1} \end{aligned}$$

and note that this is exactly what happens with the upper $n-k$ bits in the above mapping.

4.4.2 1's Complement Representation

The value of a k-bit 1's complement vector is given by

$$X = (-2^{k-1} + 1) \cdot x_{k-1} + \sum_{i=0}^{k-2} 2^i \cdot x_i$$

and so the range of values that X may take on is

$$-(2^{k-1} - 1) \leq X \leq 2^{k-1} - 1$$

Negation of a 1's complement bit-vector is performed by simply inverting every bit. That is

$$-X = [\overline{x_{k-1}}, \overline{x_{k-2}}, \ldots, \overline{x_0}]$$

Sign-extension is the same as for 2's complement.

Both of these assertions can be proved in a similar manner to the 2's complement proofs and are omitted here.

4.4.3 Sign-Magnitude Representation

The value of a k-bit sign-magnitude bit-vector is given by

$$X = (-1)^{x_{k-1}} \cdot \sum_{i=0}^{k-2} 2^i \cdot x_i$$

and so its range is

$$-(2^{k-1} - 1) \leq X \leq 2^{k-1} - 1$$

Negation of a sign-magnitude number is obviously performed by inverting the sign bit. That is

$$-X = [\overline{x_{k-1}}, x_{k-2}, \ldots, x_0]$$

Sign-extension from $X = [x_{k-1}, x_{k-2}, \ldots, x_0]$ to $X' = [x'_{n-1}, x'_{n-2}, \ldots, x'_0]$, $n > k$ is given by

$$x'_i = \begin{cases} x_i & \text{for } i = 0 \ldots k-2 \\ 0 & \text{for } i = k-1 \ldots n-2 \\ x_{k-1} & \text{for } i = n-1 \end{cases}$$

and is obvious.

Upon examination of the above three methods of signed number representation we can see that they all share a common feature; they all use a special bit, x_{k-1}, to determine the sign of the number. When trying to implement an iterative multiplier using these signed number systems in a CSA-type architecture, this special bit causes major problems. This is because, as we have seen, iterative multipliers require a shift operation to be built into the adder/accumulator circuit. When dealing with signed number systems this shift operation is basically a sign-extension operation and, as we have seen above, to be able to sign-extend a number using any of these representations requires knowledge of the special sign-bit. The crux of the matter is that this special sign bit is the top bit of each bit-vector and can only be accurately determined by the fully carry-propagated additions/subtractions. Since the CSA architecture prohibits these propagations, there is no way to determine the sign of any

partial results held in the accumulator during processing. Thus the above representations cannot be used within a CSA architecture.

What is needed is a method of signed number representation that does not rely on any special sign bit. Such a method is the *signed-digit* representation.

4.4.4 Signed-Digit Representation

A signed-digit number is one in which each digit of the number can take on positive and negative values. In general

$$A = [\alpha_{k-1}, \alpha_{k-2}, \ldots, \alpha_0]$$

is a k-digit vector whose value is given by

$$A = \sum_{i=0}^{k-1} r^i \cdot \alpha_i$$

where each α_i can take on values in the range

$$\alpha_i \in [-\gamma, \gamma]$$

for some fixed γ in the range

$$\lceil (r-1)/2 \rceil \leq \gamma \leq r-1$$

For example, with $r = 4$ and $\gamma = 3$, we have

$$A = \sum_{i=0}^{k-1} 2^{2i} \cdot \alpha_i$$

where $\alpha_i \in \{-3, -2, -1, 0, 1, 2, 3\}$ and so A has range

$$-(2^{2k} - 1) \leq A \leq 2^{2k} - 1$$

The negation of a signed-digit vector is simply the negation of each digit, i.e.

$$-A = [-\alpha_{k-1}, -\alpha_{k-2}, \ldots, -\alpha_0]$$

and 'sign-extension' from $A = [\alpha_{k-1}, \alpha_{k-2}, \ldots, \alpha_0]$ to $A' = [\alpha_{n-1}, \alpha_{n-2}, \ldots, \alpha_0]$ for $n > k$ simply involves filling the $n - k$ upper digits of A' with zeros. That is

$$\alpha'_i = \begin{cases} \alpha_i & \text{for } i = 0 \ldots k-1 \\ 0 & \text{for } i = k \ldots n-1 \end{cases}$$

The addition of two signed-digit vectors, A and B, can be performed with carry-propagation limited to one position to the left as follows. With $A = [\alpha_{k-1}, \alpha_{k-2}, \ldots, \alpha_0]$ and $B = [\beta_{k-1}, \beta_{k-2}, \ldots, \beta_0]$ and their sum $S = A + B = [s_k, s_{k-1}, \ldots, s_0]$ then the addition proceeds with the generation of intermediate sum digits w_i and transfer digits t_{i+1} obeying the relation

$$\alpha_i + \beta_i = r \cdot t_{i+1} + w_i$$

and is completed by the addition of appropriate intermediate sum and transfer digits

$$s_i = w_i + t_i$$

Note that the transfer digit generated in the first equation is in effect a limited carry from the i-th adder digit to the $(i+1)$-th digit. The transfer digit in the second equation is the limited carry that was generated in the $(i-1)$-th digit position. Thus the transfer digits are actually carries whose propagation is limited to one place to the left.

In order to ensure that this limited carry-propagation/transfer-digit scheme works, the possible values that w_i and t_i can assume must be limited so that

$$-\gamma \leq s_i = w_i + t_i \leq \gamma$$

For example, with $r = 4$ and $\gamma = 3$, then restricting

$$w_i \in \{-2, -1, 0, 1, 2\}, \quad t_i \in \{-1, 0, 1\}$$

ensures that $-3 \leq w_i + t_i \leq 3$. In particular, Table 4.1 shows just one method of generating the transfer digit and partial sum based on the sum from the input digits $\alpha_i + \beta_i$.

Although this general signed-digit approach does have simple number manipulation properties and a fast addition scheme, it suffers from a slight over-complexity [26] when

$\alpha_i + \beta_i$	t_{i+1}	w_i
-6	-1	-2
-5	-1	-1
-4	-1	0
-3	-1	1
-2	0	-2
-1	0	-1
0	0	0
1	0	1
2	0	2
3	1	-1
4	1	0
5	1	1
6	1	2

Table 4.1: Signed-digit addition; transfer and intermediate sum digits.

it comes to implementing the addition scheme in hardware. A better method for using signed-digit numbers in high-performance hardware is shown in the next section.

4.4.5 Redundant Signed-Digit (RSD) Representation

The redundant signed-digit representation of signed numbers combines the flexibility of using a signed-digit approach with the simple hardware of CSA adders.

A k-digit RSD vector X, is written as

$$X = [x_{k-1}, x_{k-2}, \ldots, x_0]$$

has digits, x_i, in the range

$$x_i \in \{-1, 0, 1\}$$

and whose value is given by

$$X = \sum_{i=0}^{k-1} 2^i \cdot x_i$$

In a manner similar to the CSA approach, RSD vectors can be viewed as the *difference* (not sum) of two distinct vectors, such that

$$x_i = x_i^+ - x_i^-$$

where

$$x_i^+, x_i^- \in \{0, 1\}$$

and so the value of X is given by

$$X = \sum_{i=0}^{k-1} 2^i \cdot (x_i^+ - x_i^-)$$

or alternatively

$$X = X^+ - X^-$$

where

$$X^+ = \sum_{i=0}^{k-1} 2^i \cdot x_i^+, \quad X^- = \sum_{i=0}^{k-1} 2^i \cdot x_i^-$$

The heart of the RSD approach is in the generalisation of the 3:2 adder to a class of General Full Adders (GFA). Each member of the class can add a particular combination of positively and negatively weighted bits. Figure 4.11 shows the circuit symbols of these GFAs and their functions are summarized as follows.

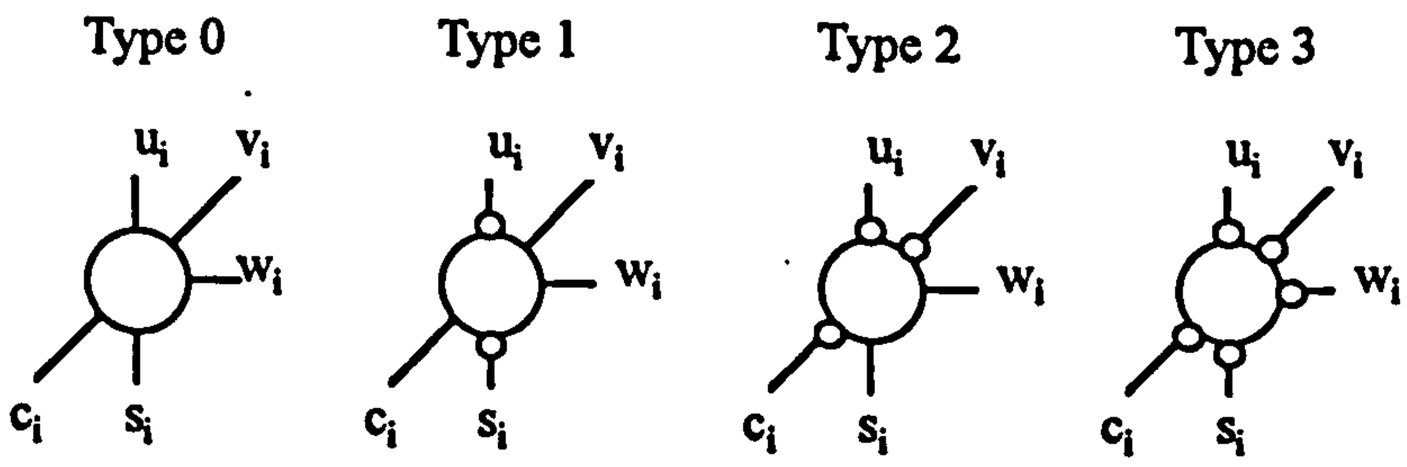


Figure 4.11: General full adders.

- Type 0: $u_i + v_i + w_i = 2c_{i+1} + s_i$
- Type 1: $-u_i + v_i + w_i = 2c_{i+1} - s_i$
- Type 2: $-u_i - v_i + w_i = -2c_{i+1} + s_i$
- Type 3: $-u_i - v_i - w_i = -2c_{i+1} - s_i$

Note that the standard one-bit full adder is classified as type 0, and that the 'bubbles' on certain GFA inputs and outputs are actually implemented in hardware as inverters and they signify which of the inputs/outputs are negatively weighted. Also note that, in CMOS VLSI technology, inverters usually 'come-for-free', meaning that taking the true or inverted value of a signal in a CMOS circuit normally just means connecting to a different

point within the circuit. This is simply a consequence of the CMOS transistor-unit being a pair of complementary p-type and n-type transistors.

The addition of two k-digit RSD vectors X and Y to yield a $(k+1)$-digit sum S, can be performed either by summing their respective positive and negative components together with fully carry-propagated additions, i.e. $S^+ = X^+ + Y^+$ and $S^- = X^- + Y^-$, or, because the vectors have a redundant representation, by using a 2-level GFA adder network as shown in Figure 4.12. Note that this method is much preferable since the

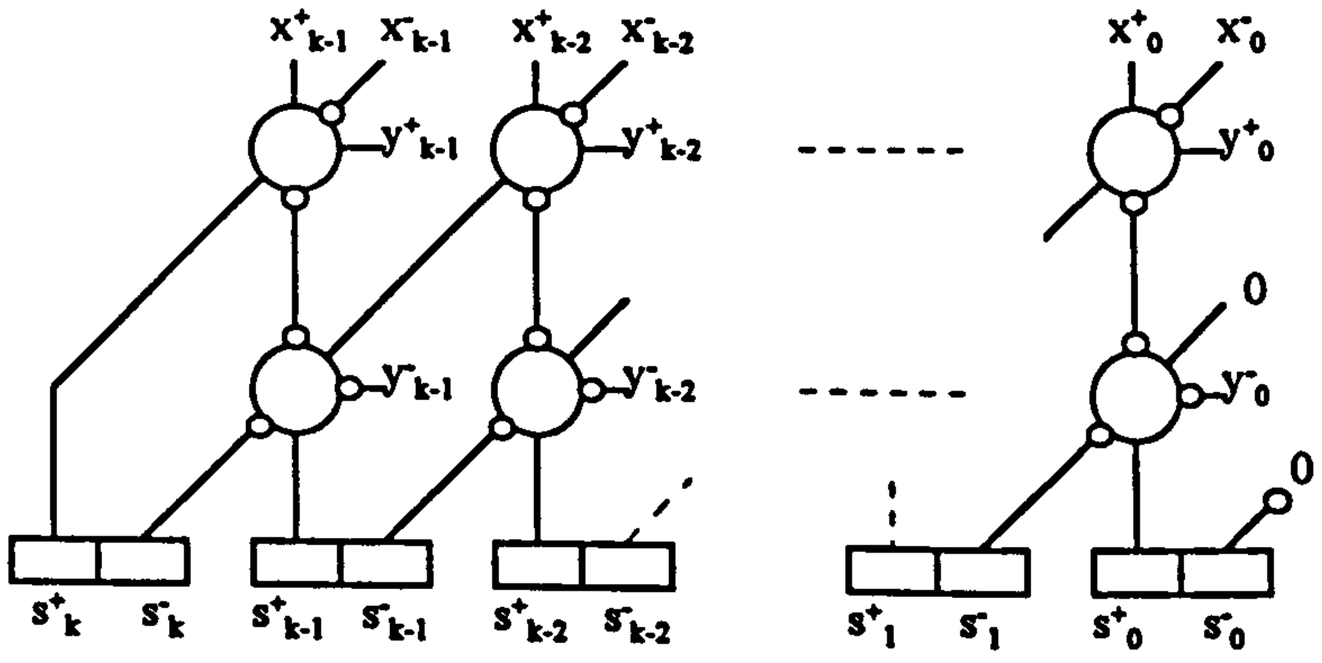


Figure 4.12: RSD addition.

addition time is constant (2-GFA delay) because no carry propagation is necessary.

Referring back to the section on 2's complement representation we can see that a 2's complement vector can be viewed as a special case of an RSD vector. That is, for $Y = [y_{k-1}, y_{k-2}, \ldots, y_0]$ with $y_i \in \{0,1\}$ a 2's complement vector, and $X = [x_{k-1}, x_{k-2}, \ldots, x_0]$ with $x_i = x_i^+ - x_i^- \in \{-1,0,1\}$ an RSD vector, the mapping

$$x_i^+ = \begin{cases} y_i & \text{for } i = 0 \ldots k-2 \\ 0 & \text{for } i = k-1 \end{cases}$$

$$x_i^- = \begin{cases} 0 & \text{for } i = 0 \ldots k-2 \\ y_i & \text{for } i = k-1 \end{cases}$$

converts from 2's complement to RSD. This can be used to advantage when adding a 2's complement vector to an RSD vector as shown in Figure 4.13. This circuit can be further simplified by noting that the lower $(k-1)$-th adder in the diagram simplifies down to a

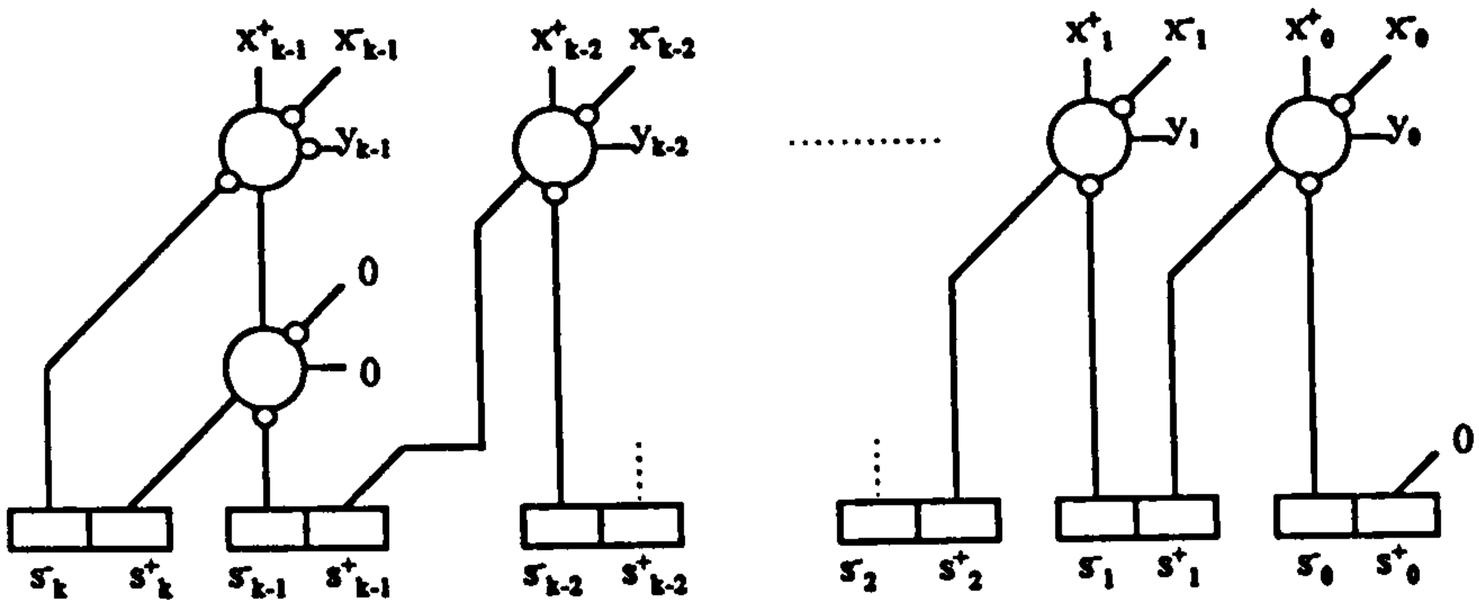


Figure 4.13: Addition of 2's complement vector to RSD vector.

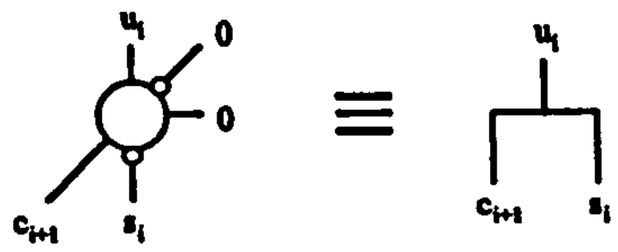


Figure 4.14: Simplification of lower $(k-1)$-th adder of Figure 4.13.

direct connection as shown in Figure 4.14. Thus a 2's complement vector can be added to an RSD vector by a single layer of GFAs. Since both of these representations allow for signed numbers, this mechanism is ideal for implementing recoded multipliers as we shall see in the next section.

Finally, a k-digit RSD vector, X, can be converted to a $(k+1)$-bit 2's complement vector Y, with the aid of a fully carry-propagated adder as shown in Figure 4.15. The addition

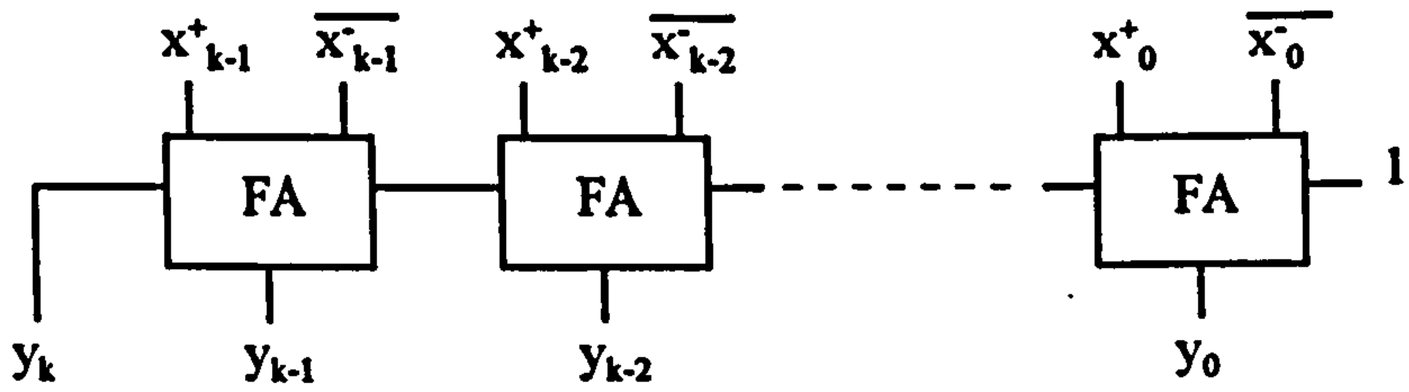


Figure 4.15: Conversion of RSD vector to 2's complement.

works by converting the negative part of the RSD vector, X^- to its 2's complement representation on the inputs to the adder. Thus

$$Y = -2^k \cdot y_k + \sum_{i=0}^{k-1} 2^i \cdot y_i$$

which is identical to the value of X.

4.5 A Recoded Multiplier

As an example of how to implement a recoded multiplier using a mixture of 2's complement and RSD signed number representations, a simple 1-level iterative multiplier will be presented. Because of the use of recoding, 2 bits of the multiplier operand, X, are examined in each iteration and multiples 0, $\pm Y$ and $\pm 2Y$ of the multiplicand, Y, are added to the accumulated result. Also, the operands X and Y are assumed to be in 2's complement form. This means that we can directly multiply positive as well as negative numbers.

Firstly, since Y is a k-bit 2's complement vector and we wish to generate multiples of $\pm 2Y$, also in 2's complement form, for addition to the adder/accumulator, we shall have to generate the multiples as a $(k+1)$-bit 2's complement vector $Y' = [y'_k, y'_{k-1}, \ldots, y'_0]$. Remembering that negating a 2's complement vector means 'invert and add one', then multiples $\pm Y$ and $\pm 2Y$ can be generated as follows.

- Generate $+Y$: $+Y = Y'$ where

$$y'_i = \begin{cases} y_i & \text{for } i = 0 \ldots k-2 \\ y_{k-1} & \text{for } i = k-1, k \end{cases}$$

- Generate $-Y$: $-Y = Y' + 1$ where

$$y'_i = \begin{cases} \overline{y_i} & \text{for } i = 0 \ldots k-2 \\ \overline{y_{k-1}} & \text{for } i = k-1, k \end{cases}$$

- Generate $+2Y$: $+2Y = Y'$ where

$$y'_i = \begin{cases} y_{i-1} & \text{for } i = 0 \ldots k-1 \text{ with } y_{-1} = 0 \\ y_{k-1} & \text{for } i = k \end{cases}$$

- Generate $-2Y$: $-2Y = Y' + 1$ where

$$y'_i = \begin{cases} \overline{y_{i-1}} & \text{for } i = 0 \ldots k-1 \text{ with } y_{-1} = 0 \\ \overline{y_{k-1}} & \text{for } i = k \end{cases}$$

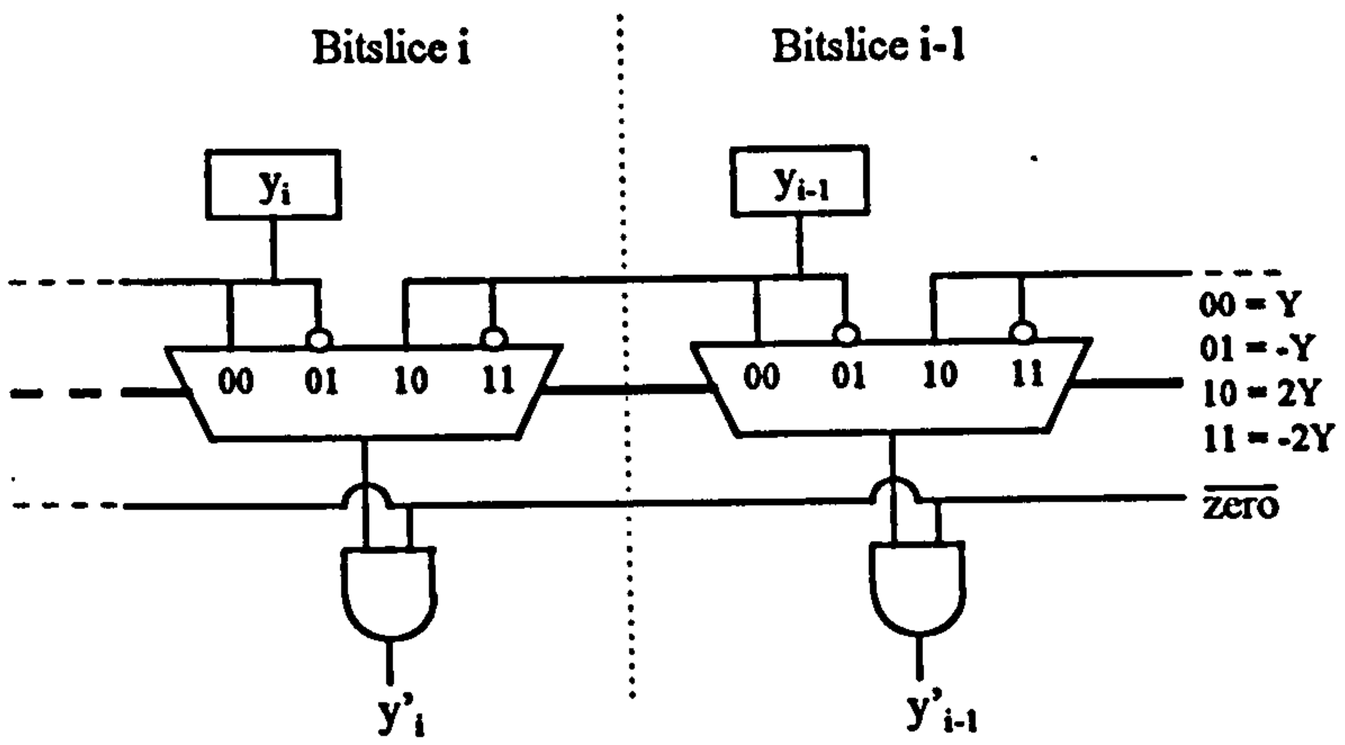


Figure 4.16: Bitslice of 0, $\pm Y$ and $\pm 2Y$ generation.

The hardware required to implement this mapping is shown in Figure 4.16.

The recoding of the multiplier operand X, when X is in 2's complement form, is particularly straightforward. This is because the string recoding techniques of Section 4.3 automatically take care of 2's complement vectors without the need for the dummy $x_k = 0$ bit. Thus a second order recoded vector of X, namely X'' can be generated, if k is a multiple of 2, with $k/2$ digits. This direct conversion of 2's complement vectors to recoded vectors is called Booth encoding (see [26]).

Thus $X \cdot Y$ can be computed with

$$X \cdot Y = \sum_{i=0}^{k/2-1} 2^{2i} \cdot x(i) \cdot Y$$

where $x(i) \in \{-2, -1, 0, 1, 2\}$ is the i-th digit of the recoded X vector.

The architecture of the multiplier is shown in Figure 4.17 where it should be noted

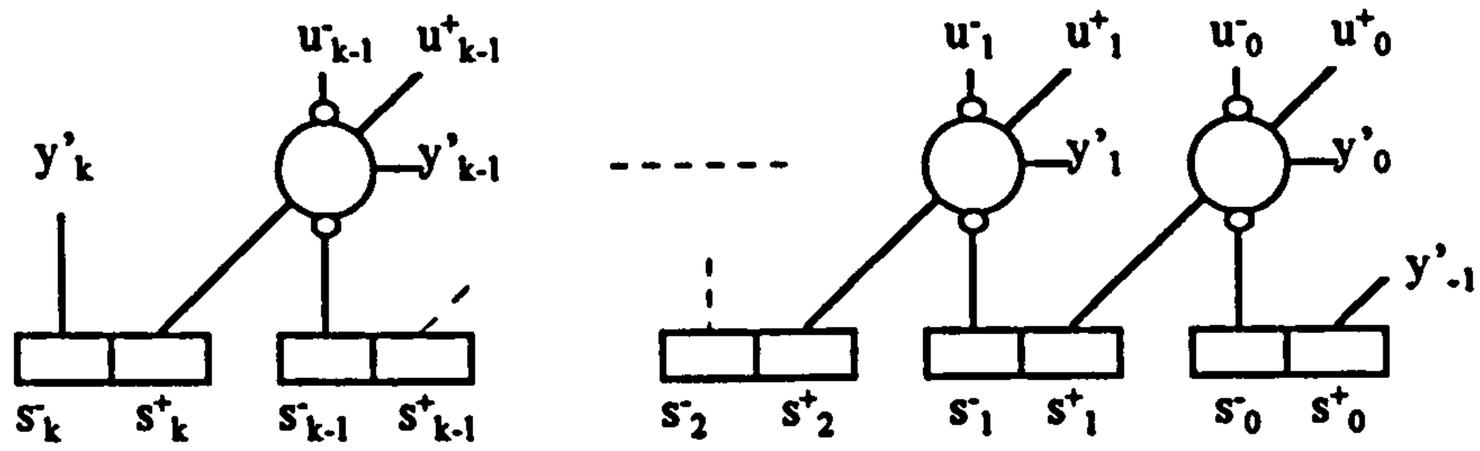


Figure 4.17: Recoded multiplier architecture.

that $y'_{-1} = 1$ if a negative multiple of Y is being added (see Y multiple generation definitions above). This represents the 'add one' instruction from the 'invert and add one' 2's complement negation process.

To implement the Right-to-Left multiplication algorithm a right-shift of 2 bits must be performed by the accumulator every iteration. This is hardwired into the feedback loop as

$$u_i^+ \leftarrow s_{i+2}^+$$

$$u_i^- \leftarrow s_{i+2}^-$$

for $i = 0 \ldots k-2$ with $u_{k-1}^+ = u_{k-1}^- = 0$ constant. The shifted out bits of the accumulator, s_1^+, s_1^-, s_0^+ and s_0^-, are assumed to be collected in a shift-register and a full 2's complement carry-propagated addition can be performed at the end of processing to yield a $2k$-bit result.

Thus a recoded multiplier has been developed with a single level adder network. Performance figures are as follows.

$$\text{Number of iterations} = \left\lceil \frac{k}{2} \right\rceil$$

$$\text{Iteration time} = \Delta_{MUX} + \Delta_{AND} + \Delta_{FA} + \Delta_{FF}$$

$$\text{Number of bitslices} = k + 1$$

$$\text{Bitslice complexity} = \Omega_{MUX} + \Omega_{AND} + \Omega_{FA} + 4 \cdot \Omega_{FF}$$

On comparison with the unrecoded $b = 2$ multiplier we see that the iteration time has been reduced by $\Delta_{FA} - \Delta_{MUX}$ and the complexity reduced by $\Omega_{FA} + \Omega_{AND} - \Omega_{MUX}$ per bitslice.

4.6 Summary

This chapter has studied the basic building blocks of arithmetic hardware. Efficient adder circuits and iterative multipliers were explained followed by a discussion of multiplier recoding techniques and the necessity for an efficient signed-number representations.

This culminated in the design of an efficient iterative multiplier with a recoded RSD architecture that can perform signed multiplications with both operands and the result in

2's complement form.

This basic architecture, modified to perform Montgomery multiplications, will be used to construct the optimised multipliers of Chapter 7.

Chapter 5

Standard RSA Hardware

Over the past 10-15 years there have been various proposals for implementing long-integer modular arithmetic circuits in custom ASIC devices, that would be suitable for use within an RSA cryptosystem. Few of these proposals have been realized in silicon. In this chapter we will review the methods that have been put forward to perform standard modular multiplication in custom hardware. By 'standard' it is meant that the proposals are based on modified versions of the algorithms that appeared in Chapter 3. Where applicable, the successful implementation of a design will be noted.

5.1 Multiple-Precision Arithmetic Hardware

Long-integer arithmetic in software is performed using multi-precision techniques. Simply put, this means constructing long-integer operations, such as addition and multiplication, from smaller sized arithmetic primitives. These primitives are usually based on the natural word-size of the computer being used, e.g. 16-bit additions, multiplications etc. Whilst long-integer arithmetic performed in this way is of course much slower than dedicated hardware, it does have the advantage of ease of implementation on more general arithmetic hardware [31] [32] [33] [34] [35].

This approach has been successfully used to implement RSA cryptography on dedicated

DSP (Digital Signal Processing) chips [36] [37]. The rationale behind using a DSP chip is that such chips usually contain fast arithmetic processors, in particular, very fast 16-32 bit multipliers are usually available.

Multi-precision techniques have also been used in designing custom ASIC devices [38]. The idea here is that these chips will be small and low-powered and so suitable for embedding in SmartCards.

5.2 Multiply-Divide Hardware

As was mentioned in Section 3.3.1, one of the ways of performing modular multiplication is by first multiplying the operands and then dividing by the modulus and keeping the remainder. Whilst this approach is acceptable in multi-precision software, it leads to ineffecient use of circuit area when implemented in hardware. Nevertheless, some proposals use this scheme including [39] and notably [40] where a chip has been fabricated, using full-custom design techniques, that can perform RSA exponentiations at the rate of 8kbps for key lengths of 1024-bit.

Other hardware division algorithms can be found in [41] [42] [43] [44] [45] [46].

5.3 Radix-2 Concurrent Multiply/Reduce Hardware

In this section we will look at radix-2 serial multiplier implementations. That is, multipliers that 'consume' the multiplier operand X 1-bit at a time and perform modular reduction concurrently with the multiplication. (i.e. $b = 1$ multipliers from the previous section modified to perform modular reduction).

5.3.1 Simple Modular Reduction

Looking again at the Left-to-Right modular multiplication algorithm (Algorithm 10) we have

$$s(i+1) \;=\; \langle 2 \cdot s(i) + x_{k-i-1} \cdot Y \rangle_N$$

re-writing this as

$$r(i) \;=\; 2 \cdot s(i) + x_{k-i-1} \cdot Y$$

$$s(i+1) \;=\; r(i) - q_i \cdot N$$

where $s(0) = 0$ and $q_i = \lfloor r(i)/N \rfloor \in \{0,1,2\}$ gives $s(k) = \langle X \cdot Y \rangle_N$ and X, Y and N are all k-bit positive integers with $X, Y \in [0, N-1]$. This can be expressed in pseudo-code as shown in Figure 5.1. Upon completion of this code, $S = \langle X \cdot Y \rangle_N$.

```
1.  S := 0
2.  FOR i := 0 TO k-1
3.      S := 2*S
4.      IF S >= N
5.          S := S - N
6.      ENDIF
7.      IF x_{k-i-1} = 1
8.          S := S + Y
9.          IF S >= N
10.             S := S - N
11.         ENDIF
12.     ENDIF
13. ENDFOR
```

Figure 5.1: L-to-R modular multiplication.

The main problem with this algorithm is the need to calculate q_i. From the pseudo-code it can be seen that this involves a comparison of S and N in lines 4 and 9. Since this is a full k-bit comparison of S and N, the time taken to perform this would be the same as that taken for a fully carry-propagated addition of S and N. As we have seen in Section 4.1 this can take a relatively long time. Nevertheless, this algorithm has been

used in an RSA chip [47] using a type of carry-completion adder. The implementation was succesful, if a little slow.

A slight improvement to the above algorithm can be made by, instead of keeping $s(i)$ in the range $[0, N-1]$, allowing it to cover the increased range of $s(i) \in [0, 2^k - 1]$. Thus again

$$\begin{aligned} r(i) &= 2 \cdot s(i) + x_{k-1} \cdot Y \\ s(i+1) &= r(i) - q_i \cdot N \end{aligned}$$

but now $q_i = \lfloor r(i)/2^k \rfloor$. Note that, since the range of $s(i)$ has increased, and it is assumed that N is a 2^k-bit integer

$$2^{k-1} < N < 2^k$$

then q_i has the increased range of $q_i \in \{0, 1, 2, 3\}$. The advantage of this approach is that q_i is determined solely from bits k and $k+1$ of $r(i)$; the 'overflow' bits of $r(i)$. Thus no long-integer comparison is required. A pseudo-code interpretation of this algorithm is shown in Figure 5.2.

```
1.  S := 0
2.  FOR i := 0 TO k-1
3.      S := 2*S
4.      IF S >= 2^k
5.          S := S - N
6.          IF S >= 2^k
7.              S := S - N
8.          ENDIF
9.      ENDIF
10.     IF x_{k-i-1} = 1
11.         S := S + Y
12.         IF S >= 2^k
13.             S := S - N
14.         ENDIF
15.     ENDIF
16. ENDFOR
```

Figure 5.2: L-to-R MM: overflow determination of subtractions of N.

The disadvantage of this approach is that the result of the algorithm $s(k)$, is in an

extended range, and to bring it back to $[0, N-1]$ may require a subtraction of N. Also, all additions and subtractions used here are fully carry-propagated so that the overflow can be accurately determined. When using algorithms such as this one in a modular exponentiator the first problem, that of the extended range of partial results, can be overcome if it can be shown that operands with such an extended range are allowed as inputs to the multiplication routine, i.e. during the repeated multiplications of an exponentiation the range of the multiplication results does not diverge. This usually requires a trade-off between the range of the operands/result and the range of q_i. The second problem, of using carry-propagated additions, is not acceptable in high-performance implementations.

5.3.2 Residue-Table Reduction

Instead of using the overflow bits to determine the multiple of N that should be subtracted, as was done in the previous section, these bits can be used to index a small table of pre-computed residues. If each entry in the table is denoted by $T[j]$ for $j = 0, 1, \ldots$, then the contents of each entry are

$$T[j] = \left\langle j \cdot 2^k \right\rangle_N$$

and they are used as follows

$$\begin{aligned} r(i) &= 2 \cdot s(i) + x_{k-i-1} \cdot Y \\ s(i+1) &= \langle r(i) \rangle_{2^k} + T[q_i] \end{aligned}$$

where again $q_i = \lfloor r(i)/2^k \rfloor$ is the overflow of $r(i)$ beyond the k-th bit. Note that, since the table entries are residues modulo N, they are added (not subtracted) to the accumulated sum. If $q_i \in \{0, 1, 2, 3\}$ then, since $T[0] = 0$, a total of three entries are required in the table.

A pseudo-code version of this algorithm is shown in Figure 5.3. The main advantage of this approach should be clear from the code – residue reduction requires only one addition as opposed to the previous methods where multiple subtractions were necessary.

```
1.  S := 0
2.  FOR i := 0 TO k-1
3.      S := 2*S
4.      IF x_{k-i-1} = 1
5.          S := S + Y
6.      ENDIF
7.      q := S DIV 2^k
8.      S := < S >_{2^k} + T[q]
9.  ENDFOR
```

Figure 5.3: L-to-R MM: residue-table lookup.

In [48] and [49] Tomlinson implements this method using a CSA architecture. The basic idea is shown in Figure 5.4. The inputs to the first and second level of adders are

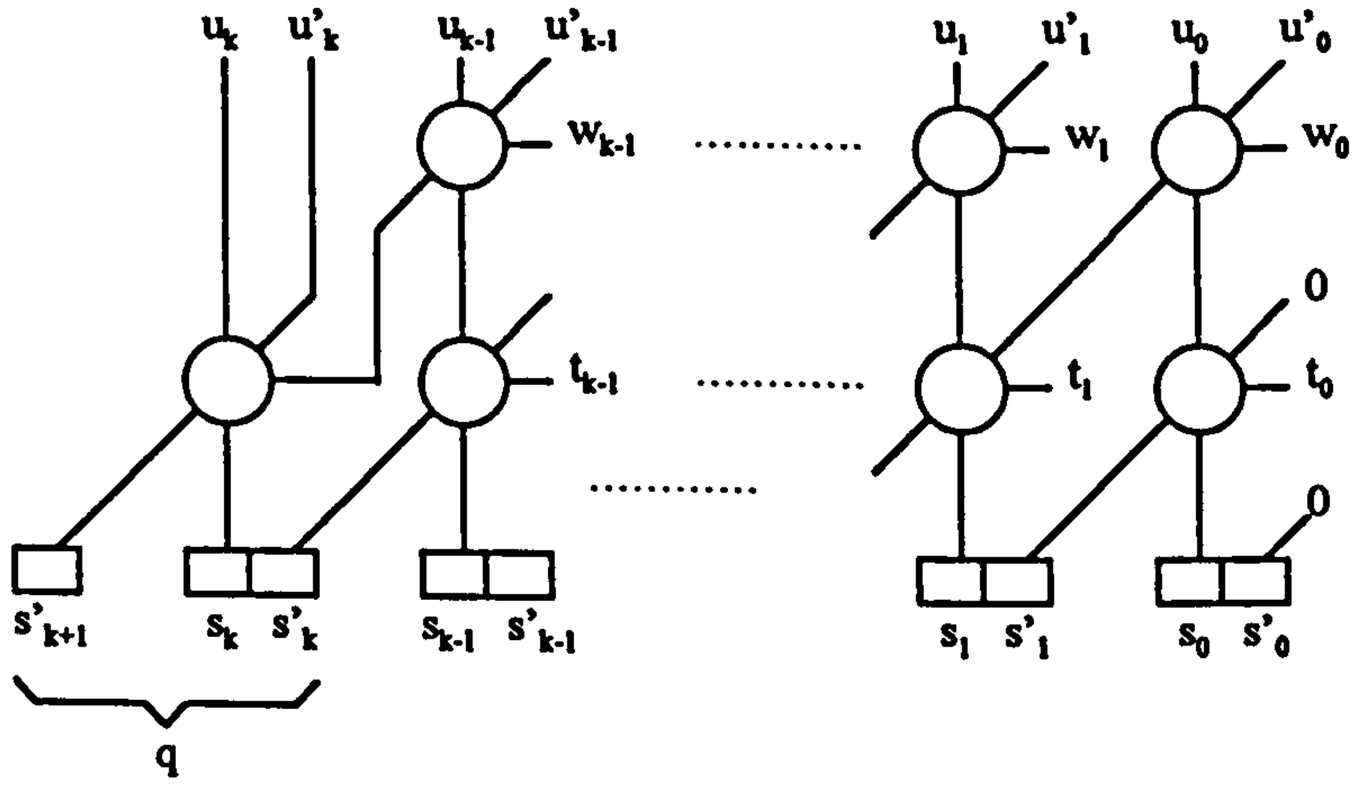


Figure 5.4: Tomlinson modular multiplier.

respectively,

$$w_j = x_{k-i-1} \cdot y_j$$

$$t_j = j\text{-th bit of table residue } T[q_{i-1}]$$

and since, in modular multiplication, the Left-to-Right multiplication algorithm is used, a built-in left shift must be included in the CSA feedback loop. So

$$u_j \leftarrow s_{j-1}$$

$$u'_j \leftarrow s'_{j-1}$$

Thus the algorithm can be expressed as

$$r(i) = 2 \cdot s(i) + x_{k-i-1} \cdot Y$$

$$s(i+1) \;=\; |r(i)|_{2^k} + T[q_{i-1}]$$

where, if

$$s(i) = 2^{k+1} \cdot s'_{k+1} + \sum_{j=0}^{k} 2^j \cdot (s_j + s'_j)$$

then $q_i = 2 \cdot s'_{k+1} + s_k + s'_k$ and Tomlinson claims (but gives no proof) that $q_i \neq 4$ so that $q_i \in \{0,1,2,3\}$. Thus a 3-element residue table is required. The notation $|r(i)|_{2^k}$ is used to imply the removal of the CSA overflow bits of $r(i)$.

Note that, in this algorithm, on each iteration the table-residue corresponding to the previous iteration, $T[q_{i-1}]$ is added to the accumulator. Thus no time is wasted during each iteration on decoding the upper bits of $s(i)$ and selecting the appropriate table entry. This takes place on the following iteration in parallel with the addition of $x_{k-i-1} \cdot Y$ in the first level of adders. Thus the adder array can be run at a rate approaching full speed with very little time wasted in waiting for any intermediate 'residue-determination' calculations. Once the multiplication is complete, an assimilation of the CSA vectors S and S' must be performed (fully carry-propagated addition) along with a subtraction of at most $3N$ to bring the result into the range $[0, N-1]$.

In [50] Iwamura et al. propose a table-lookup scheme based on a modified CSA architecture. In their architecture, the vertical bit-slices of the CSA adder are grouped together horizontally into m-bit sections, $m \geq 3$. This has the effect of reducing the CSA register size (carries are 'saved' after m bit propagations so that the carry vector has fewer elements) but at the expense of complicating the adder array. They show that the overflow after each iteration is limited to 3 bits, and that its value is in the range $[0,6]$. Therefore their design requires a 6-element residue-table.

Chiou, in [51], proposes a similar technique but with a single-level CSA adder array. With this array he alternately adds partial products $x_{k-i-1} \cdot Y$ and table residues. Thus $2k$ iterations are required to complete a modular multiplication (as opposed to k iterations above) but, since only a 1-level adder is required to add either partial products or table

residues, each iteration is almost twice as fast as a standard 2-level CSA adder. He also uses a table of 6 pre-computed residues.

A fully functional RSA chip has been implemented by the Belgian company Cryptech. In [52], Hoornaert et al. describe their system whereby the top few bits of $s(i)$ are compared with the top few bits of the pre-computed binary expansion of $1/N$. It is not stated how many bits are involved in this 'comparison', or how the comparison is performed, but the result is then used to index a 3-element residue-table. The original version of this chip was capable of performing 512-bit RSA exponentiations at the rate of 17kbps. A more recent version using improved VLSI technology (that is assumed to use the same algorithm) can operate at 32kbps.

Recently, in [53], Chiou et al. proposed a system that takes the table-lookup scheme to its extreme. They use a 1-level CSA architecture with just one addition required per cycle and the multiplication completed in k cycles. A 7-element residue table is used with three of the residues of the form $\langle j \cdot 2^k \rangle_N$ and the other four of the form $\langle j \cdot 2^k + Y \rangle_N$ for $j = 0, 1, 2, 3$. Selection of the appropriate residue to add to the accumulator during each iteration of the algorithm is performed by examining the overflow bits of the previous iteration and the current multiplier bit x_{k-i-1}. Initially this scheme looks very attractive but the hidden complication is that four of the table entries involve pre-computed residues of the multiplicand Y. These must be computed before each multiplication and, since they must be in binary, this involves fully carry-propagated additions and subtractions to be performed before the multiplication can take place. Unless these calculations are performed in parallel with fast adders, the time required for the multiplication as a whole is no better than with more conventional approaches. Of course if parallel fast adders are used then circuit complexity becomes an issue.

5.3.3 Quotient Estimation

Instead of using lookup-tables indexed by overflow bits, it is possible to estimate the number of N that should be subtracted from partial results by examining the top few bits of $s(i)$ and N. In discarding the tables we can decrease the hardware requirements of a modular multiplier.

Typically, the more bits of $s(i)$ and N that are examined the smaller is the range to which q_i can be restricted. In general, this approach can be described by

$$r(i) = 2 \cdot s(i) + x_{k-i-1} \cdot Y$$

$$s(i+1) = r(i) - q_i \cdot N$$

where $q_i = f(top(s(i)), top(N))$ is some specific calculation optimised for speed.

A pseudo-language interpretation is shown in Figure 5.5.

```
1.  S := 0
2.  FOR i := 0 TO k-1
3.      S := 2*S
4.      IF x_{k-i-1} = 1
5.          S := S + Y
6.      ENDIF
7.      q := f(top(S),top(N))
8.      S := S - q*N
9.  ENDFOR
```

Figure 5.5: L-to-R MM: quotient estimation.

In 1982 Brickell [54] proposed a design for fast modular multiplication using quotient estimation. His design is based around an adder circuit called a Delayed Carry Adder (DCA). The DCA is similar to the CSA in that carries are not propagated, but it differs in that it is constructed from Half-Adders (HA). Figure 5.6 shows a single bit-slice of a 5-stage DCA. The circuit diagram of Figure 5.7 shows the logic of a half-adder. The DCA has the property that its outputs, s_i and s'_{i+1} in the diagram, do not have a completely redundant representation as in CSA. To be specific

$$s_i \cdot s'_{i+1} \neq 1$$

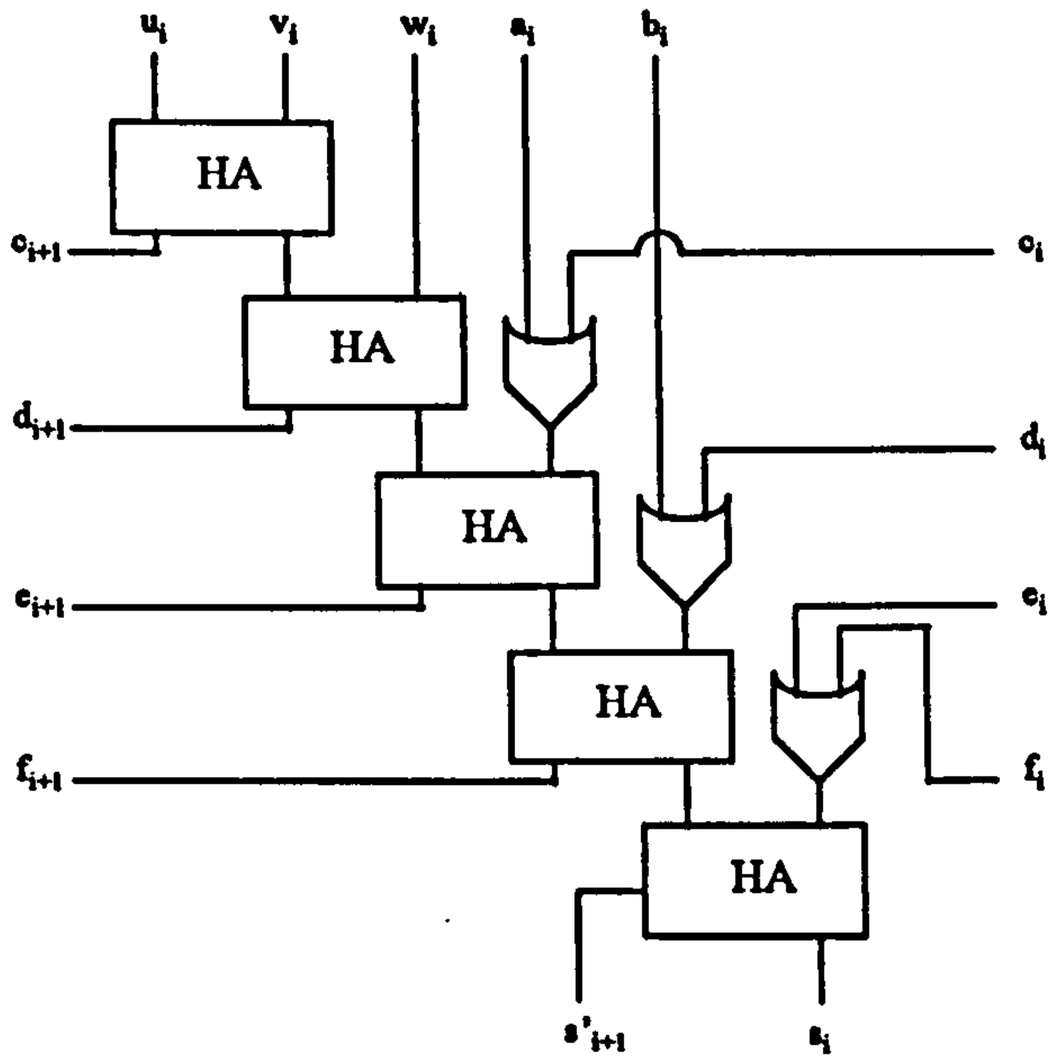


Figure 5.6: Delayed-carry adder (DCA).

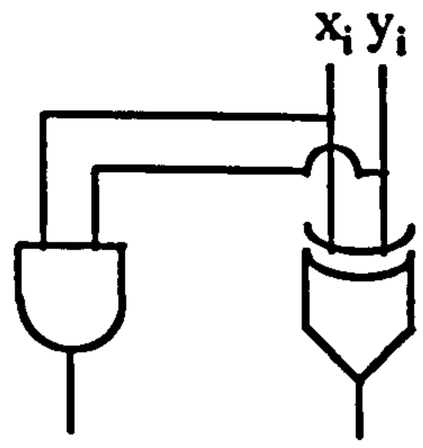


Figure 5.7: Half-adder (HA).

that is s_i and s'_{i+1} cannot both be equal to 1. This can be seen from the circuit diagram of the HA. Brickell was able to use this fact to advantage in the feedback of the DCAs output to its input during the iterations of the multiplication. Note that the speed of the half-adder is about twice that of the full-adder, thus the adder array shown above is only slightly slower than that of a 2-level CSA array.

Brickell's algorithm for quotient estimation allows the partial result $s(i)$ to overflow 2^k by 11 bits. A subtraction of either $2^{10} \cdot N$ or $2^{11} \cdot N$ is performed when appropriate. Thus the range of q is $\{0, 1, 2\}$. The determination of which multiple to subtract is based upon an addition involving the top 4 bits of $s(i)$ and N. Since this is a fully carry-propagated 4-bit addition, it will take time, and it is likely that the DCA will not be able to operate at full speed.

In [55] Baker proposed a quotient estimation technique using a CSA architecture that subtracts multiples $\pm N$ and $\pm 2N$. The selection of the appropriate multiple depends upon a comparison of the fully assimilated top 6 bits of $s(i)$ and N. Since this involves 6-bit carry propagated arithmetic, it too will not allow the CSA to operate at full speed.

Takagi proposed, in [56], to use an RSD architecture with each step of the algorithm reduced separately using multiples $\pm N$. That is

$$\begin{aligned} r(i) &= 2 \cdot s(i) - q_i' \cdot N \\ s(i+1) &= r(i) + x_{k-i-1} \cdot Y - q_i'' \cdot N \end{aligned}$$

where $q_i', q_i'' \in \{-1, 0, 1\}$. Selection of $0, \pm 1$ for q_i' and q_i'' is based on a comparison of the top 3 digits of $r(i)$ and $s(i+1)$ respectively. That is, if

$$\begin{aligned} v &= 4 \cdot r_{k+1} + 2 \cdot r_k + r_{k-1} \\ &= 4(r_{k+1}^+ - r_{k+1}^-) + 2(r_k^+ - r_k^-) + (r_{k-1}^+ - r_{k-1}^-) \end{aligned}$$

then

$$q_i' = \begin{cases} 0 & \text{if } v = 0 \\ 1 & \text{if } v > 0 \\ -1 & \text{if } v < 0 \end{cases}$$

Since an RSD vector has signed digits, the determination of the sign of v depends only on the highest non-zero digit. However, finding the highest non-zero digit still implies a propagation of some sort, albeit limited to 3 digits. Note that splitting the reduction into two parts, that is separate reductions of $r(i)$ and $s(i+1)$, can be advantageous since, for $x_{k-i-1} = 0$, no addition of a partial product and therefore no reduction takes place. However, this only happens approximately 50% of the time and so the multiple selection logic would have to deal with about 4-5 digit propagations per iteration, making the determination of $q_i = q_i' + q_i''$ again slower than the adder array.

5.4 Radix-4 Concurrent Multiply/Reduce Hardware

Implementing radix-4 modular multiplication, we have as a starting point

$$\begin{aligned} r(i) &= 4 \cdot s(i) + X_{k-i-1} \cdot Y \\ s(i+1) &= r(i) - q_i \cdot N \end{aligned}$$

where $X_i \in \{0,1,2,3\}$ and $q_i = \lfloor r(i)/N \rfloor \in \{0,1,2,3,4,5,6\}$. Immediately we see that the range of q_i is much increased from the radix-2 case, and because of this, no proposals have been published in the literature for high-radix residue-table modular multipliers. The tables simply become too large. This leaves quotient estimation as the only viable high-radix modular multiplication method.

Takagi extended his radix-2 multiplication scheme to a radix-4 method in [56]. Each iteration of the algorithm is split into four separate calculations,

$$\begin{aligned} r(i) &= 2 \cdot s(i) - q_i' \cdot N \\ r'(i) &= 2 \cdot r(i) - q_i'' \cdot N \\ t(i) &= x(i) \cdot Y - q_i''' \cdot N \\ s(i+1) &= r'(i) + t(i) - q_i'''' \cdot N \end{aligned}$$

where the vector X is recoded such that $x(i) \in \{-2,-1,0,1,2\}$ and the multiples of N are determined as above in the radix-2 case for $q_i', q_i'', q_i''', q_i'''' \in \{-1,0,1\}$. Whilst being a simple extension of the radix-2 method, it obviously suffers from the many calculations that have to be performed during each iteration, and so is not very efficient.

In [57] Takagi improved upon this algorithm to give the following

$$\begin{aligned} r(i) &= 4 \cdot s(i) + x(i) \cdot Y \\ s(i+1) &= r(i) - 4 \cdot q_{i-1} \cdot N \end{aligned}$$

where $x(i), q_i \in \{-2,-1,0,1,2\}$. Thus, since $x(i)$ and q_i share the same range, a simple recoded RSD architecture (2-level adder array) can be used to implement the multiplier.

The problem with this approach however, is that to restrict q_i to this range requires 5 parallel comparisons of the top 8 bits of $s(i)$ and N. Since an 8-bit calculation requires more time to complete than the $2 \cdot \Delta_{FA}$ delay of the adder array, this method again fails to allow the adder array to operate at full speed.

In [58] Morita uses a similar algorithm that requires multiple concurrent comparisons involving the top 7 bits of $s(i)$ and N.

5.5 Radix-2^b Concurrent Multiply/Reduce Hardware

Generalizing the quotient estimation technique to b-bit high-radix modular multipliers helps us to understand the trade-offs that are implicit in this approach. The trade-off concerns three elements,

1. minimizing the calculation time of q_i,
2. minimizing the range of q_i, and
3. increasing b – the number of bits of X that are consumed during each iteration.

In general, trying to optimize just one of the above elements will have an adverse effect on the other two.

For example, in [59] with their VICTOR design, Orup et al. show that, for $X, Y < N$ all lb-bit positive integers, i.e.

$$X = [X_{l-1}, X_{l-2}, \ldots, X_0]$$

where $X_i \in [0, 2^b - 1]$, and using the following algorithm

$$\begin{aligned} r(i) &= 2^b \cdot s(i) + X_{l-i-1} \cdot Y \\ s(i+1) &= r(i) - 2^{2b} \cdot q_{i-1} \cdot N \end{aligned}$$

then using a method to calculate q_i that involves a multiplication of the top δ bits of $s(i)$ and the top ϵ bits of $1/N$ an equation governing the design tradeoffs is given in [59] as

follows

$$q_{MAX} = \left\lfloor \frac{2^b(2^{b+3-\delta} + 1 + 2^{1-b})}{1 - 2^{b-\epsilon}} \right\rfloor$$

For fixed values of b the above equation is asymptotic in q_{MAX} as δ and ϵ are varied. Table 5.1 shows the minimum values of δ and ϵ necessary to achieve minimum q_{MAX} for $b = 1 \ldots 6$. Optimized designs are then derived by noting the following,

b	δ	ε	q_{MAX}
1	6	5	4
2	8	7	6
3	10	10	10
4	12	12	18
5	14	13	34
6	16	15	66

Table 5.1: Minimum δ and ϵ for minimum q_{MAX}.

1. All integers in the range $0 \ldots 10$ can be expressed as the sum (or difference) of at most two powers of 2. i.e. $3 = 2^1 + 2^0$, $7 = 2^3 - 2^0$, and $10 = 2^3 + 2^1$. Thus multiples $q_i \cdot N$ for $q_i \in [0, 10]$ can be expressed as the sum of two partial sub-multiples $q_i' \cdot N + q_i'' \cdot N$ with $q_i' \in \{0, 2, 4, 8\}$ and $q_i'' \in \{-1, 0, 1, 2\}$ that are easy to generate in hardware.

2. All integers in the range $0 \ldots 42$ can be expressed as the sum (or difference) of at most three powers of 2. i.e. $11 = 2^3 + 2^1 + 2^0$, $27 = 2^5 - 2^2 - 2^0$, and $42 = 2^5 + 2^3 + 2^1$. Thus multiples $q_i \cdot N$ for $q_i \in [0, 42]$ can be expressed as the sum of three partial sub-multiples $q_i' \cdot N + q_i'' \cdot N + q_i''' \cdot N$ with $q_i' \in \{0, 8, 16, 32\}$, $q_i'' \in \{-8, -4, 0, 4, 8\}$ and $q_i''' \in \{-2, -1, 0, 1, 2\}$ that are easy to generate in hardware.

By selecting a b for which the asymptotic value of q_{MAX} is less than one of the 'optimum' values of 10 or 42, and then reducing δ and ϵ such that q_{MAX} is equal to (or slightly less than) one of these values, an 'optimum' trade-off in terms of q_i calculation time and range can be achieved for each selection of b. These values are shown in Table 5.2. In terms of multiplier circuit complexity and the number of iterations required to complete a multiplication, the optimum values for b in Table 5.2 are $b = 3$ and $b = 5$. This is because

b	δ	ε	q_{MAX}
1	3	3	10
2	5	6	10
3	10	10	10
4	7	7	38
5	10	11	42
6	—	—	—

Table 5.2: Optimum δ and ε.

the maximum values of b for which the ranges of q_i are $[0, 10]$ and $[0, 42]$ are $b = 3$ and $b = 5$ respectively.

The architecture used for VICTOR is CSA with a limited amount of parallelism in the organization of the 3:2 adders. A general diagram is shown in Figure 5.8. With reference

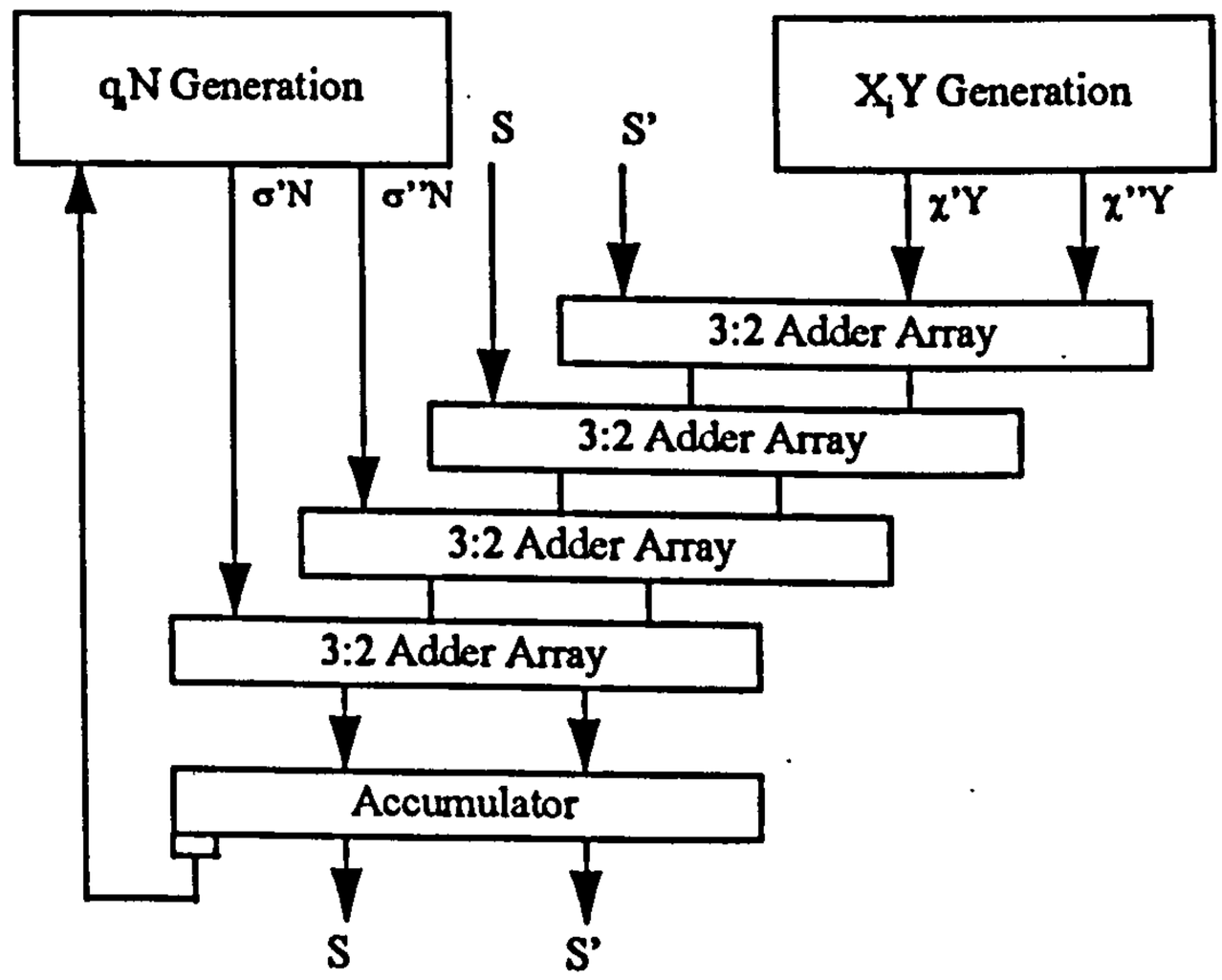


Figure 5.8: VICTOR architecture.

to the diagram, and for the cases of $b = 1 \ldots 5$, the adder array is constructed as follows

- $b = 1$: The multiplicand multiple $\chi' \cdot Y = x_{l-i-1} \cdot Y$, while $\chi'' \cdot Y = 0$. Therefore the top 3:2 adder array shown in the diagram is not required. The modulus multiples, $\sigma' \cdot N$ and $\sigma'' \cdot N$, are generated as in the $0 \ldots 10$ sum-of-two-powers-of-2 coding scheme described above.

  Thus a 3-level adder array is required.

- $b = 2$: The multiplicand multiples, $\chi' \cdot Y$ and $\chi'' \cdot Y$, are derived from $X_{l-i-1} \cdot Y$ in the obvious way ($\chi' = X_{l-i-1,(0)}$ and $\chi'' = X_{l-i-1,(1)}$). The modulus multiples, $\sigma' \cdot N$ and $\sigma'' \cdot N$, are generated as in the sum-of-two-powers-of-2 coding scheme.

  Thus a 4-level adder array is required.

- $b = 3$: The multiplicand multiples, $\chi' \cdot Y$ and $\chi'' \cdot Y$, are derived from the 3-bit multiple $X_{l-i-1} \cdot Y$ by using the sum-of-two-powers-of-2 coding method. The modulus multiples, $\sigma' \cdot N$ and $\sigma'' \cdot N$, are again generated by the sum-of-two-powers-of-2 method.

  Thus a 4-level adder array is again required.

- $b = 4$: The multiplicand multiples, $\chi' \cdot Y$ and $\chi'' \cdot Y$, are derived from the 4-bit multiple $X_{l-i-1} \cdot Y$ by a combination of the sum-of-three-powers-of-2 coding technique and a row of 3:2 adders to reduce the three powers-of-2 vectors to just two vectors. Thus a 1-level adder array is 'hidden' inside the $X_i \cdot Y$ generation sub-block. The modulus multiples, $\sigma' \cdot N$ and $\sigma'' \cdot N$, are also generated by the sum-of-three-powers-of-2 plus 1-level adder method. Therefore a 1-level adder array is also hidden inside the $q_i \cdot N$ generation sub-block.

  Thus 6 1-level adders are used in the design, but because the generation of the multiples of N is paralleled with the main adder array the delay through the whole array is equal to $5 \cdot \Delta_{FA}$.

- $b = 5$: The multiplicand multiples, $\chi' \cdot Y$ and $\chi'' \cdot Y$, are derived from the 5-bit multiple $X_{l-i-1} \cdot Y$ again by a combination of the sum-of-three-powers-of-2 coding technique and a row of 3:2 adders. The modulus multiples, $\sigma' \cdot N$ and $\sigma'' \cdot N$, are also generated by the sum-of-three-powers-of-2 method and a 3:2 adder array.

  Therefore, 6 1-level adders are again required to implement the design and the delay through the whole array is equal to $5 \cdot \Delta_{FA}$.

Examining the q_i calculation requirements for the more area-efficient designs of $b = 3$ and $b = 5$, we see that in each case approximately 10-bits of $s(i)$ and $1/N$ are required to be multiplied together to yield a value for the approximated quotient. Even with the non-trivial sum-of-two-powers-of-2 and sum-of-three-powers-of-2 coding techniques being used, this calculation is likely to take longer than the delay time of the adder array. If this is not the case (maybe for $b = 5$), then we still cannot say that the adder array is running at full speed because it will be the complex coding schemes that limit the cycle time of the multiplier – particularly the coding of the modulus multiple $q_i \cdot N$.

In [60] Orton et al. propose a Diminished Radix (DR) modular multiplier. In this scheme the k-bit modulus, N, is modified to produce a new modulus, M. The modification is to multiply N by a relatively small number T, such that

$$M = T \cdot N = 2^{k+c} - A$$

where T is chosen so that $A < 2^k$. Therefore T is, at most, a $(c+1)$-bit number. This modification ensures that the top c-bits of M are all '1's.

The multiplication algorithm then proceeds as follows,

$$\begin{aligned} r(i) &= 2^b \cdot s(i) + X_{l-i-1} \cdot Y \\ s(i+1) &= |r(i)|_{2^{k+c}} + 2^b \cdot q_{i-1} \cdot A \end{aligned}$$

where $q_i = \left\lfloor \frac{s(i)}{2^{k+c}} \right\rfloor$ and $l = \left\lceil \frac{k+c}{b} \right\rceil$. As before, the notation $|r(i)|_{2^{k+c}}$ means discard the overflow bits of $r(i)$ beyond the $(k+c)$-th bit.

Implied by the equation for q_i (and proved in [60]) is that the range of q_i depends to a certain extent on the value of c. That is, the larger the 'modulus extension' c, the smaller the range of q_i. This is in comparison with Orup et al. 's method which increased the precision of the q_i calculation to limit its range. Orton et al. found optimum values for c that depend only on the multiplier bit-scan width b. They are

- $b = 1$: $c = 6$ gives $q_i \in [0, 3]$, and

- $b = 2 \ldots 10$: $c = 2b + 5$ gives $q_i \in [0, 2^{b+2} - 1]$.

These are shown for $b = 1 \ldots 6$ in Table 5.3. On comparison of Table 5.2 (the VICTOR

b	c	q_{MAX}
1	6	3
2	9	15
3	11	31
4	13	63
5	15	127
6	17	255

Table 5.3: Optimum values for b, c and q_{MAX}.

design) with Table 5.3 we can see that, in the DR case, a design trade-off has been made in favour of simplifying the calculation of q_i at the expense of increasing its range and slightly increasing the size of the modulus. The latter trade-off, although slightly increasing the number of iterations required by the algorithm, is trivial because the total number of iterations required for long-integer multiplication is large. The former trade-off, that of increasing the range of the approximated quotient q_i, can be alleviated to some extent by partitioning q_i into 2-bit sub-blocks and holding values for both A and $3A$ on-chip. Then each 2-bit digit of q_i can use the stored value of A to generate the multiples A and $2A$, and the stored value of $3A$ to generate the remaining multiple of $3A$. This is effectively a hybrid quotient-estimation/table approach with the size of the table less than any of the previously reviewed table-residue schemes.

For example, with $b = 2$ and $q_i \in [0, 15]$, then q_i can be expressed as

$$q_i = 4 \cdot q_i' + q_i''$$

with $q_i', q_i'' \in \{0, 1, 2, 3\}$. A simple diagram of the $b = 2$ CSA architecture is shown in Figure 5.9.

Thus the following implementations are possible,

- $b = 1$: With partial product $x_{l-i-1} \cdot Y$ and partial residue $q_i \cdot A$ where $q_i \in \{0, 1, 2, 3\}$, then a 2-level CSA is required.

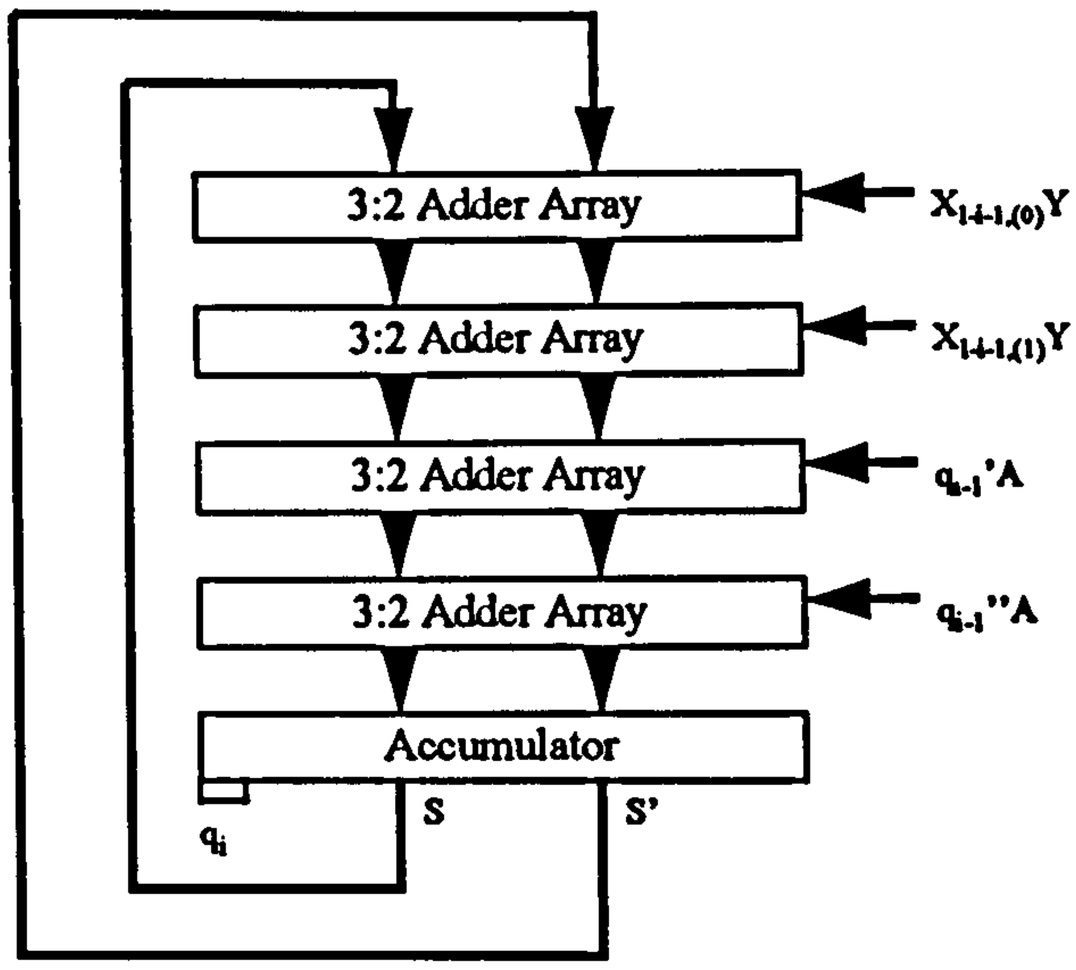


Figure 5.9: DR $b = 2$ architecture.

- $b = 2$: With partial products $X_{l-i-1} \cdot Y = 2 \cdot X_{l-i-1,(1)} \cdot Y + X_{l-i-1,(0)} \cdot Y$ and partial residues $q_i \cdot A = 4 \cdot q_i' \cdot A + q_i'' \cdot A$ where $q_i', q_i'' \in \{0,1,2,3\}$, then a 4-level CSA is required

- $b = 3$: With partial products $X_{l-i-1} \cdot Y = 4 \cdot X_{l-i-1,(2)} \cdot Y + 2 \cdot X_{l-i-1,(1)} \cdot Y + X_{l-i-1,(0)} \cdot Y$ and partial residues $q_i \cdot A = 16 \cdot q_i' \cdot A + 4 \cdot q_i'' \cdot A + q_i''' \cdot A$ where $q_i' \in \{0,1\}$ and $q_i'', q_i''' \in \{0,1,2,3\}$, then a 6-level CSA is required

- $b = 4$: With partial products $X_{l-i-1} \cdot Y = 8 \cdot X_{l-i-1,(3)} \cdot Y + 4 \cdot X_{l-i-1,(2)} \cdot Y + 2 \cdot X_{l-i-1,(1)} \cdot Y + X_{l-i-1,(0)} \cdot Y$ and partial residues $q_i \cdot A = 16 \cdot q_i' \cdot A + 4 \cdot q_i'' \cdot A + q_i''' \cdot A$ where $q_i', q_i'', q_i''' \in \{0,1,2,3\}$, then a 7-level CSA is required

- $b = 5$: With partial products $X_{l-i-1} \cdot Y = 16 \cdot X_{l-i-1,(4)} \cdot Y + 8 \cdot X_{l-i-1,(3)} \cdot Y + 4 \cdot X_{l-i-1,(2)} \cdot Y + 2 \cdot X_{l-i-1,(1)} \cdot Y + X_{l-i-1,(0)} \cdot Y$ and partial residues $q_i \cdot A = 64 \cdot q_i' \cdot A + 16 \cdot q_i'' \cdot A + 4 \cdot q_i''' \cdot A + q_i'''' \cdot A$ where $q_i' \in \{0,1\}$ and $q_i'', q_i''', q_i'''' \in \{0,1,2,3\}$, then a 9-level CSA is required

On comparison with VICTOR we see that the DR method is good for $b \leq 2$ but that it needs more adders for $b > 2$. This is because the trade-off in the DR design has been made in favour of the speed of q_i calculation, as opposed to the VICTOR design where

the restriction of the range of q_i and a more efficient multiple-encoding scheme have been developed.

To assimilate q_i, from a $(b+2)$-digit CSA vector, to binary form requires adder logic with carries propagating through all $b+2$ positions. Attempting to complete this assimilation before $q_i \cdot A$ is needed may still not be possible. This is why Orton et al. suggested using a pipelined approach for high-radix multiplication. Using this approach the addition of the $q_i \cdot A$ multiples is delayed by p cycles such that the assimilation of q_i has had time to complete. The algorithm can be described as follows,

$$\begin{aligned} r(i) &= 2^b \cdot s(i) + X_{l-i-1} \cdot Y \\ s(i+1) &= |r(i)|_{2^{k+c}} + 2^{pb} \cdot q_{i-p} \cdot A \end{aligned}$$

where

$$q_{i-p} = \begin{cases} 0 & \text{for } i-p < 0 \\ \left\lfloor s(i-p)/2^{k+c} \right\rfloor & \text{for } 0 \leq i-p < l+p \end{cases}$$

and also, since the modulus multiples are not subtracted immediately from the partial result, c must be increased by pb bits such that

$$c = (p+2)b + 5$$

The algorithm now requires $l+p$ iterations.

The architecture for implementing this multiplier is a concurrent pipelined design where separate data-paths are used for the partial products and partial residues. For example, whilst the pipeline design is not made explicit in [60], a possible architecture for the case of $b = 4$ and $p = 3$ using multiples X_{l-i-1} and q_i such that

$$\begin{aligned} X_{l-i-1} &= 8 \cdot X_{l-i-1,(3)} + 4 \cdot X_{l-i-1,(2)} + 2 \cdot X_{l-i-1,(1)} + X_{l-i-1,(0)} \\ q_i &= 16 \cdot q_i' + 4 \cdot q_i'' + q_i''' + q_i'''' \end{aligned}$$

where $q_i', q_i'' \in \{0,1,2,3\}$ use stored multiples of A and $3A$, and $q_i''', q_i'''' \in \{0,1\}$ use only multiples of A is shown in Figure 5.10.

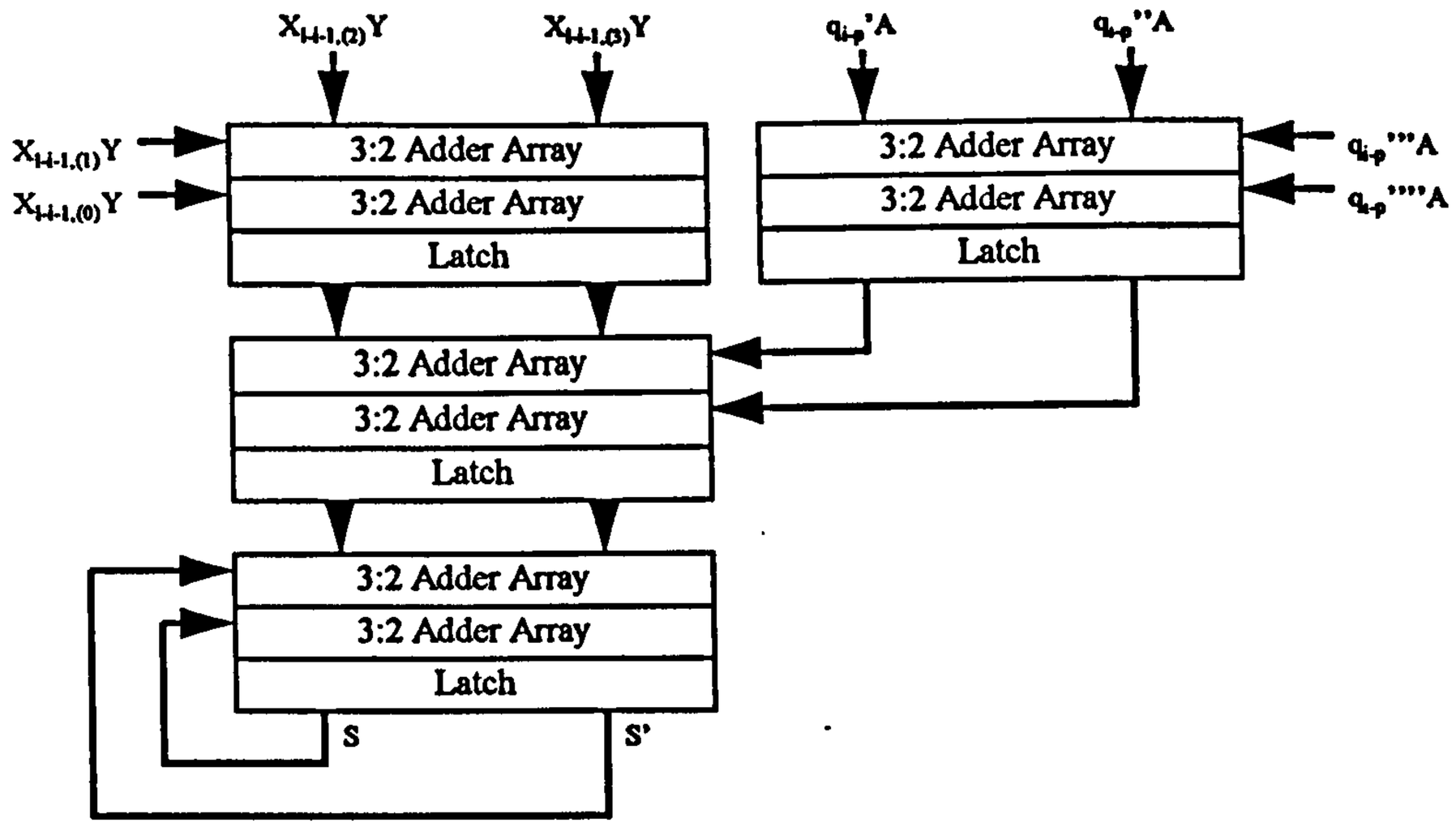


Figure 5.10: 3-stage pipelined, $b = 4$ DR multiplier.

Using this concurrent pipelined technique it may well be possible to run the multiplier at full speed (i.e. a speed determined solely by the delay inherent in each adder array – not dependent upon quotient estimation delays) but the cost in hardware terms is large since to achieve this 'break-even point' requires a large, complex multiplier. Also, with a large number of pipeline stages the multiplier becomes inefficient because of the extra iterations required to fill and empty the multiplier at the beginning and end of a multiplication respectively. Nevertheless, the DR method is probably the most promising high-speed method proposed so far among the 'standard' modular multiplication algorithms.

5.6 Other Proposed Systems

A few companies have manufactured RSA chips without publishing details of the algorithms and techniques used in their design. The most recent survey of known and working RSA chips was conducted by Brickell in [61]. In this survey performance figures were quoted for all designs and it should be noted that none of them outperformed the Cryptech chip reviewed above.

Other radix-2 designs include [62] [63] [64] and [65]. The latter uses a super-fast experimental 150MHz silicon-on-insulator technology to achieve claimed rates of over 64kbps.

A simple radix-2 multiple subtraction algorithm is used but the precise details of its operation are not given. It is thought that the algorithm is probably not very efficient, and the speed of the device is directly attributable to the implementation technology. An efficient algorithm implemented with this technology would no doubt yield much higher encryption rates.

Other proposed designs based on 'standard' algorithms include Sedlak's [66] complex '0'-skip multiplier/reducer incorporating barrel-shifters and complex control circuitry, Iwamura's [50] systolic array modular multiplier using localized ROM-table lookup, Kochanski's [67] processor array, and Prasanna's [68] highly parallel residue reduction and selection method. Others designs, more suitable for short-word modular multipliers, have been proposed by Alia [69] and Piestrak [70]. A small review of some of these (and other) techniques can be found in [71]. They have not been included in any detail here because they are thought to be either too complex or too inefficient for VLSI design.

5.7 Summary

This chapter reviewed the current literature concerning the implementation of RSA cryptosystems using standard modular multipliers. Descriptions of radix-2, radix-4 and general radix-2^b multipliers were included. In depth desciptions of the VICTOR [59] and DR [60] designs were given showing the trade-offs that have to be made in an effort to create a fast and efficient design. The problem of quotient estimation was identified as the core limiting factor in these designs. Designs with simple quotient estimation algorithms suffer from an extended quotient range which leads to more adders being necessary to sum the modulus multiple which in turn leads to increased multiplier circuitry and longer addition times. Designs with complex quotient estimation circuitry are limited by the time taken to calculate the next modulus multiple.

In Chapter 7 it will be shown that these limitiations can be removed with optimised

Montgomery multipliers.

Chapter 6

Montgomery Arithmetic

In 1985 Peter Montgomery published a paper [72] showing that it is possible to perform modular arithmetic modulo N without having to perform divisions by N. The technique relies upon a non-standard representation of the residues modulo N and is explained in the following sections.

6.1 Montgomery Multiplication

As was shown in Chapters 2 and 3, the standard method of multiplying two positive integers X and Y modulo N is,

$$\langle X \cdot Y \rangle_N = X \cdot Y - Q \cdot N \tag{6.1}$$

where

$$Q = \left\lfloor \frac{X \cdot Y}{N} \right\rfloor$$

and so $\langle X \cdot Y \rangle_N \in [0, N-1]$.

The Montgomery product, P (an integer), of the integers X and Y can be expressed as follows,

$$P = \frac{X \cdot Y + Z \cdot N}{R} \tag{6.2}$$

where R is a constant coprime to, and greater than, N. The integer Z can be viewed as the number of N's that have to be added onto $X \cdot Y$ in order to make the sum $X \cdot Y + Z \cdot N$ a multiple of R and thus make P an integer. We know that there is such a multiple Z that can do this because R and N are coprime.

If N is an odd number (in RSA $N = p \cdot q$ is odd) then to satisfy the coprimeness constraint we can make R a power of 2. Specifically if N is a k-bit integer

$$2^{k-1} < N < 2^k$$

then make

$$R = 2^k$$

and thus we see that, in a binary system, division by R in Equation 6.2 is now a trivial matter.

6.1.1 Calculating Z

Since $X \cdot Y + Z \cdot N$ is a multiple of R, then

$$X \cdot Y + Z \cdot N \equiv 0 \pmod{R}$$

and so

$$Z \cdot N \equiv -X \cdot Y \pmod{R}$$

Now since $\gcd(R, N) = 1$, therefore there exists an integer N^{-1} such that

$$N \cdot N^{-1} \equiv 1 \pmod{R}$$

therefore

$$Z \equiv -X \cdot Y \cdot N^{-1} \pmod{R}$$

Since Z is in the numerator of Equation 6.2, we will take the least non-negative residue of Z modulo R in order to limit the range of P, thus

$$Z = \left\langle -X \cdot Y \cdot N^{-1} \right\rangle_R$$

or, with pre-computed constant $N' = \langle -N^{-1} \rangle_R$ dependant only upon N and R, then

$$Z = \langle X \cdot Y \cdot N' \rangle_R$$

noting that, since $R = 2^k$, modular reduction modulo R is simple in a binary system.

Thus, if $X, Y < N$ and $Z < R$, the maximum value of P

$$P_{MAX} < \frac{N \cdot N + R \cdot N}{R}$$

is greater than N but less than $2N$, therefore the range of P is certainly limited to

$$P \in [0, 2N)$$

6.1.2 Interpreting P

We have calculated the integer P, but what exactly does it mean? Well, from Equation 6.2 we have

$$P \cdot R = X \cdot Y + Z \cdot N$$

therefore

$$P \cdot R \equiv X \cdot Y \pmod{N}$$

and since $\gcd(R, N) = 1$ there exists an integer R^{-1} such that

$$R \cdot R^{-1} \equiv 1 \pmod{N}$$

therefore

$$P \equiv X \cdot Y \cdot R^{-1} \pmod{N}$$

In other words P is the product of X and Y and a constant R^{-1} modulo N. The constant R^{-1} depends only upon R and N and does not vary with X and Y. Thus, the Montgomery method allows us to calculate a number that is related, by a constant, to the product of two integers modulo a third integer, and whose range is much reduced from that of the product. Therefore it should be possible to use Montgomery multiplications in algorithms that require modular multiplication but that do not make any decisions based on the

results of these multiplications. A post-conversion operation can then be performed at the end of the algorithm to remove the accumulated constants. As we saw in Section 3.3.2, exponentiation algorithms fall into this category.

A pseudo-code interpretation of Montgomery multiplication with reduction of the product P to the range $[0, N-1]$ is shown in Figure 6.1. Thus, on completion of the code, $P = \langle X \cdot Y \cdot R^{-1} \rangle_N$. Declaring reduction modulo R and division by R to be trivial

```
1.  t1 := X * Y
2.  t2 := < t1 >R
3.  Z := < t2 * N' >R
4.  t3 := Z * N
5.  t4 := t1 + t3
6.  P := t4 / R
7.  IF P >= N
8.     P := P - N
9.  ENDIF
```

Figure 6.1: Montgomery modular multiplication.

operations, then we see from the pseudo-code that Montgomery multiplication involves three multiplications (one of them modulo R) and a possible subtraction. Comparing this with standard modular reduction where a multiplication and a division is required we see that Montgomery multiplication effectively 'trades-off' a division in favour of two multiplications. Since one of these multiplications is modulo R, which when implemented using multi-precision operations is less time-consuming than standard multiplication, and multiplication algorithms are inherently faster than division algorithms anyway, then we can see that the computational complexity of the Montgomery approach may well be quite attractive. Selection of either standard or Montgomery algorithms depends very much on the specifics of the implementation environment. As we shall see in the remaining chapters, the Montgomery approach (with certain optimizations) can lead to very efficient modular exponentiators in VLSI.

6.2 Montgomery Exponentiation

Looking again at the modular exponentiation algorithms of Section 3.3.2 we have, for the Right-to-Left algorithm, $s(0) = 1$, $t(0) = A$

$$\begin{aligned} s(i+1) &= \langle s(i) \cdot (t(i))^{e_i} \rangle_N \\ t(i+1) &= \left\langle (t(i))^2 \right\rangle_N \end{aligned}$$

re-writing this in pseudo-code form in Figure 6.2 we see from this code that the $\langle S \cdot T \rangle_N$

```
1.  S := 1, T := A
2.  FOR i := 0 TO k-1
3.    IF e_i = 1
4.      S := < S * T >_N
5.    ENDIF
6.    T := < T^2 >_N
7.  ENDFOR
```

Figure 6.2: Right-to-Left modular exponentiation.

operation is performed only when $e_i = 1$ whilst the $\langle T^2 \rangle_N$ operation is performed on every iteration. If Montgomery multiplications are used instead of standard modular multiplications then each multiplication will introduce the constant factor of R^{-1} into its result.

For example, the variable $t(i)$ is modified as

$$t(i+1) = \left\langle (t(i))^2 \cdot R^{-1} \right\rangle_N$$

Thus, since in standard exponentiation $t(i) = \left\langle A^{2^i} \right\rangle_N$, in Montgomery exponentiation $t(i)$ will be

$$\begin{aligned} t(i) &= \left\langle A^{2^i} \cdot \left(R^{-1}\right)^i \right\rangle_N \\ &= \left\langle A^{2^i} \cdot R^{-i} \right\rangle_N \end{aligned}$$

Substituting this into the expression for evaluating $s(i+1)$ and also using Montgomery multiplication in this calculation we have

$$s(i+1) = \left\langle s(i) \cdot \left(A^{2^i} \cdot R^{-i}\right)^{e_i} \cdot \left(R^{-1}\right)^{e_i} \right\rangle_N$$

$$= \left\langle s(i) \cdot \left(A^{2^i} \cdot R^{-(i+1)}\right)^{e_i} \right\rangle_N$$

$$= \left\langle s(i) \cdot \left(A^{2^i e_i} \cdot R^{-e_i(i+1)}\right) \right\rangle_N$$

which when multiplied over $i = 0 \ldots k-1$ gives

$$s(k) = \left\langle A^E \cdot R^{-\sum_{i=0}^{k-1} e_i(i+1)} \right\rangle_N$$

Thus the result can be post-converted to $\left\langle A^E \right\rangle_N$ by Montgomery multiplying $s(k)$ by the 'constant' $\left\langle R^{\varepsilon+1} \right\rangle_N$ where

$$\varepsilon = \sum_{i=0}^{k-1} e_i(i+1)$$

and the $+1$ term in the exponent of $R^{\varepsilon+1}$ negates the effect of the constant R^{-1} introduced in the post-conversion Montgomery multiplication. Thus the 'constant', $\left\langle R^{\varepsilon+1} \right\rangle_N$, depends only on R, N and E. It does not depend on A.

A similar post-conversion constant that depends only on R, N and E can be derived for the Left-to-Right exponentiation method.

Although the above method is quite acceptable for modular exponentiations with N and E constant, another more general method is available that makes use of both pre- and post-conversions with all intermediate numbers represented in a special 'N-residue' form.

6.2.1 N-residue Representation

In [72] Montgomery shows how numbers in the range $[0, N-1]$ can be converted into N-residue form where the modular operations of multiplication and addition can be performed using Montgomery and standard methods respectively, such that the results of these operations are also in N-residue form. At the end of processing the results can be post-converted back to their normal residue representation modulo N.

Consider the constant

$$H = \left\langle R^2 \right\rangle_N$$

that depends only on R and N. Substituting H for Y in the Montgomery multiplication

algorithms we get

$$\begin{aligned} P &= \left\langle X \cdot H \cdot R^{-1} \right\rangle_N \\ &= \left\langle X \cdot R^2 \cdot R^{-1} \right\rangle_N \\ &= \langle X \cdot R \rangle_N \end{aligned}$$

So for two positive integers, X and Y, in the range $[0, N-1]$ two new integers can be calculated, X' and Y', via Montgomery multiplication such that

$$\begin{aligned} X' &= \left\langle X \cdot H \cdot R^{-1} \right\rangle_N = \langle X \cdot R \rangle_N \\ Y' &= \left\langle Y \cdot H \cdot R^{-1} \right\rangle_N = \langle Y \cdot R \rangle_N \end{aligned}$$

Now, if the product of X and Y using standard modular multiplication is $W = \langle X \cdot Y \rangle_N$, then converting this product as we did for X and Y gives

$$W' = \left\langle W \cdot H \cdot R^{-1} \right\rangle_N = \langle X \cdot Y \cdot R \rangle_N$$

But now looking at the Montgomery product of X' and Y', we get

$$\begin{aligned} \left\langle X' \cdot Y' \cdot R^{-1} \right\rangle_N &= \left\langle (X \cdot R) \cdot (Y \cdot R) \cdot R^{-1} \right\rangle_N \\ &= \langle X \cdot Y \cdot R \rangle_N \\ &= W' \end{aligned}$$

Thus we see that the Montgomery product of two 'converted' integers, X' and Y', is the same as the 'conversion' of the standard modular product of the two integers X and Y. If the integers X, Y and W are said to be in standard residue form, then their counterparts X', Y' and W' are said to be in N-residue form.

The implication of the above is that,

1. conversion of an integer, X, from standard residue form to N-residue form is achieved either by

   (a) standard modular multiplication of X by R, or

(b) Montgomery multiplication of X by $H = \langle R^2 \rangle_N$.

2. the Montgomery product of any two integers in N-residue form yields a result that is also in N-residue form,

3. the standard modular addition of any two integers in N-residue form yields a result that is also in N-residue form, and

4. conversion of an integer, X', from N-residue form to standard residue form is achieved either by

   (a) standard modular multiplication of X' by $\langle R^{-1} \rangle_N$, or

   (b) Montgomery multiplication of X' by 1.

At the beginning of the chapter it was stated that the Montgomery technique uses a non-standard representation of residue classes. This can be understood as follows. Since the conversion of an integer $X \in [0, N-1]$ to its N-residue form corresponds to the calculation $X' = \langle X \cdot R \rangle_N$ then, because R and N are coprime, this operation can be viewed as a re-ordering of the numbers $0 \ldots N-1$. That is, if X is viewed as a variable that can range over the interval $[0, N-1]$, then letting X take on the numbers $0 \ldots N-1$ in order leads to its N-residue representation X' taking on the numbers $0 \ldots N-1$ in a different order.

For example, with $N = 21$ and $R = 32$, Table 6.1 shows the standard and N-residue representations of X as it ranges over $[0, N-1]$.

6.2.2 N-residue Exponentiation

Using the notation that X' refers to the integer X in N-residue form, and denoting the Montgomery product of two integers X and Y modulo N with the constant R as

$$\mathcal{M}_{R,N}(X, Y) = \left\langle X \cdot Y \cdot R^{-1} \right\rangle_N$$

X	std residue $\langle X \rangle_{21}$	N-residue $\langle 32X \rangle_{21}$
0	0	0
1	1	11
2	2	1
3	3	12
4	4	2
5	5	13
6	6	3
7	7	14
8	8	4
9	9	15
10	10	5
11	11	16
12	12	6
13	13	17
14	14	7
15	15	18
16	16	8
17	17	19
18	18	9
19	19	20
20	20	10

Table 6.1: Standard and N-residue representations for $N = 21$ and $R = 32$.

then the conversion of an integer, X, to N-residue form is

$$X' = \mathcal{M}_{R,N}(X, H)$$

and conversion of X' from N-residue form to standard residue form is

$$X = \mathcal{M}_{R,N}(X', 1)$$

As before, H is the pre-computed constant $H = \langle R^2 \rangle_N$.

Thus, the Right-to-Left and Left-to-Right modular exponentiation algorithms of Section 3.3.2 can be modified to use Montgomery multiplications by including pre- and post-conversions into and out of N-residue form respectively (see for example [73]).

Algorithm 13 (Right-to-Left Montgomery N-residue Exponentiation) *Given an integer A, a positive k-bit exponent E, modulus N, constant R and pre-computed constant $H = \langle R^2 \rangle_N$, then calculating $\langle A^E \rangle_N$ is a 3-stage process. Setting*

$$s(0) = 1, \quad t(0) = A$$

then

1. Pre-conversion

$$s'(0) = \mathcal{M}_{R,N}(s(0), H)$$

$$t'(0) = \mathcal{M}_{R,N}(t(0), H)$$

2. Processing (for $i = 0 \ldots k-1$)

$$s'(i+1) = \begin{cases} s'(i) & \text{if } e_i = 0 \\ \mathcal{M}_{R,N}(s'(i), t'(i)) & \text{if } e_i = 1 \end{cases}$$

$$t'(i+1) = \mathcal{M}_{R,N}(t'(i), t'(i))$$

3. Post-conversion

$$s(k) = \mathcal{M}_{R,N}(s'(k), 1)$$

results in $s(k) = \left\langle A^E \right\rangle_N$.

A pseudo-code version of this algorithm is shown in Figure 6.3.

```
1.  S := 1, T := A
2.  S' := M_R,N(S,H)
3.  T' := M_R,N(T,H)
4.  FOR i := 0 TO k-1
5.    IF e_i = 1
6.      S' := M_R,N(S',T')
7.    ENDIF
8.    T' := M_R,N(T',T')
9.  ENDFOR
10. S := M_R,N(S',1)
```

Figure 6.3: R-to-L Montgomery N-residue exponentiation.

Algorithm 14 (Left-to-Right Montgomery N-residue Exponentiation) *Given an integer A, a positive k-bit exponent E, modulus N, constant R and pre-computed constant $H = \langle R^2 \rangle_N$, then calculating $\left\langle A^E \right\rangle_N$ is a 3-stage process. Setting*

$$s(0) = 1$$

then

1. Pre-conversion

$$A' = \mathcal{M}_{R,N}(A, H)$$

$$s'(0) = \mathcal{M}_{R,N}(s(0), H)$$

2. Processing (for $i = 0 \ldots k-1$)

$$s'(i+1) = \begin{cases} \mathcal{M}_{R,N}(s'(i), s'(i)) & \text{if } e_{k-i-1} = 0 \\ \mathcal{M}_{R,N}(\mathcal{M}_{R,N}(s'(i), s'(i)), A') & \text{if } e_{k-i-1} = 1 \end{cases}$$

3. Post-conversion

$$s(k) = \mathcal{M}_{R,N}(s'(k), 1)$$

results in $s(k) = \left\langle A^E \right\rangle_N$.

A pseudo-code version of this algorithm is shown in Figure 6.4.

```
1.  S := 1
2.  A' := M_R,N(A,H)
3.  S' := M_R,N(S,H)
4.  FOR i := 0 TO k-1
5.    S' := M_R,N(S',S')
6.    IF e_k-i-1 = 1
7.      S' := M_R,N(S',A')
8.    ENDIF
9.  ENDFOR
10. S := M_R,N(S',1)
```

Figure 6.4: L-to-R Montgomery N-residue exponentiation.

6.3 Iterative Montgomery Multiplication

Algorithms that perform Montgomery multiplication in an iterative fashion, similar to the algorithms of Section 3.3.1, exist. These algorithms consume the multiplier, X, b-bits at a time for radix-2^b multiplication.

We will look first at the $b = 1$ radix-2 multiplication algorithm, and then generalize this to the b-bit radix-2^b algorithm.

6.3.1 Radix-2 Montgomery Multiplication

The following algorithm can be found in Montgomery's original paper [72].

Algorithm 15 (Radix-2 Montgomery Multiplication) *Given a constant* $R = 2^k$, *odd modulus* $N < R$, *and two positive integers* $X, Y \in [0, N-1]$, *then setting*

$$s(0) = 0$$

and letting

$$s(i+1) = \frac{s(i) + x_i \cdot Y + z_i \cdot N}{2}$$

with

$$z_i = \langle s(i) + x_i \cdot Y \rangle_2$$

will give

$$s(k) = \frac{X \cdot Y + Z \cdot N}{R}$$

where $Z = \langle X \cdot Y \cdot N' \rangle_R$ *for* $N' = \langle -N^{-1} \rangle_R$. *Also, the quantities* $z_i \in \{0, 1\}$ *for* $i = 0 \ldots k-1$ *form the* k *bits of* Z *in its bit-vector representation* $Z = [z_{k-1}, z_{k-2}, \ldots, z_0]$.

Proof: From the above we have

$$2 \cdot s(i+1) = s(i) + x_i \cdot Y + z_i \cdot N$$

For fixed k this gives

$$\begin{aligned} 2 \cdot s(k) &= s(k-1) + x_{k-1} \cdot Y + z_{k-1} \cdot N \\ &= \frac{s(k-2) + x_{k-2} \cdot Y + z_{k-2} \cdot N}{2} + x_{k-1} \cdot Y + z_{k-1} \cdot N \end{aligned}$$

therefore

$$2^2 \cdot s(k) = s(k-2) + x_{k-2} \cdot Y + z_{k-2} \cdot N + 2 \cdot x_{k-1} \cdot Y + 2 \cdot z_{k-1} \cdot N$$

and, by extension, for variable α

$$2^\alpha \cdot s(k) = s(k-\alpha) + \sum_{i=0}^{\alpha-1} 2^{\alpha-i-1} \cdot x_{k-i-1} \cdot Y + \sum_{i=0}^{\alpha-1} 2^{\alpha-i-1} \cdot z_{k-i-1} \cdot N$$

For $\alpha = k$ this gives

$$2^k \cdot s(k) = s(0) + \sum_{i=0}^{k-1} 2^{k-i-1} \cdot x_{k-i-1} \cdot Y + \sum_{i=0}^{k-1} 2^{k-i-1} \cdot z_{k-i-1} \cdot N$$

changing the order of summation and noting that $s(0) = 0$, $2^k = R$ and

$$\sum_{i=0}^{k-1} 2^i \cdot x_i = X$$

then

$$R \cdot s(k) = X \cdot Y + \sum_{i=0}^{k-1} 2^i \cdot z_i \cdot N$$

Assuming here that

$$\sum_{i=0}^{k-1} 2^i \cdot z_i = Z$$

(it will be proven as a special case in the general radix-2^b algorithm's proof), then

$$s(k) = \frac{X \cdot Y + Z \cdot N}{R}$$

■

Note that the above algorithm proceeds in a Right-to-Left direction along the multiplier X. This is in contrast to all the standard modular multiplication algorithms reviewed in Chapter 5 which proceed in a Left-to-Right direction.

A visual interpretation of the way in which this algorithm works can be understood as follows. First, re-write the main loop of the algorithm as

$$r(i) = s(i) + x_i \cdot Y$$

$$s(i+1) = \frac{r(i) + z_i \cdot N}{2}$$

where now $z_i = \langle r(i) \rangle_2$. We can think of the calculation of $r(i)$ as being the 'multiplication' part of each iteration, and the subsequent calculation of $s(i+1)$ as being the 'reduction' part of each iteration. Now writing r(i) as a $(k+1)$-bit bit-vector

$$r(i) = [r_k^{(i)}, r_{k-1}^{(i)}, \ldots, r_0^{(i)}]$$

then the goal of the reduction part of each iteration is simply to stop $r(i)$ from growing in magnitude. It tries to do this by dividing $r(i)$ by 2 (i.e. a right-shift of $r(i)$ by one bit position) but if $r(i)$ is an odd number (i.e. $r_0^{(i)} = 1$ so that $z_i = 1$) then it cannot do this and keep the partial result $s(i+1)$ an integer. To overcome this problem $r(i)$ must first be converted to an even number so that $r_0^{(i)} = 0$. Since we are working modulo N the only number that we can add to $r(i)$ without affecting the result is N. Also, since N is an odd number then its addition to the odd $r(i)$ will yield an even number and so enable the reduction (division by 2) to take place. This happens on every iteration of the algorithm.

An alternative way of performing Montgomery multiplication is to completely separate the multiplication and reduction phases of the algorithm. Thus if $T = X \cdot Y$ is the $2k$-bit product of two k-bit positive integers, such that

$$T = [t_{2k-1}, t_{2k-2}, \ldots, t_0]$$

then, in a manner directly analogous to the above algorithm, T can be Montgomery reduced to the range $[0, 2N)$ by proceeding in the direction $i = 0 \ldots k-1$ and, for every $t_i = 1$, adding $2^i \cdot N$ onto T. At the end of this process the lower k bits of T will all be zero. If T is then right-shifted by k bit positions we will have $T \equiv X \cdot Y \cdot 2^{-k} \equiv X \cdot Y \cdot R^{-1} \pmod{N}$.

6.3.2 Radix-2^b Montgomery Multiplication

The radix-2 Montgomery multiplication algorithm can be generalized to the radix-2^b case as follows.

Algorithm 16 (Radix-2^b Montgomery Multiplication) *Given a constant $R = 2^{lb}$, odd modulus $N < R$, and two positive integers $X, Y \in [0, N-1]$, expressing X as the l-digit vector $X = [X_{l-1}, X_{l-2}, \ldots, X_0]$ with $X_i \in [0, 2^b - 1]$, then setting*

$$s(0) = 0$$

and letting

$$s(i+1) = \frac{s(i) + X_i \cdot Y + Z_i \cdot N}{2^b}$$

with

$$Z_i = \langle (s(i) + X_i \cdot Y) \cdot N' \rangle_{2^b}$$

will give

$$s(l) = \frac{X \cdot Y + Z \cdot N}{R}$$

where $Z = \langle X \cdot Y \cdot N' \rangle_R$ *for* $N' = \langle -N^{-1} \rangle_R$. *Also, the quantities* $Z_i \in [0, 2^b - 1]$ *for* $i = 0 \ldots l-1$ *form the* l *digits of* Z *in its vector representation* $Z = [Z_{l-1}, Z_{l-2}, \ldots, Z_0]$.

Proof: Similar to the radix-2 proof we have, for variable α

$$2^{\alpha b} \cdot s(l) = s(l-\alpha) + \sum_{i=0}^{\alpha-1} 2^{\alpha-i-1} \cdot X_{l-i-1} \cdot Y + \sum_{i=0}^{\alpha-1} 2^{\alpha-i-1} \cdot Z_{l-i-1} \cdot N$$

which for $\alpha = l$, $s(0) = 0$, $2^{lb} = R$ and

$$\sum_{i=0}^{l-1} 2^{ib} \cdot X_i = X$$

gives

$$R \cdot s(l) = X \cdot Y + \sum_{i=0}^{l-1} 2^{ib} \cdot Z_i \cdot N$$

Now to show that

$$\sum_{i=0}^{l-1} 2^{ib} \cdot Z_i = Z$$

we can do the following. From the definition of the algorithm we have

$$Z_i \equiv (s(i) + X_i \cdot Y) \cdot N' \pmod{2^b}$$

then setting $i = l-1$ and recursing down into $s(i)$ we get

$$\begin{aligned} Z_{l-1} &\equiv (s(l-1) + X_{l-1} \cdot Y) \cdot N' \pmod{2^b} \\ &\equiv \left(\frac{s(l-2) + X_{l-2} \cdot Y + Z_{l-2} \cdot N}{2^b} + X_{l-1} \cdot Y \right) \cdot N' \pmod{2^b} \end{aligned}$$

therefore

$$2^b \cdot Z_{l-1} \equiv \left(s(l-2) + X_{l-2} \cdot Y + Z_{l-2} \cdot N + 2^b \cdot X_{l-1} \cdot Y \right) \cdot N' \pmod{2^{2b}}$$

and, in general for variable α

$$2^{\alpha b} \cdot Z_{l-1} \equiv \left(s(l-\alpha-1) + \sum_{i=0}^{\alpha} 2^{(\alpha-i)b} \cdot X_{l-i-1} \cdot Y + \sum_{i=1}^{\alpha} 2^{(\alpha-i)b} \cdot Z_{l-i-1} \cdot N \right) \cdot N' \pmod{2^{(\alpha+1)b}}$$

Setting $\alpha = l-1$ gives

$$2^{(l-1)b} \cdot Z_{l-1} \equiv \left(s(0) + \sum_{i=0}^{l-1} 2^{(l-i-1)b} \cdot X_{l-i-1} \cdot Y + \sum_{i=1}^{l-1} 2^{(l-i-1)b} \cdot Z_{l-i-1} \cdot N \right) \cdot N' \pmod{2^{lb}}$$

Since $s(0) = 0$, $2^{lb} = R$ and changing the order of summation gives

$$2^{(l-1)b} \cdot Z_{l-1} \equiv \left(\sum_{i=0}^{l-1} 2^{ib} \cdot X_i \cdot Y + \sum_{i=0}^{l-2} 2^{ib} \cdot Z_i \cdot N \right) \cdot N' \pmod{R}$$

Now, since $N \cdot N' \equiv -N \cdot N^{-1} \equiv -1 \pmod{R}$ then

$$2^{(l-1)b} \cdot Z_{l-1} \equiv X \cdot Y \cdot N' - \sum_{i=0}^{l-2} 2^{ib} \cdot Z_i \pmod{R}$$

and therefore

$$\sum_{i=0}^{l-1} 2^{ib} \cdot Z_i \equiv X \cdot Y \cdot N' \pmod{R}$$

Since $Z_i \in [0, 2^b - 1]$ the left-hand-side of the above equation is in the range $[0, 2^{lb} - 1]$

and so

$$\sum_{i=0}^{l-1} 2^{ib} \cdot Z_i = \langle X \cdot Y \cdot N' \rangle_R = Z$$

Returning to the main algorithm we therefore have

$$s(l) = \frac{X \cdot Y + Z \cdot N}{R}$$

■.

For the special case of $b = 1$, that is radix-2 multiplication, we have

$$z_i = \langle (s(i) + x_i \cdot Y) \cdot N' \rangle_2$$

but since $N \cdot N' = -1 + q \cdot R$ for some q, then with $R = 2^{lb}$ the right-hand-side of the above equation is an odd number, and therefore so is the left-hand-side. Since N is odd this means that N' also must be odd and so $\langle N' \rangle_2 = 1$. Thus the equation for z_i will simplify to

$$z_i = \langle s(i) + x_i \cdot Y \rangle_2$$

6.4 Montgomery Multiplier Implementations

The Montgomery multiplication technique has been implemented in a few systems that perform RSA cryptography. These are as follows.

6.4.1 Multi-precision Implementations

In [37] Dusse et al. show how the Montgomery technique can be implemented on a standard DSP chip. Multi-precision operations are based on the 24-bit word-size of the chip with Montgomery reduction performed simultaneously with the convolution-like method of multiplying large integers (see [74]).

The Montgomery method is also efficient for general software implementations of the RSA scheme [75].

6.4.2 Systolic Array Implementations

A systolic array is an n-dimensional array that consists of interconnected processing elements (PEs) such that each element only communicates with its immediate neighbours. For example a 1-dimensional systolic array is shown in Figure 6.5. As can be seen from

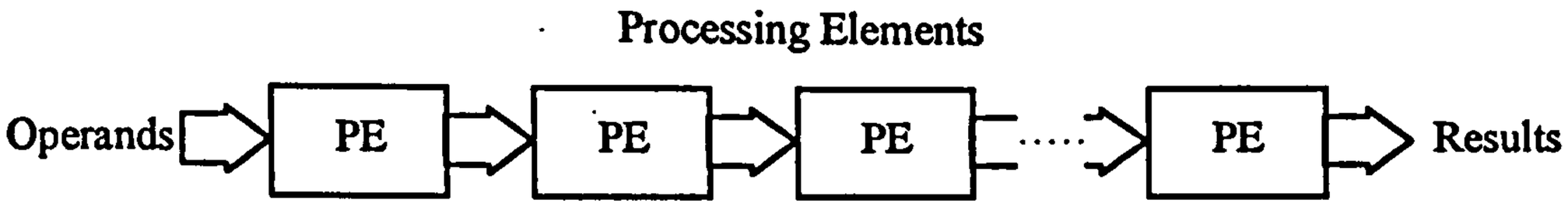


Figure 6.5: A 1-dimensional systolic array.

the diagram, the operands enter the array on the left and the results exit the array on the right. Each processing element performs some simple operation on the data entering it from the left, and sends the result to the next element on the right. All processing elements can be active simultaneously. The combined processing power of all these simple processing elements performs the desired transformation on the input operands. The systolic array gets its name from the movement of the data through the array which appears

to be a kind of 'pumping' action – similar to the pumping of the human heart – and thus systolic.

In [76], Even coupled a systolic multiplier with an array of his own design that performs Montgomery reduction. Assuming that the output of the multiplier array feeds directly into the reducer array, then the combination array takes $2k$ clock cycles to Montgomery multiply two k-bit integers modulo a k-bit modulus.

Sauerbrey, in [77], coupled a b-bit multiplier array and b-bit Montgomery reducer array so that each processing element operates on b-bits of data at a time. The array takes $2l$ clock cycles to complete the multiplication.

In [78], Iwamura developed a single systolic array that could perform Montgomery multiplication with each processing element operating on b bits of data. The array takes $2l$ clock cycles to complete.

The main advantage of using systolic arrays in VLSI design is that, although a global clock signal must still be distributed to each processing element, all other communications to/from processing elements are local (i.e. to/from neighbouring cells). This can simplify the routing and buffering requirements for a VLSI design. Specific to the problem of long-integer multiplier design and in particular with the parallel bit-slice approach that has been developed in this thesis, it means that the quantities X_i and Z_i do not have to be distributed to all processing elements simultaneously. Thus simplifying this aspect of the design. The disadvantages of systolic arrays for long-integer multiplication however are twofold. Firstly, the complexity of each processing element tends to be greater than that of a bit-slice element, and secondly, systolic arrays usually require twice the number of clock cycles to 'pump' the result out of the end of the array. These, when combined, tend to make systolic array implementations of modular multipliers less efficient than their parallel bit-slice counterparts.

6.4.3 A Pipelined Implementation

In [75] and [79] Shand et al. show how the Montgomery method can be implemented using pipelining. The method is similar to the pipeline technique used in the Diminished Radix design of Orton et al. in [60] (reviewed in Chapter 5), in that it is designed to stop the computation time of the Z_i calculation from affecting performance.

In [79] Shand delays the addition of the $Z_i \cdot N$ multiple until the $(i+p)$-th cycle where p is the level of pipelining used. This is done by modifying the algorithm and generating a slightly different Z_i, which we will call Z'_i, as follows.

With $s(0) = 0$, then let

$$s(i+1) = \left\lfloor \frac{s(i) + X_i \cdot Y}{2^b} \right\rfloor + \frac{s(i-p) + Z'_{i-p} \cdot N}{2^{(p+1)b}}$$

where $s(i-p) = 0$ for $i - p < 0$, and

$$Z'_{i-p} = \begin{cases} 0 & \text{for } i - p < 0 \\ \langle s(i-p) \cdot N' \rangle_{2^{(p+1)b}} & \text{for } i - p \geq 0 \end{cases}$$

then

$$s(l+p) \equiv X \cdot Y \cdot R^{-1} \pmod{N}$$

Note that the multiple Z'_i is generated such that, on the i-th cycle when $s(i)$ is 'sampled' and the calculation of $\langle s(i) \cdot N' \rangle_{2^{(p+1)b}}$ is started, then on the subsequent cycles $i+1, i+2, \ldots, i+p-1$ the lower bits of $s(i)$ are all 'predictable' in the sense that the intermediate additions of $Z'_{i-p+1} \cdot N$, $Z'_{i-p+2} \cdot N, \ldots, Z'_{i-1} \cdot N$ will not effect the group of b bits that, on the $(i+p)$-th iteration with the addition of $Z'_i \cdot N$, are set to zero in the intended manner by that multiple. This is why Z'_i must be a $(p+1)b$-bit quantity as opposed to the unpipelined b-bit quantity. But by delaying the addition of $Z'_i \cdot N$ then, it is claimed, enough time will be available to calculate this extended multiple.

The above method has been implemented by Orton et al. on a circuit board known as a Programmable Active Memory (PAM). The PAM consists of an array of Field Programmable Gate Array (FPGA) chips mounted on a slot-in computer card such that it is

suitable for use as a general-purpose co-processor engine for the host machine. Configuration of the PAM for any processing task can be achieved by first designing the appropriate FPGA circuitry (with suitable ECAD tools), generating the FPGA setup data from the design and then downloading this data to the board. The host computer and co-processor can then operate together to perform whatever specialized task was required.

Using such a PAM the fastest reported RSA processor to date has been constructed. When the prime components of the modulus $N = p \cdot q$ are known, then using the Chinese Remainder Theorem (which gives a speed-up factor of approx. 400% in hardware – see Section 2.6.5), it can encipher data at a rate of 600kbps for 512-bit moduli and 165kbps for 1024-bit moduli. Shand et al. claim that the design can be shrunk down into a single gate-array device for even higher performance, but this has yet to be done. Although they do not state the number of pipeline stages that were used in the PAM design, it is unlikely that a large number of such stages can be implemented in a single gate-array device.

6.5 Summary

This chapter introduced Montgomery modular arithmetic. The technique and its meaning were explained and basic algorithms for Montgomery multiplication and exponentiation were studied. A review of some of the proposed hardware implementation schemes was given.

The algorithms of this chapter together with the RSD architecture of Chapter 4 serve as the basis from which new optimised Montgomery multipliers are developed in the next chapter.

Chapter 7

Optimized Montgomery Multiplication

In this chapter new, optimized versions of Montgomery multipliers will be presented that allow the multiplier adder array to operate at full-speed. That is, the determination of Z_i (the number of N's to be added on each cycle) can be performed completely in parallel with the operation of the adder array.

Optimized multipliers are designed for the cases of

- $b = 1$ with a simple CSA type architecture,
- $b = 2$ using recoding techniques and an RSD architecture, and
- $b \geq 2$ general high-radix multipliers with recoding and RSD architectures.

Very recent work by Eldridge and Walter in [80], [81] and [82] has shown how to build optimized designs for non-recoded multipliers with a CSA architecture. The methods developed here go further than this, and use efficient multiplier recoding techniques with an RSD architecture. The methods of Eldridge and Walter and those shown here have been developed concurrently and the work presented in this chapter is independent of their own.

7.1 Radix-2 Multiplication

In Section 6.3 the following general radix-2^b algorithm was shown to perform Montgomery multiplication. It will be called AMMM (Additive Montgomery Modular Multiplication). With $s(0) = 0$, then letting

$$r(i) = s(i) + X_i \cdot Y$$

$$s(i+1) = \frac{r(i) + Z_i \cdot N}{2^b}$$

where $Z_i = \langle r(i) \cdot N' \rangle_{2^b}$ will give

$$s(k) = \frac{X \cdot Y + Z \cdot N}{R}$$

Implementing this algorithm directly in hardware would lead to the design of Figure 7.1. This diagram shows how the least-significant b bits of $r(i)$ – the output from the top level

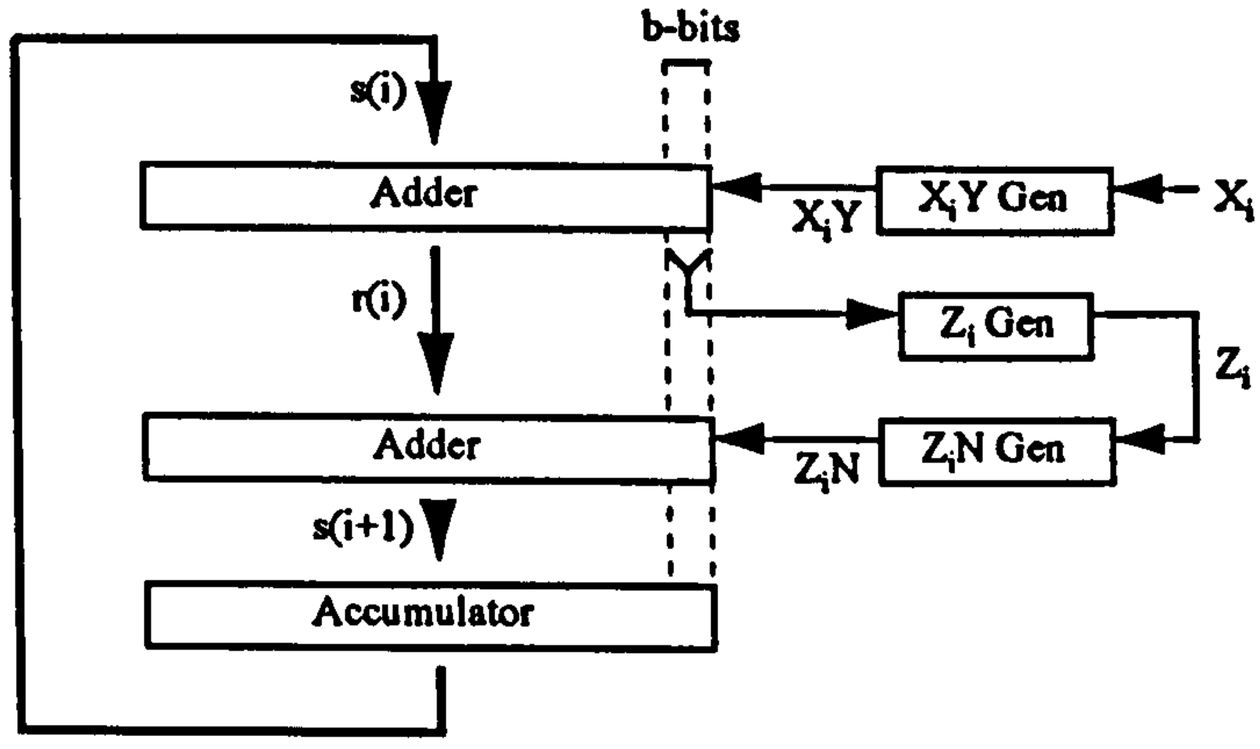


Figure 7.1: Implementation of AMMM.

of adders – is used to generate Z_i. The value of Z_i thus generated is then used to construct $Z_i \cdot N$ which is then added to $r(i)$ by the lower level of adders. The problem with this approach is that the generation of Z_i and $Z_i \cdot N$ takes place between the upper and lower adders, and so, although $r(i)$ is made available at the input of the lower adder immediately after it emerges from the upper adder, the presentation of $Z_i \cdot N$ to the adder inputs is delayed by the amount of time it takes to generate it, and thus the adder array cannot operate at full speed. In order to optimize the design, the generation of Z_i and $Z_i \cdot N$ must be taken out of the critical path of the adder array. This is achieved in the next section.

7.1.1 The DAMMM algorithm

The DAMMM (Delayed Additive Montgomery Modular Multiplication) algorithm allows $Z_i \cdot N$ to be generated in parallel with the operation of the top level of adders in the adder array. It can be specified as follows.

Algorithm 17 (DAMMM) *Given a constant $R = 2^{lb}$, odd modulus $N < R$, constant $N' = \langle -N^{-1} \rangle_R$, and two positive integers $X, Y \in [0, N-1]$, expressing X as the l-digit vector $X = [X_{l-1}, X_{l-2}, \ldots, X_0]$ with $X_i \in [0, 2^b - 1]$, then setting*

$$s(0) = 0$$

and letting

$$\begin{aligned} r(i) &= s(i) + 2^b \cdot X_i \cdot Y \\ s(i+1) &= \frac{r(i) + Z_i \cdot N}{2^b} \end{aligned}$$

with

$$Z_i = \langle s(i) \cdot N' \rangle_{2^b}$$

will give

$$s(l+1) = \frac{X \cdot Y + Z \cdot N}{R}$$

with $s(l+1) \in [0, 2N)$.

Proof: Noting that $X_i \cdot Y$ has been left-shifted by b bits (pre-multiplied by 2^b) in the calculation of $r(i)$, therefore $r(i) \equiv s(i) \pmod{2^b}$ and so Z_i can be calculated directly from $s(i)$.

On the first iteration $Z_0 = 0$ and because it takes an extra cycle for $X_i \cdot Y$ to be shifted down to the least-significant bits of the accumulator, then in general, Z_i in the DAMMM algorithm will be the same as Z_{i-1} in AMMM for $i = 1 \ldots l$. Therefore

$$\sum_{i=0}^{l} 2^{ib} \cdot Z_i = 2^b \cdot Z$$

where $Z = \langle X \cdot Y \cdot N' \rangle_R$, and since $X_l = 0$ we have

$$s(l+1) = \frac{2^b \cdot X \cdot Y + 2^b \cdot Z \cdot N}{2^b \cdot R} = \frac{X \cdot Y + Z \cdot N}{R}$$

■.

An implementation of the DAMMM algorithm is shown in Figure 7.2. Note that

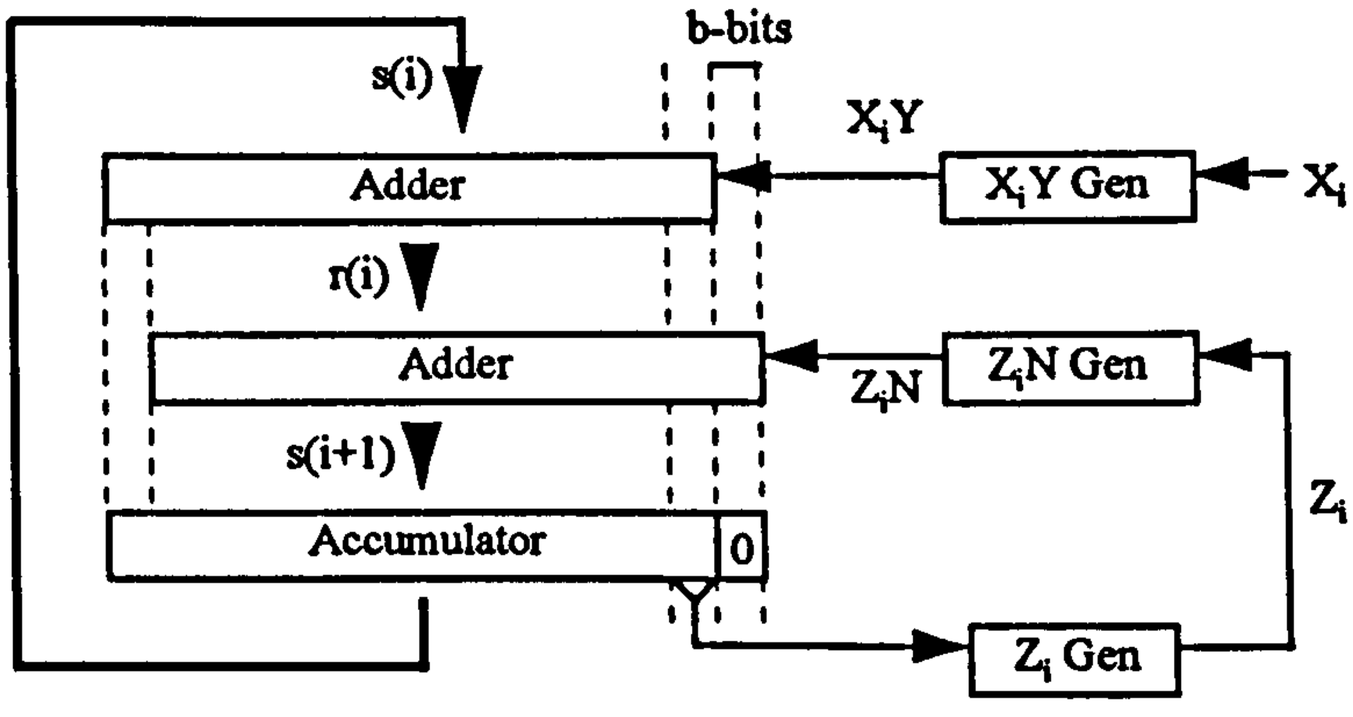


Figure 7.2: Implementation of DAMMM.

the lower b bits of the accumulator are all zero. This is essentially the function of adding $Z_i \cdot N$ — to ensure that the b least-significant bits of the accumulator are all zero prior to performing the b-bit right-shift operation embedded in the feedback of the accumulator's outputs to the top-level adder's inputs.

The diagram also shows that, with the partial product $X_i \cdot Y$ shifted left by b bits, the value of Z_i no longer depends directly on the output of the top adder. In accordance with Algorithm 17 Z_i is derived from $s(i)$ which, by retracing the feedback loop of $s(i)$ to its origin at the output of the accumulator (taking into account the b-bit right-shift built into the loop), can be seen to be equivalent to sampling $s(i)$ at the next to lowest b-bit block of accumulator output. Thus Z_i and $Z_i \cdot N$ generation can be performed in parallel with both the generation of $X_i \cdot Y$ and the operation of the top level adder.

A radix-2 implementation of DAMMM with a CSA architecture is shown in Figure 7.3. For N a k-bit positive integer, the adder inputs are defined as follows.

$$W = [w_{k-1}, w_{k-2}, \ldots, w_0]$$

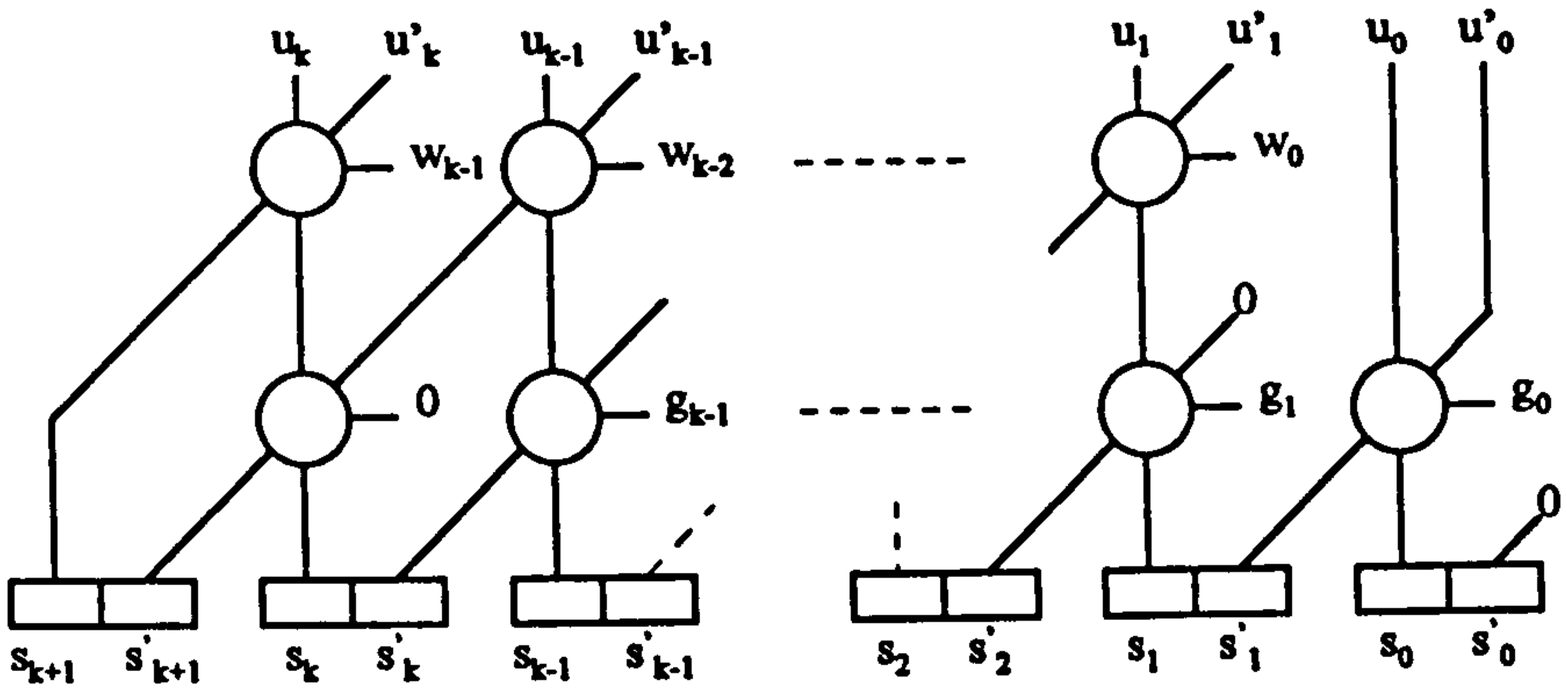


Figure 7.3: Radix-2 DAMMM CSA array.

$$= x_i \cdot Y$$

where $x_i \in \{0,1\}$ for iterations $i = 0 \ldots k-1$ and $x_k = 0$ on the final iteration.

$$G = [g_{k-1}, g_{k-2}, \ldots, g_0]$$

$$= z_i \cdot N$$

where $z_i \in \{0,1\}$ for iterations $i = 0 \ldots k$. The feedback loop maps register outputs to adder inputs with a right-shift as,

$$u_j \leftarrow s_{j+1}$$

$$u'_j \leftarrow s'_{j+1}$$

for $j = 0 \ldots k-1$. The s_{j+1} and s'_{j+1} outputs correspond to the sum and carry outputs respectively of the CSA accumulator.

The generation of the multiples $x_i \cdot Y$ and $z_i \cdot N$ is shown in Figure 7.4. Since each x_i

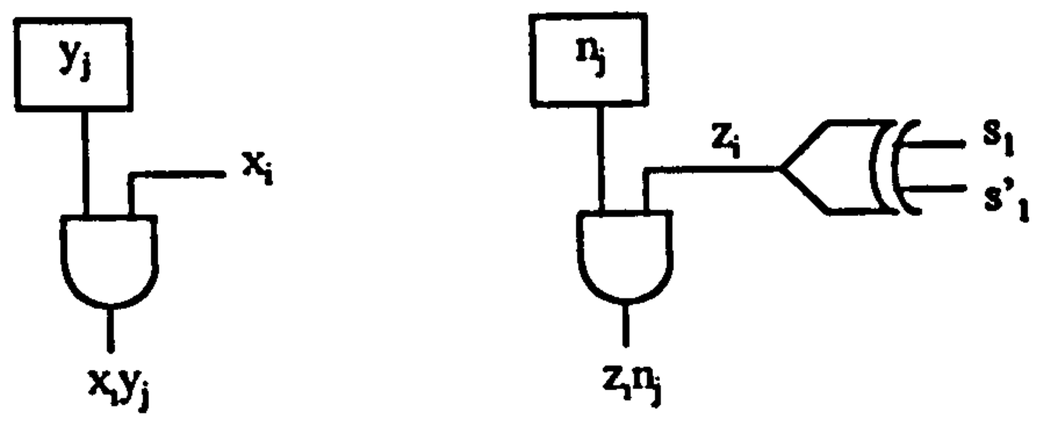


Figure 7.4: Radix-2 DAMMM $x_i \cdot Y$ and $z_i \cdot N$ generation.

is merely the i-th bit of X, and assuming that X is stored in a shift-register that is clocked by the same clock that operates the accumulator, then immediately after the active edge

of this clock the quantities x_i and $s(i)$ are available for use. A diagram showing the delay-path of signals around the DAMMM circuit is shown in Figure 7.5. Since the generation

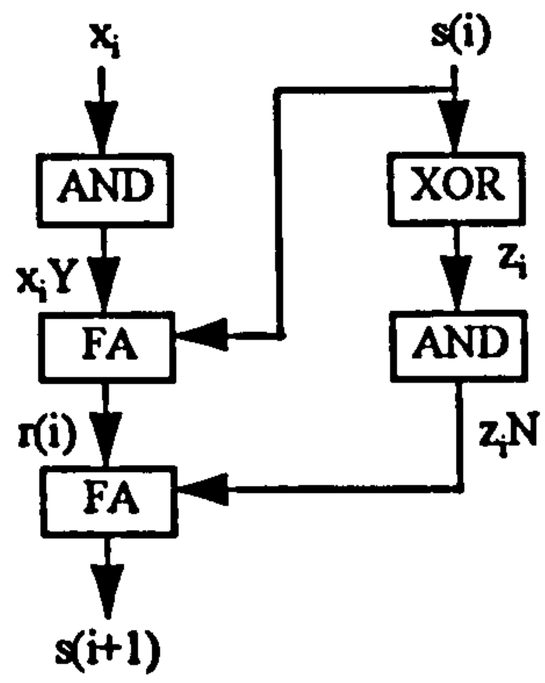


Figure 7.5: Radix-2 DAMMM delay-path.

of z_i is performed by an XOR gate and, with reference to the circuit diagram of the full adder in Figure 4.1, the delay through an XOR gate is less than that through a full adder, we can see that the multiple $z_i \cdot N$ will be presented to the second level of adders in the CSA adder array just before $r(i)$ is made available. Thus we have succeeded in removing the calculation of $z_i \cdot N$ from the critical path of the adder array. The 2-level CSA array can now operate at full speed.

7.1.2 Result Range

Although we have just created a fast radix-2 Montgomery multiplier, when used as part of a high-speed exponentiator the circuit will suffer from a major disadvantage. This is that the inputs are restricted to the range $[0, N-1]$ but that the result will occupy the range $[0, 2N)$. Thus it will not be possible to use the result of one multiplication as one of the operands in the next multiplication. To do this we have to ensure that the operand input ranges and result output ranges are the same.

Consider the size of N limited such that

$$N < \frac{R}{4}$$

and inputs X and Y limited to the extended range

$$X, Y \in [0, R/2)$$

with Z as before in the range $[0, R)$. Then we have

$$\frac{X \cdot Y + Z \cdot N}{R} < \frac{\frac{R}{2} \cdot \frac{R}{2} + R \cdot \frac{R}{4}}{R} = \frac{R}{2}$$

and thus the range of the result is the same as that of X and Y.

Since, at the beginning of an exponentiation, inputs to the multiplication routines will be in the range $[0, N)$ which by the above restriction on N is contained in the range $[0, R/2)$, this means that all of the intermediate results of multiplications during an exponentiation can be kept in the range $[0, R/2)$, and the multiplications can proceed one after another without any intermediate correction steps (such as a comparison and possible subtraction of N from a multiplication result) being necessary. Although a carry-propagated addition of the S and S' vectors is still required at the end of a multiplication, the removal of any comparison and possible subtraction of N considerably simplifies the circuitry of the multiplier. The final result of the exponentiation is the only one that has to be reduced to $[0, N-1]$.

7.1.3 Radix-2 DAMMM Performance Summary

Using the condition $4 \cdot N < R$ from the previous section, we see that for N a k-bit modulus we need $R \geq 2^{k+2}$. Since the DAMMM algorithm requires an extra iteration to complete, we can summarize the radix-2 CSA implementation as follows.

$$\begin{aligned} \text{Number of iterations} &= k+3 \\ \text{Iteration time} &= \Delta_{AND} + 2 \cdot \Delta_{FA} + \Delta_{FF} \\ \text{Number of bitslices} &= k+4 \\ \text{Bitslice complexity} &= 5 \cdot \Omega_{FF} + 2 \cdot \Omega_{AND} + 2 \cdot \Omega_{FA} \end{aligned}$$

7.2 Radix-4 Multiplication

Stating the DAMMM algorithm for radix-4 multiplication we have

$$\begin{aligned} r(i) &= s(i) + 4 \cdot X_i \cdot Y \\ s(i+1) &= \frac{r(i) + Z_i \cdot N}{4} \end{aligned}$$

with $X_i, Z_i \in \{0,1,2,3\}$ and where $Z_i = \langle s(i) \cdot N' \rangle_4$.

Applying string recoding to this algorithm we would first recode $X = [X_{l-1}, X_{l-2}, \ldots, X_0]$ using the technique of Section 4.3 so that

$$X = \sum_{i=0}^{l} 2^{2i} \cdot x(i)$$

where $x(i) \in \{-2,-1,0,1,2\}$. This can be accomplished either by working on one digit at a time in either the left-to-right or right-to-left directions or by calculating all digits at once in parallel. This freedom in implementing the recoding technique is exactly what makes it attractive for use in fast, non-modular, parallel multipliers and also in some of the left-to-right modular multipliers of Chapter 5. The reason why it is used to recode the multiplier X in some of these left-to-right modular multipliers is because the recoding process can be performed one digit at a time in the left-to-right direction as the multiplier is 'consumed' during multiplication. This leads to a more efficient multiplier design – compared to the parallel recoding approach – for long-integer applications. The reason why this recoding technique can be used in this way is because the recoding process does not generate any carries while it is being applied. Each digit of the second-order recoded vector depends only on groups of three adjacent bits of X. If carries were generated then, since carries propagate from right-to-left along a bit-vector, a carry generated at the i-th step of a left-to-right recoding process would necessitate the re-evaluation of all the previously generated $i-1$ digits. If however, a right-to-left multiplication technique were being used, then the generation of any carries during the recoding process would not matter. A carry generated when calculating the i-th digit of the recoded vector would simply be saved and used in the

next step to generate the $(i+1)$-th digit. Since Montgomery multiplication is essentially a right-to-left process, we can take advantage of this and develop a recoding technique that does indeed generate carries, but offers a reduced range for the recoded digits.

7.2.1 Recoding X

The following recoding technique differs from the technique of Section 4.3 in that, as explained above, it is only suitable for right-to-left multiplication algorithms. It is based around the simple observation that $3 = 4 - 1$ and $2 = 4 - 2$. The idea is that the digits of a radix-4 vector each have an associated weight. That is, for $X = [X_{l-1}, X_{l-2}, \ldots, X_0]$ a radix-4 vector, then its value is given by

$$X = \sum_{i=0}^{l-1} 2^{2i} \cdot X_i$$

where 2^{2i} is the weight associated with the i-th digit of the vector. This can be viewed as the i-th digit having weight 4-times that of its right-hand neighbour the $(i-1)$-th digit. The recoding procedure is then to look at each digit X_i for $i = 0 \ldots l-1$ and set the recoded digit $x(i)$ to either 0,1,−2 or −1 accordingly as to whether X_i is either 0,1,2 or 3. In the latter two cases, by the previous observations, a 1 must be added to the next X_{i+1} digit to compensate for the negative values assumed by $x(i)$ in this iteration. This is the carry. The next iteration must therefore take note of this carry and, calling it $c(i)$, leads to the following

$$x(i) = \begin{cases} 0 & \text{for } X_i + c(i-1) = 0 \\ 1 & \text{for } X_i + c(i-1) = 1 \\ -2 & \text{for } X_i + c(i-1) = 2 \\ -1 & \text{for } X_i + c(i-1) = 3 \\ 0 & \text{for } X_i + c(i-1) = 4 \end{cases}$$

for $i = 0 \ldots l$ with $c(-1) = 0$ and

$$c(i) = \begin{cases} 0 & \text{for } X_i + c(i-1) = 0 \\ 0 & \text{for } X_i + c(i-1) = 1 \\ 1 & \text{for } X_i + c(i-1) = 2 \\ 1 & \text{for } X_i + c(i-1) = 3 \\ 1 & \text{for } X_i + c(i-1) = 4 \end{cases}$$

for $i = 0 \ldots l-1$.

Thus we have a right-to-left recoding system that converts

$$X = \sum_{i=0}^{l-1} 2^{2i} \cdot X_i$$

with $X_i \in \{0,1,2,3\}$ to

$$X = \sum_{i=0}^{l} 2^{2i} \cdot x(i)$$

with $x(i) \in \{-2,-1,0,1\}$. Note the extra l-th digit in the recoded vector.

This recoding mechanism can be implemented as shown in Figure 7.6. Note that the

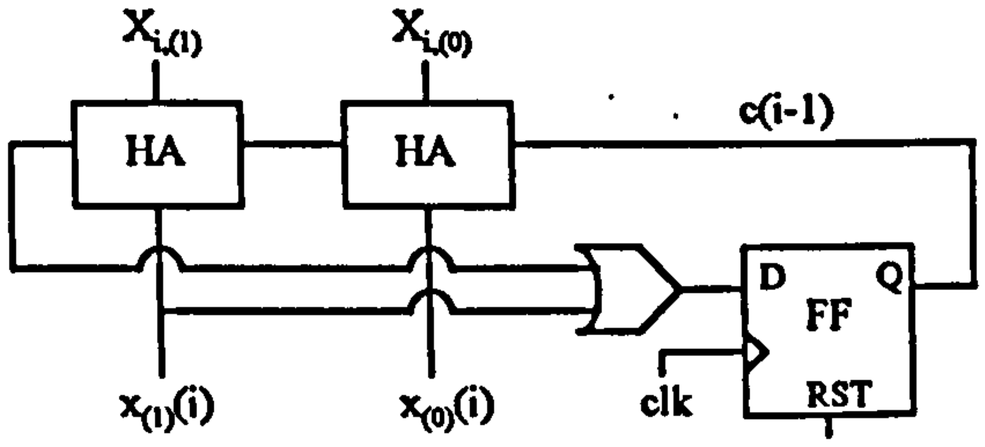


Figure 7.6: Recoding $X_i \in \{0,1,2,3\}$ to $x(i) \in \{-2,-1,0,1\}$.

bits of X are assumed to be stored in a 2-bit wide shift register which is clocked by the same clock as the flip-flop shown in the diagram. The circuit converts

$$X_i = 2 \cdot X_{i,(1)} + X_{i,(0)}$$

to the recoded digits

$$x(i) = -2 \cdot x_{(1)}(i) + x_{(0)}(i)$$

The $x(i) \cdot Y$ multiple generation circuit is shown in Figure 7.7. From this diagram we see that the $x(i) \cdot Y$ generation circuitry has been reduced from the MUX/AND configuration

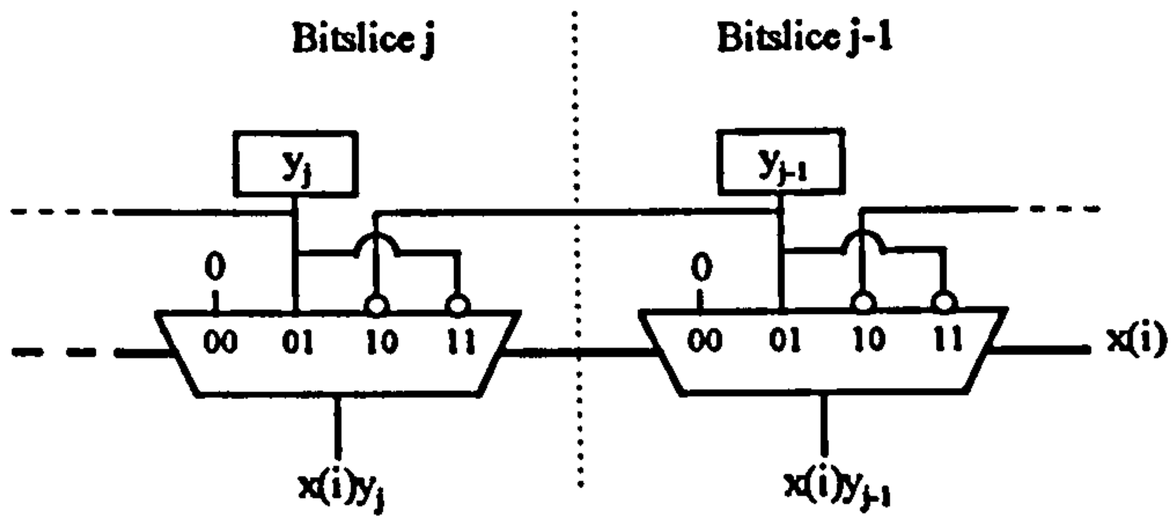


Figure 7.7: Multiple $x(i) \cdot Y$ generation.

of Figure 4.16 to just a single MUX. Also, the requirement for routing a special $\overline{\text{zero}}$ signal to each bitslice has been removed.

The only problem with using the recoded value, $x(i)$, instead of the original, X_i, is that the signals $x_{(1)}(i)$ and $x_{(0)}(i)$ are not available on the active edge of the shift-register clock. The delay of $2 \cdot \Delta_{HA}$ for these signals to become available would increase the cycle time of the multiplier. To get around this problem we can pipeline the generation of $x(i)$. A simple 1-stage pipeline is all that is needed and this is shown in Figure 7.8. The clock

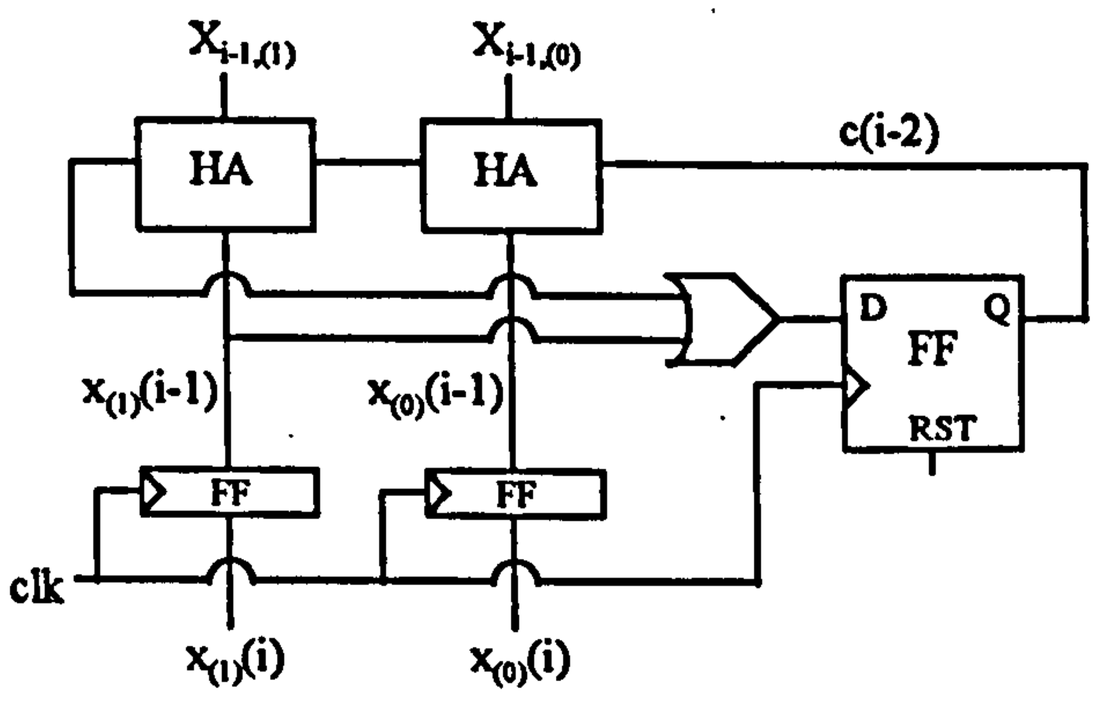


Figure 7.8: Pipelined generation of $x(i)$.

shown in the diagram is the same clock that is used by the X shift-register and by the multiplier's accumulator. Thus the $x(i)$ signals are able to proceed directly to the $x(i) \cdot Y$ multiplexors on the active edge of the clock and no overhead is added to the cycle time of the multiplication by using this recoding technique. Indeed, the delay of one AND gate has been removed from the critical path.

7.2.2 Recoding Z

On reviewing the theory of Montgomery multiplication we see that the only requirement on Z_i is that

$$r(i) + Z_i \cdot N \equiv 0 \pmod{2^b}$$

therefore Z_i may take on any set of values that covers the range $[0, 2^b - 1]$ modulo 2^b. This means that for $b = 2$ we may map

$$Z_i \in \{0, 1, 2, 3\}$$

onto

$$z(i) \in \{0, 1, -2, -1\}$$

the only difference this will make is that, if

$$Z = \sum_{i=0}^{l-1} 2^{2i} \cdot z(i)$$

then Z may take on positive and negative values so that

$$Z \in (-R, R)$$

This in turn means that

$$P = \frac{X \cdot Y + Z \cdot N}{R}$$

will also assume positive and negative values. However, because we are using signed-digit arithmetic this is not a problem. If N is limited as before such that

$$4 \cdot N < R$$

but now X and Y are limited such that

$$X, Y \in (-R/2, R/2)$$

then by a similar reasoning as was used before, we have

$$P \in (-R/2, R/2)$$

In other words, we allow the results of multiplications to be negative as well as positive but their ranges will not diverge during an exponentiation. As we saw in Section 4.5 we can easily perform signed-number multiplications using an RSD architecture.

7.2.3 An RSD Montgomery Multiplier

For X and Y signed-numbers in 2's complement form, then from the above we have

$$|Y| < \frac{R}{2}$$

If $R = 2^{lb}$ then X and Y can be expressed as lb-bit 2's complement bit-vectors.

For

$$W = x(i) \cdot Y$$

with $x(i) \in \{-2, -1, 0, 1\}$ then W can be expressed as a $(lb+1)$-bit 2's complement vector, so that

$$W = -2^{lb} \cdot w_{lb} + \sum_{j=0}^{lb-1} 2^j \cdot w_j + w_{-1}$$

where w_{-1} is the 'add one' term for when a negative multiple of Y is being created by the 'invert and add one' technique of 2's complement negation. This allows us to create negative multiples of Y without having to perform the carry propagation implied by the 'add one' instruction. (See the recoded multiplier of Section 4.5).

Similarly

$$G = z(i) \cdot N$$

with $z(i) \in \{-2, -1, 0, 1\}$ so that

$$G = -2^{lb} \cdot g_{lb} + \sum_{j=0}^{lb-1} 2^j \cdot g_j + g_{-1}$$

Using the DAMMM algorithm with an RSD architecture leads to the circuit shown in Figure 7.9. Note that at the top end of the adder array the G vector has been sign-extended by two bit positions.

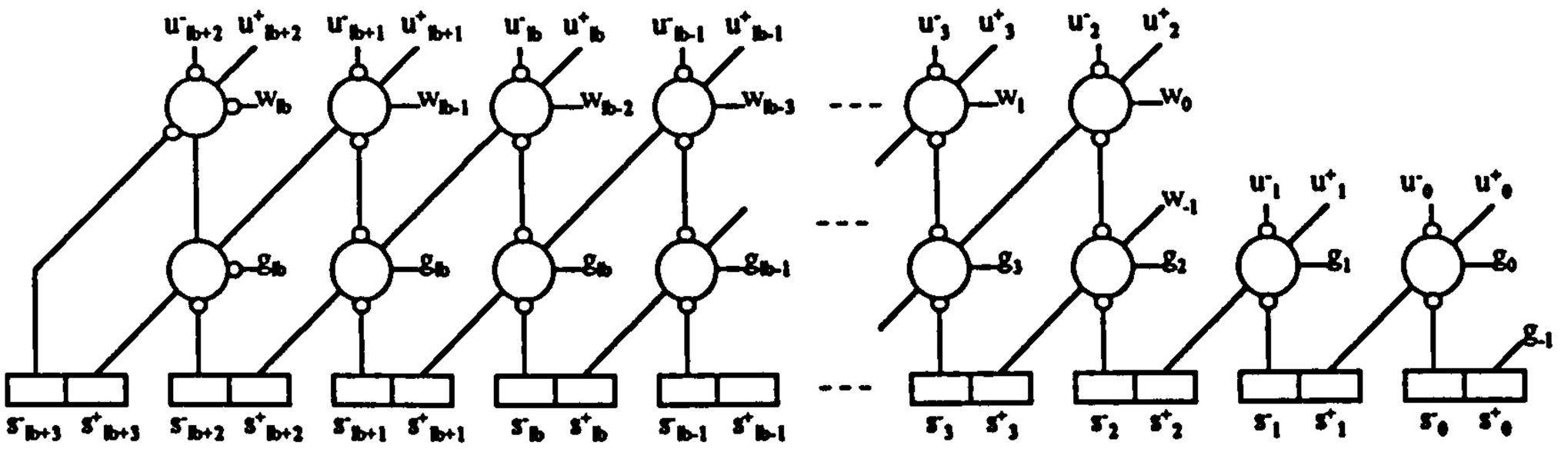


Figure 7.9: Radix-4 recoded DAMMM with RSD architecture.

A divide-by-4 is embedded in the feedback loop such that

$$u_j^- \leftarrow s_{j+2}^-$$

$$u_j^+ \leftarrow s_{j+2}^+$$

for $j = 0 \ldots lb+1$ with $u_{lb+2}^- = u_{lb+2}^+ = 0$.

Generating $x(i)$

To generate $x(i) \in \{-2, -1, 0, 1\}$ from the lb-bit 2's complement vector X then, since the DAMMM algorithm requires $l+1$ iterations, the first step is to sign-extend X to an $(l+1)b$-bit 2's complement vector X'. Since $b = 2$ this means creating two bits x'_{2l+1} and x'_{2l} so that

$$X' = -2^{2l+1} \cdot x'_{2l+1} + \sum_{j=0}^{2l} 2^j \cdot x'_j$$

has the same value as X. From Section 4.4.1 we see that

$$x'_{2l+1} = x'_{2l} = x_{2l-1}$$

and

$$x'_j = x_j$$

for $j = 0 \ldots 2l-1$. Therefore X' may be viewed as an $(l+1)$-digit vector $[X'_l, X'_{l-1}, \ldots, X'_0]$ with $X'_i \in \{0, 1, 2, 3\}$ for $i = 0 \ldots l-1$ but with $X'_l \in \{-1, 0\}$ because $X'_{l,(1)} = X'_{l,(0)}$.

On applying the left-to-right recoding technique we see that, when creating the l-th recoded digit,

$$X'_l + c(l-1) \in \{-1, 0, 1\}$$

Since this can be expressed by $x(l) \in \{-2, -1, 0, 1\}$ this means that no carry should be generated by the last recoded digit.

What this means in terms of the generating circuit of Figure 7.6 is that, as long as we sign-extend X to an $(l+1)$-digit vector, we can use the circuit unchanged by simply ignoring any $(l+2)$-th digit that it might generate.

Generating $z(i)$

To generate $z(i)$ we first have to calculate $Z_i = \langle s(i) \cdot N' \rangle_4$ then perform the mapping $\{0, 1, 2, 3\} \rightarrow \{0, 1, -2, -1\}$. For

$$z(i) = -2 \cdot z_{(1)}(i) + z_{(0)}(i)$$

this is shown in Table 7.1. A circuit to generate $z(i)$ is shown in Figure 7.10.

$<s(i)>_4$	$<N'>_4$	Z_i	$z(i)$	$z_{(1)}(i)$	$z_{(0)}(i)$
0	1	0	0	0	0
1	1	1	1	0	1
2	1	2	-2	1	0
3	1	3	-1	1	1
0	3	0	0	0	0
1	3	3	-1	1	1
2	3	2	-2	1	0
3	3	1	1	0	1

Table 7.1: Generating $z(i)$

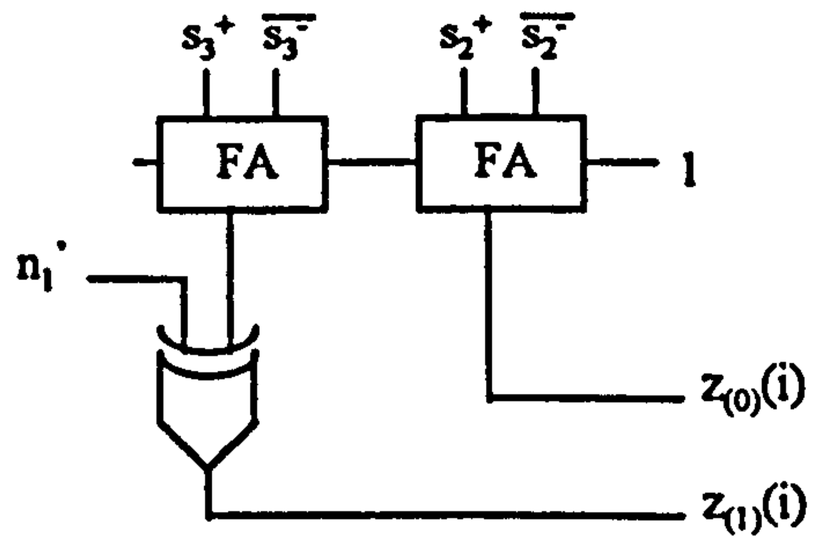


Figure 7.10: Circuit for generating $z(i)$.

Assembling all of these elements into a radix-4 Montgomery multiplier will produce the critical path delay diagram of Figure 7.11. However, from this diagram we can clearly

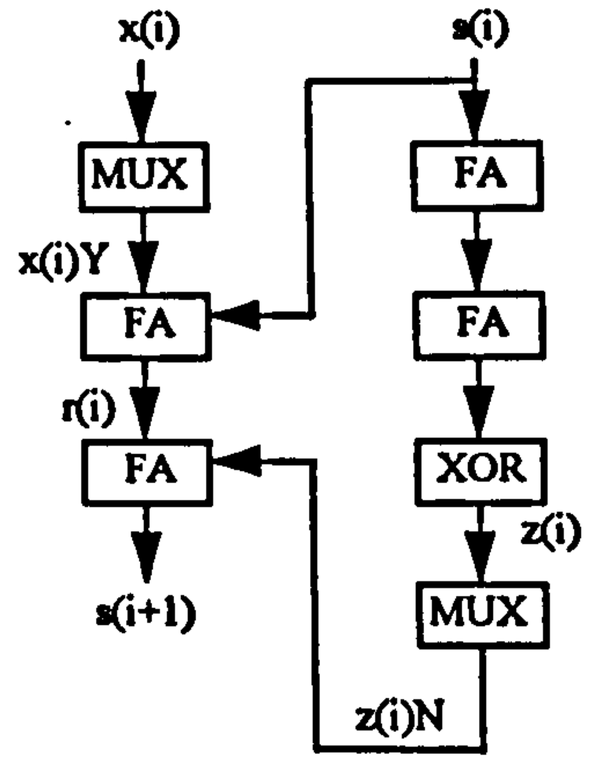


Figure 7.11: Radix-4 DAMMM delay path.

see that the time taken to calculate $z(i)$ is too long. The adder array will not be able to operate at full-speed. Other optimization methods are needed to speed up the calculation of $z(i)$, and they are presented in the following sections.

7.2.4 The MMDAMMM Algorithm

The MMDAMMM (Modified Modulus Delayed Additive Montgomery Modular Multiplication) algorithm enables us to remove the multiplicative part of the Z_i calculation. The basic idea is to create a new modulus, M, from the original modulus, N, by multiplying N by a small constant.

Note that, since M is just a simple multiple of N, the new modulus can be used in place of the old modulus in modular arithmetic calculations. This can be seen as follows. Supposing $M = T \cdot N$ then

$$\begin{aligned} a &\equiv b \pmod{M} \\ &= b + q \cdot M \\ &= b + q \cdot (T \cdot N) \\ &= b + (q \cdot T) \cdot N \\ &\equiv b \pmod{N} \end{aligned}$$

Thus to perform, say, an exponentiation modulo N, we could perform all of the exponentiation's multiplications modulo M, and then, at the end of processing, reduce the final result to its least non-negative residue modulo N.

As was suggested by Walter in [80], the new modulus M can be created such that $\langle M' \rangle_{2^b} = \pm 1$. In the case of Walter's systolic design this did not offer any significant improvements, but with the bitslice architecture presented in this thesis a considerable speedup can be achieved since the calculation of Z_i is much simplified. For the sake of definiteness, we will create M such that $\langle M' \rangle_{2^b} = +1$.

Consider a modified modulus M, such that

$$M = N \cdot \langle N' \rangle_{2^b}$$

Now

$$N' \equiv -N^{-1} \pmod{R}$$

and since $R = 2^{lb}$ therefore

$$N' \equiv -N^{-1} \pmod{2^b}$$

thus

$$M \equiv N \cdot (-N^{-1}) \equiv -1 \pmod{2^b}$$

Similarly

$$M' \equiv -M^{-1} \pmod{R}$$

therefore

$$M' \equiv -M^{-1} \pmod{2^b}$$

and substituting $M \equiv -1 \pmod{2^b}$ we have

$$M' \equiv -(-1)^{-1} \equiv 1 \pmod{2^b}$$

and so therefore $\langle M' \rangle_{2^b} = 1$ for any odd modulus N as required. Note that to create M we only had to multiply N by a b-bit quantity. Thus the growth in size from N to M is minimal.

Using the modified modulus M in the radix-4 DAMMM algorithm together with the convergence restriction that $4 \cdot M < R$ and the multiplier recoding techniques developed in the previous section, leads to the radix-4 recoded MMDAMMM algorithm as follows.

Algorithm 18 (Radix-4 recoded MMDAMMM) *Given a constant* $R = 2^{2l}$, *odd modulus* N *such that*

$$2^4 \cdot N < R$$

a constant $N' = \langle -N^{-1} \rangle_R$, *and a modified modulus* M *such that*

$$M = N \cdot \langle N' \rangle_4$$

Then two integers $X, Y \in (-R/2, R/2)$, *with* X *expressed as the* $(l+1)$*-digit vector* $X = [X_l, X_{l-1}, X_{l-2}, \ldots, X_0]$ *with* $X_i \in \{0,1,2,3\}$ *for* $i = 0 \ldots l-1$ *and* $X_l \in \{-1,0\}$, *can be Montgomery multiplied by setting*

$$s(0) = 0$$

and letting

$$\begin{aligned} r(i) &= s(i) + 4 \cdot x(i) \cdot Y \\ s(i+1) &= \frac{r(i) + z(i) \cdot M}{4} \end{aligned}$$

with $x(i) \in \{-2,-1,0,1\}$ *being the recoded digits of* X, *and* $z(i) \in \{-2,-1,0,1\}$ *such that*

$$z(i) \equiv s(i) \pmod 4$$

will give

$$s(l+1) \equiv X \cdot Y \cdot R^{-1} \pmod M$$

with $s(l+1) \in (-R/2, R/2)$.

Proof: The condition $2^4 \cdot N < R$ means that the modified modulus M will satisfy $4 \cdot M < R$. The rest follows from the proof of DAMMM together with the convergence restrictions and recoding techniques given in the previous sections ■.

7.2.5 Generating $z(i)$ under MMDAMMM

Referring back to Figure 7.10 we see that, using MMDAMMM, it is possible to remove the XOR gate from the $z(i)$ generation circuit. Furthermore, since the carry-in to right-hand full adder is a '1', and the carry-out of the left-hand adder is not used, we can simplify the logic of this circuit to that of Figure 7.12. Upon comparing the circuit of Figure 7.12

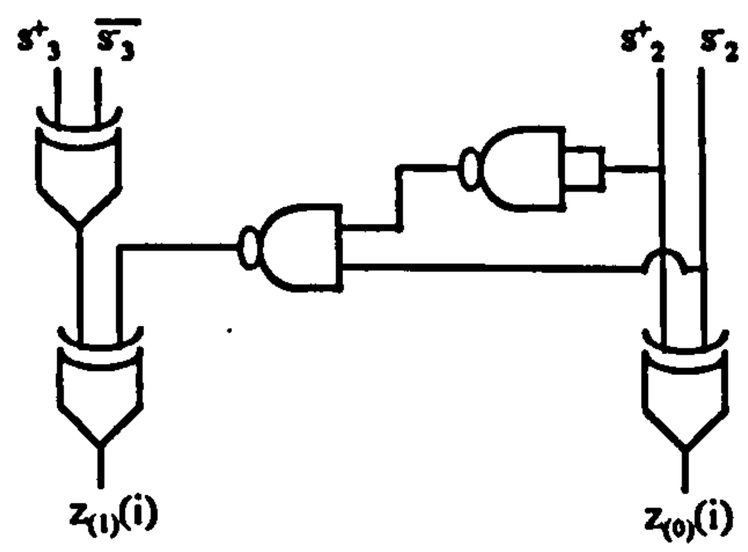


Figure 7.12: Optimized generation of $z(i)$ using MMDAMMM.

with that of the full adder in Figure 4.1 we see that the maximum delay path through these circuits is the same, namely $2 \cdot \Delta_{XOR}$.

Using the MMDAMMM algorithm together with the $z(i)$ generation circuit of Figure 7.12, the $x(i)$ generation circuit of Figure 7.8 and the RSD architecture of Figure 7.9 gives the critical delay path diagram of Figure 7.13. From this diagram we can see that

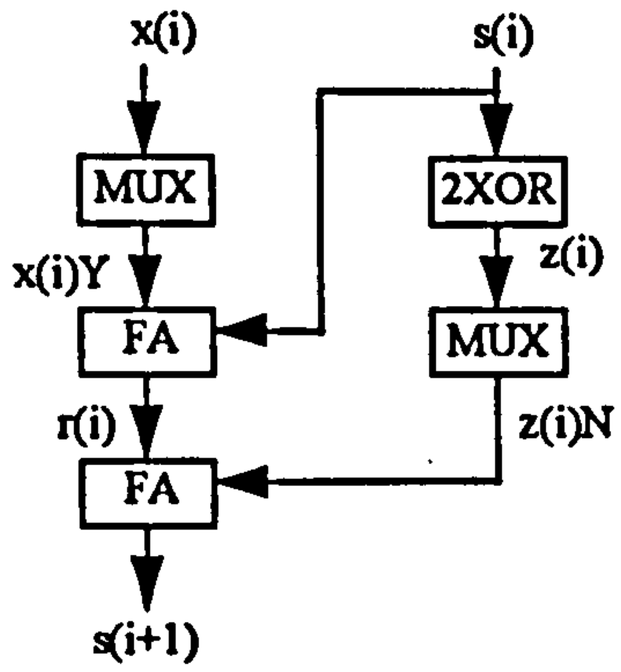


Figure 7.13: Delay path for MMDAMMM.

the generation of $z(i)$ does not add to the iteration time of the multiplier. The adder array can operate at full-speed.

7.2.6 Radix-4 recoded MMDAMMM Performance Summary

Using the condition $2^{b+2} \cdot N < R$ then, for N a k-bit modulus and using $b = 2$, we need $R \geq 2^{k+4}$. Since the MMDAMMM algorithm requires an extra iteration to complete, we can summarize the radix-4 recoded RSD implementation as follows.

$$\begin{aligned} \text{Number of iterations} &= \left\lceil \frac{k}{2} \right\rceil + 3 \\ \text{Iteration time} &= \Delta_{MUX} + 2 \cdot \Delta_{FA} + \Delta_{FF} \\ \text{Number of bitslices} &= k + 8 \\ \text{Bitslice complexity} &= 5 \cdot \Omega_{FF} + 2 \cdot \Omega_{MUX} + 2 \cdot \Omega_{FA} \end{aligned}$$

Comparing the radix-4 recoded design with an unrecoded design would show that the hardware requirements have been much reduced since the former requires only two levels of adders against the latter's four levels. Also, the iteration time is reduced because signal propagation through a MUX and two FAs is less than that through an AND and four FAs.

On comparison with the radix-2 DAMMM CSA design we see that the iteration time has been increased only by the difference $\Delta_{MUX} - \Delta_{AND}$. Compared with the delays of $2 \cdot \Delta_{FA}$ and Δ_{FF} for a typical implementation technology, the difference is small. At the same time the number of iterations required has almost halved. The bitslice complexity has been increased by the difference $2 \cdot (\Omega_{MUX} - \Omega_{AND})$. Compared to the rest of the bitslice circuitry, this increase is small. The number of bitslices is roughly the same. Although an accurate comparison of the two designs would require details of the implementation technology, we can make the qualitative statement that the recoded MMDAMMM RSD design will go almost twice as fast as the DAMMM CSA design with only slightly more hardware.

7.3 Radix-2^b Multiplication

In this section we will examine general radix-2^b recoded multipliers with, for efficient recoding, b an even number.

The general architecture for a b bit MMDAMMM recoded RSD multiplier is shown in Figure 7.14. Each adder sums a 2-bit multiple, $\{-2,-1,0,1\}$, of either Y or M. That

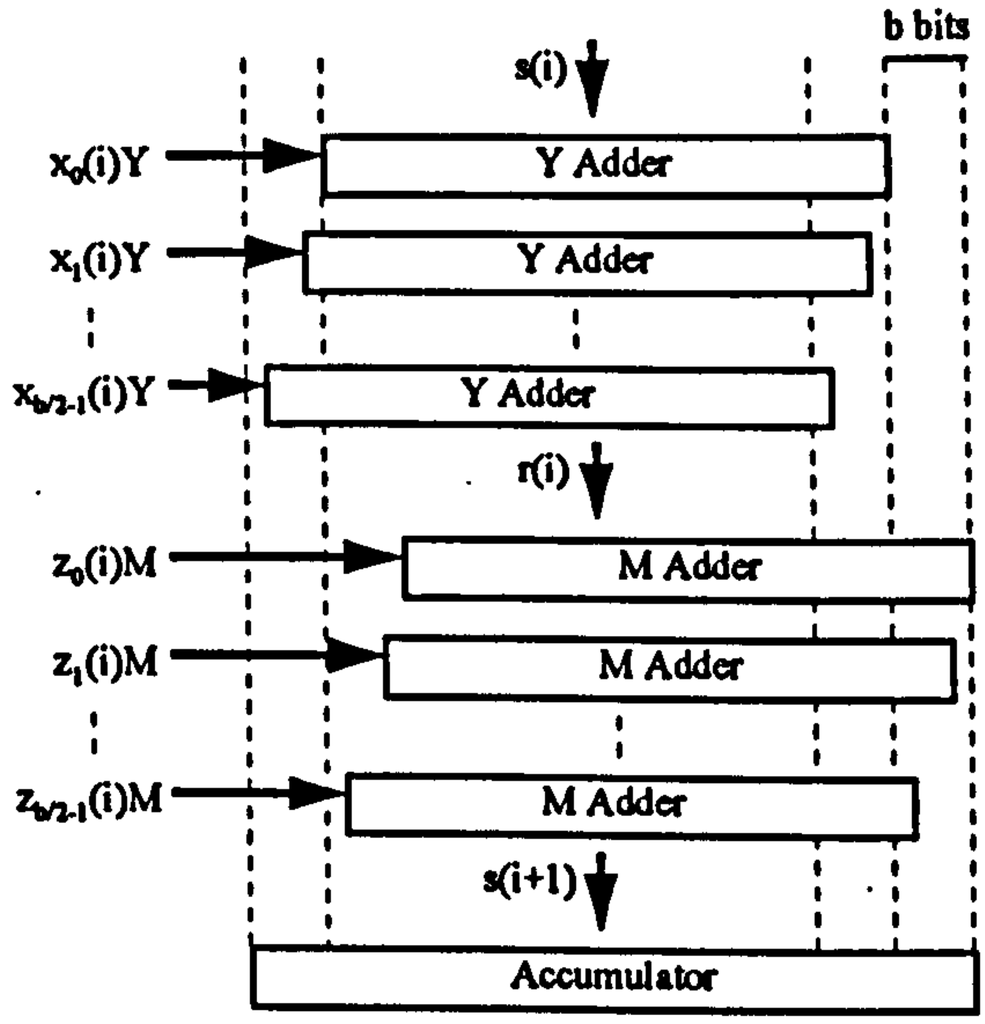


Figure 7.14: MMDAMMM recoded RSD multiplier.

is, each recoded digit $x(i)$ of X and $z(i)$ of Z is composed of 'sub-digits' $x_j(i)$ and $z_j(i)$ respectively such that

$$x(i) = \sum_{j=0}^{b/2-1} 2^{2j} \cdot x_j(i)$$

and

$$z(i) = \sum_{j=0}^{b/2-1} 2^{2j} \cdot z_j(i)$$

where

$$x_j(i), z_j(i) \in \{-2,-1,0,1\}$$

For the adder array to operate at full-speed then

- the j-th sub-digit of $x(i)$ must be ready in time $j \cdot \Delta_{FA}$ after the accumulator's active clock edge, and
- the j-th sub-digit of $z(i)$ must be ready in time $(j + b/2) \cdot \Delta_{FA}$ after the clock edge.

7.3.1 Generating $x(i)$

We can recode a 2's complement vector $X = [X_l, X_{l-1}, \ldots, X_0]$ with $X_i \in [0, 2^b - 1]$ for $i = 0 \ldots l-1$ and $X_l \in [-2^{b-1}, 2^{b-1} - 1]$ to

$$X = \sum_{i=0}^{l} 2^{ib} \cdot x(i)$$

where

$$x(i) = \sum_{j=0}^{b/2-1} 2^{2j} \cdot x_j(i)$$

with $x_j(i) \in \{-2, -1, 0, 1\}$ expressed as

$$x_j(i) = -2 \cdot x_{j,(1)}(i) + x_{j,(0)}(i)$$

with the right-to-left recoding scheme discussed in Section 7.2.3. The circuitry required to perform this recoding is essentially the same as that used in Figure 7.6 for the $b = 2$ case, but expanded out by $b/2$ steps. This is shown in Figure 7.15. A 1-stage pipeline,

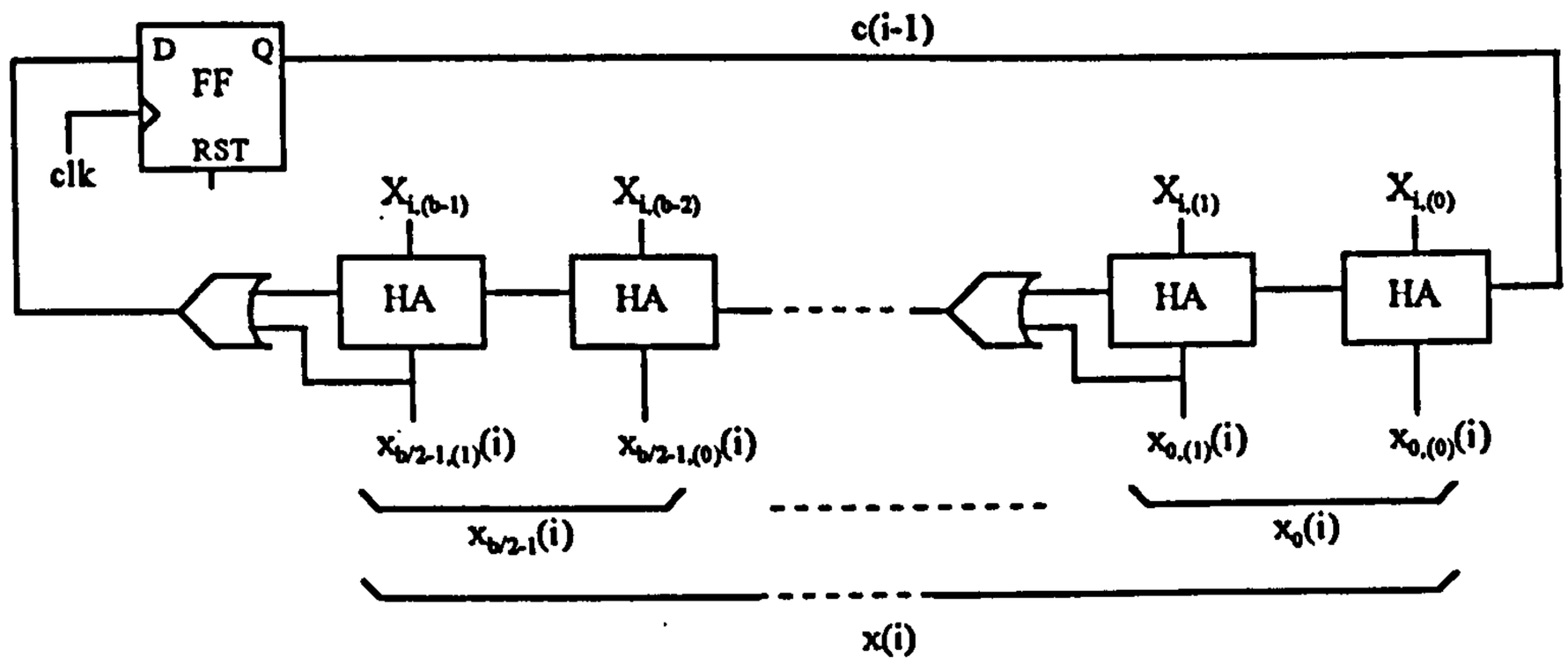


Figure 7.15: General $x(i)$ recoding.

similar to that of Figure 7.8, can also be used in the generation circuitry so that $x(i)$ is available directly after the clock edge.

7.3.2 Generating $z(i)$

The quantity $z(i)$ can be generated in one of two ways, either by lookup-table or by calculation. The former approach offers simplicity but its performance may be technology-dependent. That is, it is difficult to say whether $z(i)$ can be generated fast enough to allow

full-speed adder array operation without examining technology-specific issues. The latter approach however, allows us to derive timing values for $z(i)$ in terms of the primitives that are used to construct the rest of the multiplier circuit. This is the technique that will be used in this section.

In a manner similar to the radix-4 case, $z(i)$ is generated first by calculating

$$Z_i = \langle s(i) \rangle_{2^b}$$

and then by recoding Z_i to $z(i)$ such that

$$z(i) = \sum_{j=0}^{b/2-1} 2^{2j} \cdot z_j(i)$$

satisfies $z(i) \equiv Z_i \pmod{2^b}$. The recoding process is essentially the same as that used to generate $x(i)$ with the exception that carries from one iteration to the next do not have to be saved. Thus the $z(i)$ generation circuitry is as shown in Figure 7.16. Upon

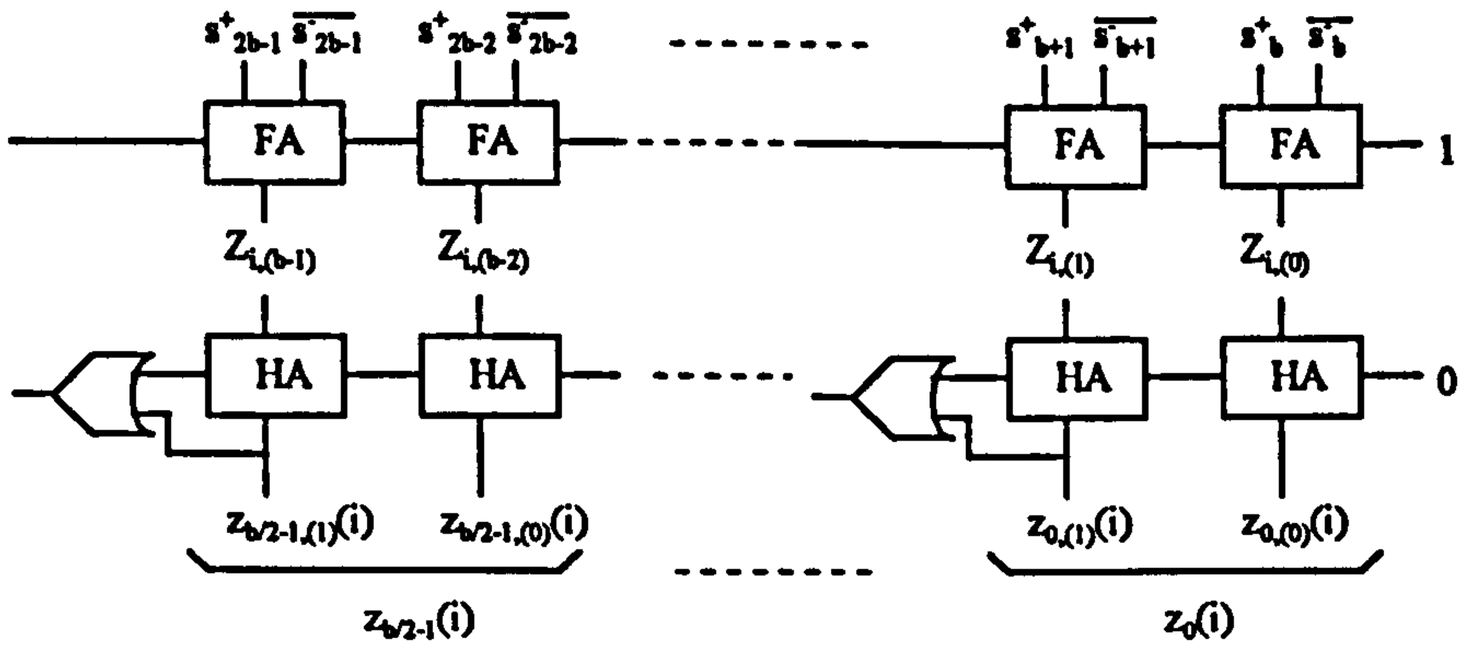


Figure 7.16: Radix-2^b $z(i)$ generation.

examining this circuit we see that, because the carry-in signals to the FA chain and to the HA/OR chain are preset to '1' and '0' respectively, the $z_0(i)$ generation block can be simplified. This simplification takes the form of removing the HA/OR block (because the carry-in of the right-most block is zero – thus no carry-out will be generated and so the $Z_{i,(1)}$ and $Z_{i,(0)}$ signals simply pass straight through to the $z_{0,(1)}(i)$ and $z_{0,(0)}(i)$ signals) and optimizing the FA block such that the resultant circuit is basically the same as that of Figure 7.12 but with the addition of a carry-out signal. The circuit is shown in Figure 7.17.

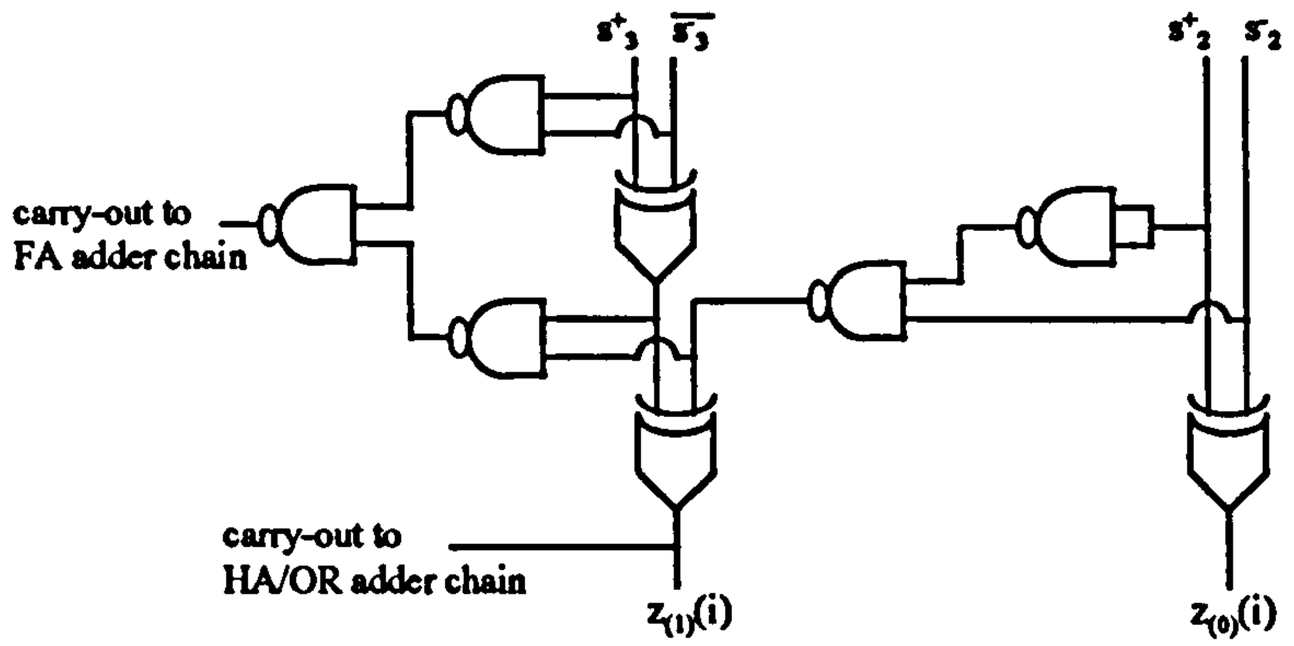


Figure 7.17: Radix-2^b $z_0(i)$ generation.

From this diagram we can see that the maximum propagation delay of signals through this circuit is bounded by $2 \cdot \Delta_{XOR}$.

Looking again at Figure 7.16 we see that the carry-out signals of the FA and HA/OR chains for the $z_{b/2-1}(i)$ generation block are not used. Therefore this block can also be optimized as shown in Figure 7.18. The maximum propagation delay for signals through

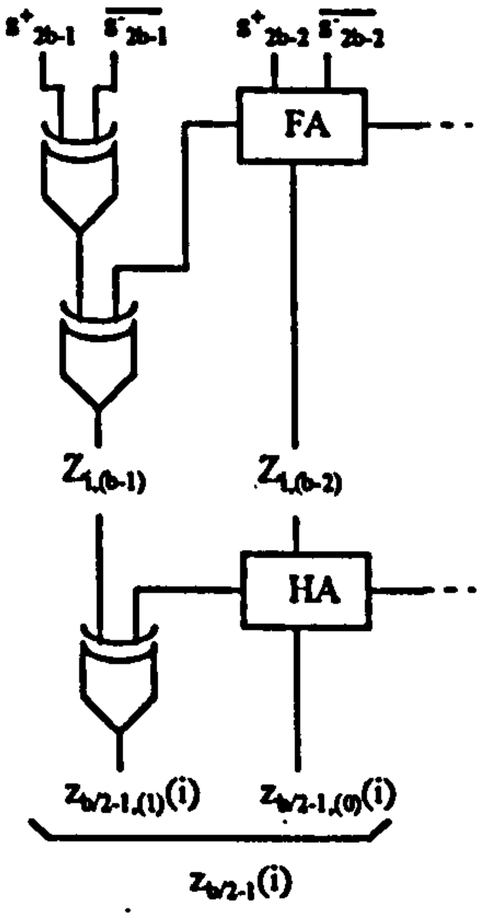


Figure 7.18: Radix-2^b $z_{b/2-1}(i)$ generation.

this circuit is $\Delta_{FA} + 2 \cdot \Delta_{XOR}$.

Assembling the optimized $z(i)$ generation circuitry of Figures 7.16, 7.17 and 7.18 and setting δ_j equal to the generation delay of the j-th sub-digit $z_j(i)$ of $z(i)$ we have

- $\delta_0 = \Delta_{FA}$,
- $\delta_j = \Delta_{FA} + j \cdot 2 \cdot \Delta_{FA} + \Delta_{HA}$ for $j = 1 \ldots b/2 - 2$, and

- $\delta_{b/2-1} = \Delta_{FA} + (b/2-2)\cdot 2\cdot\Delta_{FA} + \Delta_{FA} + 2\cdot\Delta_{XOR}$.

As was stated at the beginning of this section, in order for the adder array to operate at full-speed we need to satisfy $\delta_j \leq (j + b/2)\cdot\Delta_{FA}$. Examining each of the above cases in order, we have

- For $j = 0$ then

$$\delta_j = \Delta_{FA} \leq (b/2)\cdot\Delta_{FA}$$

which is satisfied.

- For $j = 1 \ldots b/2-2$ then

$$\delta_j = \Delta_{FA} + j\cdot 2\cdot\Delta_{FA} + \Delta_{HA} \leq (j + b/2)\cdot\Delta_{FA}$$

which leads to

$$j\cdot\Delta_{FA} \leq (b/2-1)\cdot\Delta_{FA} - \Delta_{HA}$$

which for maximum j gives

$$(b/2-2)\cdot\Delta_{FA} \leq (b/2-1)\cdot\Delta_{FA} - \Delta_{HA}$$

and since $\Delta_{HA} < \Delta_{FA}$ the condition is satisfied.

- For $j = b/2-1$ then

$$\delta_{b/2-1} = \Delta_{FA} + (b/2-2)\cdot 2\cdot\Delta_{FA} + \Delta_{FA} + 2\cdot\Delta_{XOR} \leq (b-1)\cdot\Delta_{FA}$$

which leads to

$$(b-2)\cdot\Delta_{FA} + 2\cdot\Delta_{XOR} \leq (b-1)\cdot\Delta_{FA}$$

and since $\Delta_{FA} = 2\cdot\Delta_{XOR}$ the two sides are equal and the condition is again satisfied.

Thus we have succeeded in generating $z(i)$ fast enough to allow the adder array to operate at full-speed.

7.3.3 MMDAMMM Performance Summary

For N a k-bit modulus we can summarize the radix-2^b recoded RSD implementation as follows. Firstly, N is a k-bit quantity, therefore M is $(k+b)$-bit and R is $(k+b+2)$-bit. Since MMDAMMM requires $l+1$ iterations, then

$$\begin{aligned} \text{Number of iterations} &= \left\lceil \frac{k+2}{b} \right\rceil + 2 \\ \text{Iteration time} &= \Delta_{MUX} + b \cdot \Delta_{FA} + \Delta_{FF} \\ \text{Number of bitslices} &= k + 2b \\ \text{Bitslice complexity} &= 5 \cdot \Omega_{FF} + b \cdot \Omega_{MUX} + b \cdot \Omega_{FA} \end{aligned}$$

Although we have succeeded in generating $z(i)$ fast enough so that, on paper anyway, the multiplier appears to be able to operate at full-speed, a physical realization of the device in VLSI silicon will require buffering of the $z(i)$ signals as they are distributed to all bitslices of the processor. This will likely slow down the operation of the adder array. To get around this problem we will have to generate the $z(i)$ signals and start to distribute them before they are needed. This is the subject of the next section.

7.4 The MMDDAMMM Algorithm

The MMDDAMMM (Modified Modulus Double Delayed Additive Montgomery Modular Multiplication) algorithm is a simple extension of the MMDAMMM algorithm. The difference being that the partial products $X_i \cdot Y$ are further left-shifted by b bits before being added to the accumulated partial product.

Algorithm 19 (MMDDAMMM) *Given a constant $R = 2^{lb}$, odd modulus N such that*

$$2^{b+2} \cdot N < R$$

a constant $N' = \langle -N^{-1} \rangle_R$, and a modified modulus M such that

$$M = N \cdot \langle N' \rangle_{2^b}$$

Then two integers $X, Y \in (-R/2, R/2)$, *with* X *expressed as the* $(l+2)$*-digit vector* $X = [X_{l+1}, X_l, \ldots, X_0]$ *with* $X_i \in [0, 2^b - 1]$ *for* $i = 0 \ldots l$ *and* $X_{l+1} \in [-2^{b-1}, 2^{b-1} - 1]$, *can be Montgomery multiplied by setting*

$$s(0) = 0$$

and letting

$$\begin{aligned} r(i) &= s(i) + 2^{2b} \cdot x(i) \cdot Y \\ s(i+1) &= \frac{r(i) + z(i) \cdot M}{2^b} \end{aligned}$$

with

$$x(i) = \sum_{j=0}^{b/2-1} 2^{2j} \cdot x_j(i)$$

where $x_j(i) \in \{-2, -1, 0, 1\}$ *and*

$$z(i) \equiv s(i) \pmod{2^b}$$

with

$$z(i) = \sum_{j=0}^{b/2-1} 2^{2j} \cdot z_j(i)$$

where $z_j(i) \in \{-2, -1, 0, 1\}$. *will give*

$$s(l+2) \equiv X \cdot Y \cdot R^{-1} \pmod{M}$$

with $s(l+2) \in (-R/2, R/2)$.

Proof: A straightforward extension of MMDAMMM ■.

The implementation of this algorithm using recoding techniques for both $x(i)$ and $z(i)$ would be very similar to the MMDAMMM case of the previous section. The main difference being that one more iteration of the multiplier is required. However, there is a way of altering the $z(i)$ generation circuitry such that $z(i)$ can be generated before it is actually needed. We will examine this as follows.

7.4.1 Radix-4 MMDDAMMM

Looking first at the simple case of $b = 2$, then the arrangement of the two adder levels that make up the multiplier is as shown in Figure 7.19. Using the $z(i)$ generation method

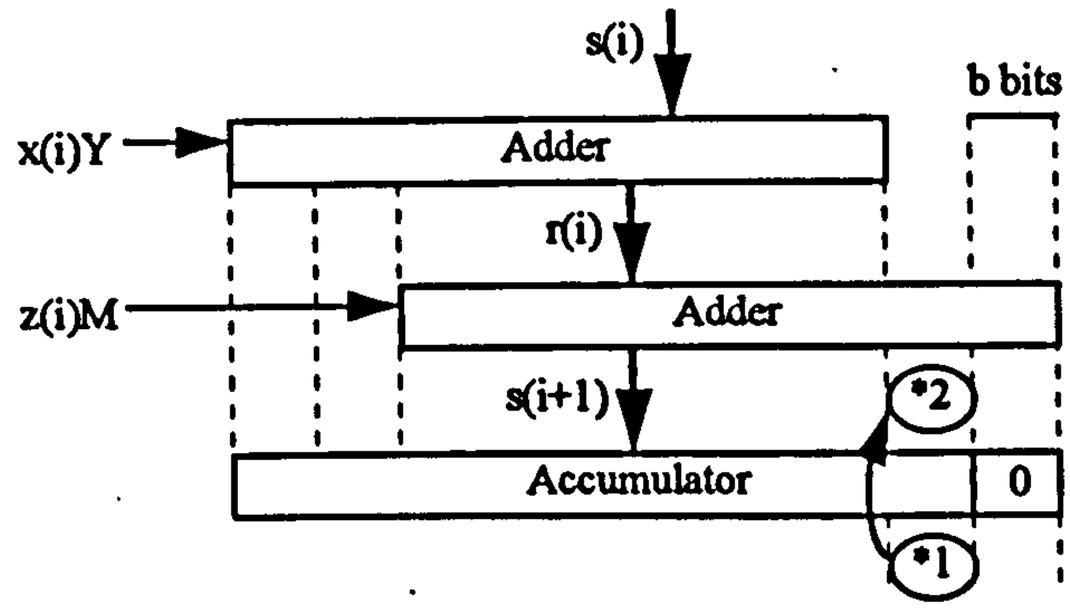


Figure 7.19: Radix-4 MMDDAMMM adder structure.

of the previous section we would 'sample' $\langle s(i) \rangle_{2^b}$ at the position shown as (*1) in the diagram (refer back to Figure 7.2). In this section we will move the 'sample-point' to the position shown as (*2). Thus we will sample the value of $\langle s(i) \rangle_{2^b}$ to use for generating this cycle's $z(i)$ at the end of the previous cycle before the accumulator is clocked. The reason we can do this is because there are no adders located between the (*1) and (*2) points, and so the values at these points will be the same.

Why do we sample at (*2)? To answer this question we first have to make the assumption that $z(i)$ is available directly after the active clock edge that operates the accumulator. (i.e. assume that, contrary to the findings of the previous section, there is no $2 \cdot \Delta_{XOR}$ delay after the clock edge and before $z(i)$ becomes available). Now, because there is no part of the Y adder above the M adder in the sampling column (the b-bit wide column in which (*2) resides – shown by dotted lines in the diagram) and also because we are at the least-significant end of the adder array where the $z(i)$ generation circuit resides (i.e. there is no need to buffer the transmission of the $z(i)$ signals to the $z(i) \cdot M$ multiplexor of the sampling column – they are next door to each other), then the addition of $s(i)$ and $z(i) \cdot M$ by the M adder will be complete in $\Delta_{MUX} + \Delta_{FA}$ time. This means that

there will be a Δ_{FA} timeslot available after the output of the M adder settles and before the accumulator is clocked. Since, as we saw in the previous section, for $b = 2$ it takes $2 \cdot \Delta_{XOR} = \Delta_{FA}$ time to generate $z(i)$, then it could actually be generated here. If we did this then the calculation of $z(i)$ would be complete before the accumulator is clocked, and so $z(i)$ could be made available directly after the clock edge. This actually agrees with our initial assumption. The architecture for such an implementation is shown in Figure 7.20.

In order for this implementation to work, all that is required is that $z(i)$ be available

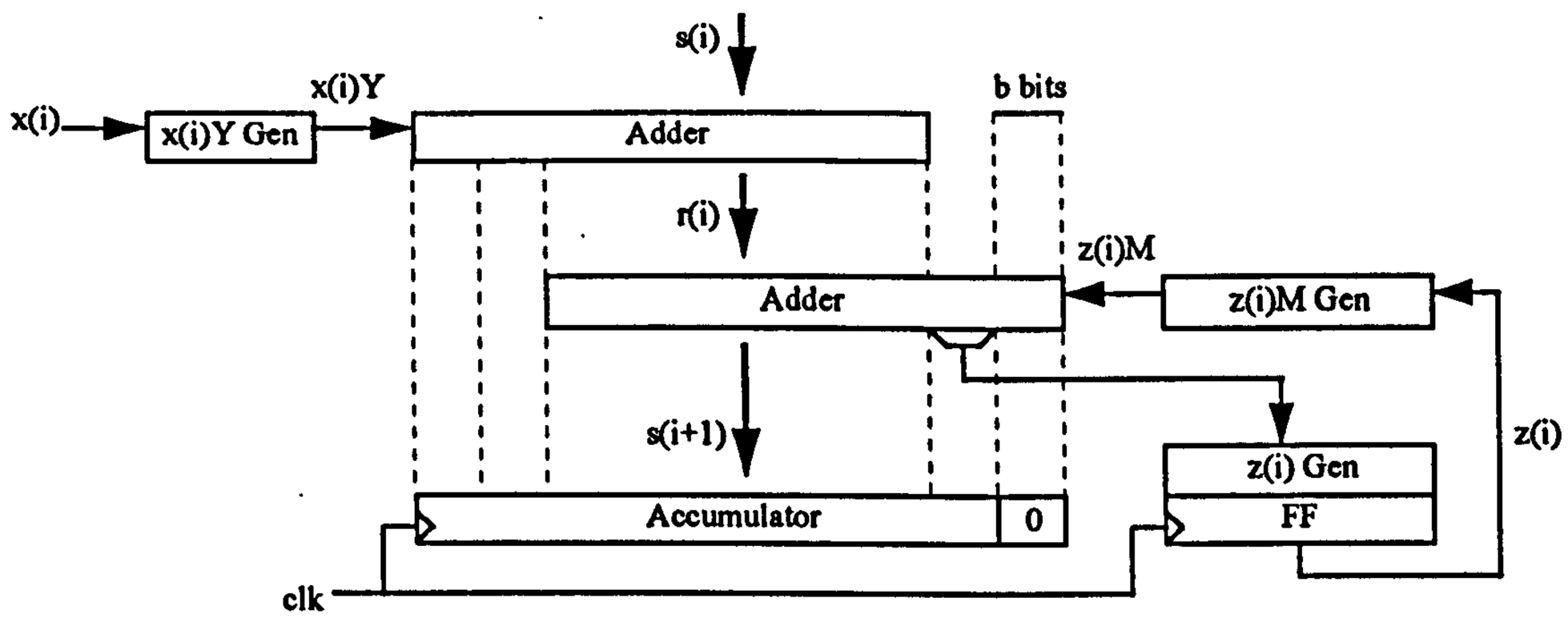


Figure 7.20: Radix-4 MMDDAMMM $z(i)$ generation.

on the active edge of the clock on the first iteration. But this is easy to arrange since, from the algorithm, we can see that $z(0) = 0$ and this can be achieved in the circuitry of Figure 7.20 by resetting the output flip-flops of the $z(i)$ generation block at the same time that the accumulator is reset before a multiplication is started. This fact, coupled with the knowledge that the $z(i)$ generation circuit and the multiplexor and adder circuits for $z(i) \cdot M$ addition in the two least-significant b-bit columns of the adder array can all be located physically close to each other, means that $z(i)$ can always be calculated before the accumulator is clocked, and therefore be available for use directly after the clock edge.

Putting all this together, it means that the $z(i)$ signal is available $2 \cdot \Delta_{XOR}$ time before it is needed by the $z(i) \cdot M$ multiplexors in each bitslice. Therefore this time is available for the buffered transmission of $z(i)$ to the bitslices. Exactly how this will help to keep

the adder array operating at full-speed depends very much on the technology used by the VLSI implementation and also by the buffering strategy used by other signals (e.g. the clock and $x(i)$) in their distribution to the bitslice processor.

Note that, refering again to Figure 7.20, irrespective of how far left-shifted the multiplicand Y may be, it is not possible to move the $s(i)$ sample point further left (either before or after the accumulator) to try and gain more time for $z(i)$ evaluation. The reason is that this would mean that, during the cycles where $z(i)$ is being evaluated, further additions of multiples of M are being performed, and thus the particular $z(i)$ being calculated will not, when used, be the correct value required to zero the lower bits of the accumulator.

7.4.2 Radix-2^b MMDDAMMM

To use this technique for $b > 2$ then the generation of $z(i)$ becomes a 2-stage process. For

$$z(i) = \sum_{j=0}^{b/2-1} 2^{2j} \cdot z_j(i)$$

then, using the $z(i)$ generation method of the previous section, each $z_j(i)$ is available at or before the time it is required. To be more specific, all but the $(b/2-1)$-th sub-multiple are available before they are needed. The last multiple is available exactly at the time it is needed. Splitting the generation of $z(i)$ into two stages so that

- Stage 1: Calculate $z_0(i)$
- Stage 2: Calculate $z_j(i)$ for $j = 1 \ldots b/2-1$

with the first stage completed before the accumulator is clocked and the second completed afterwards, will result in each $z_j(i)$ for $j = 0 \ldots b/2-1$ becoming available at least Δ_{FA} before it is needed. Therefore at least Δ_{FA} time will be available for the transmission of each $z_j(i)$ to the bitslice array. This is shown in Figure 7.21.

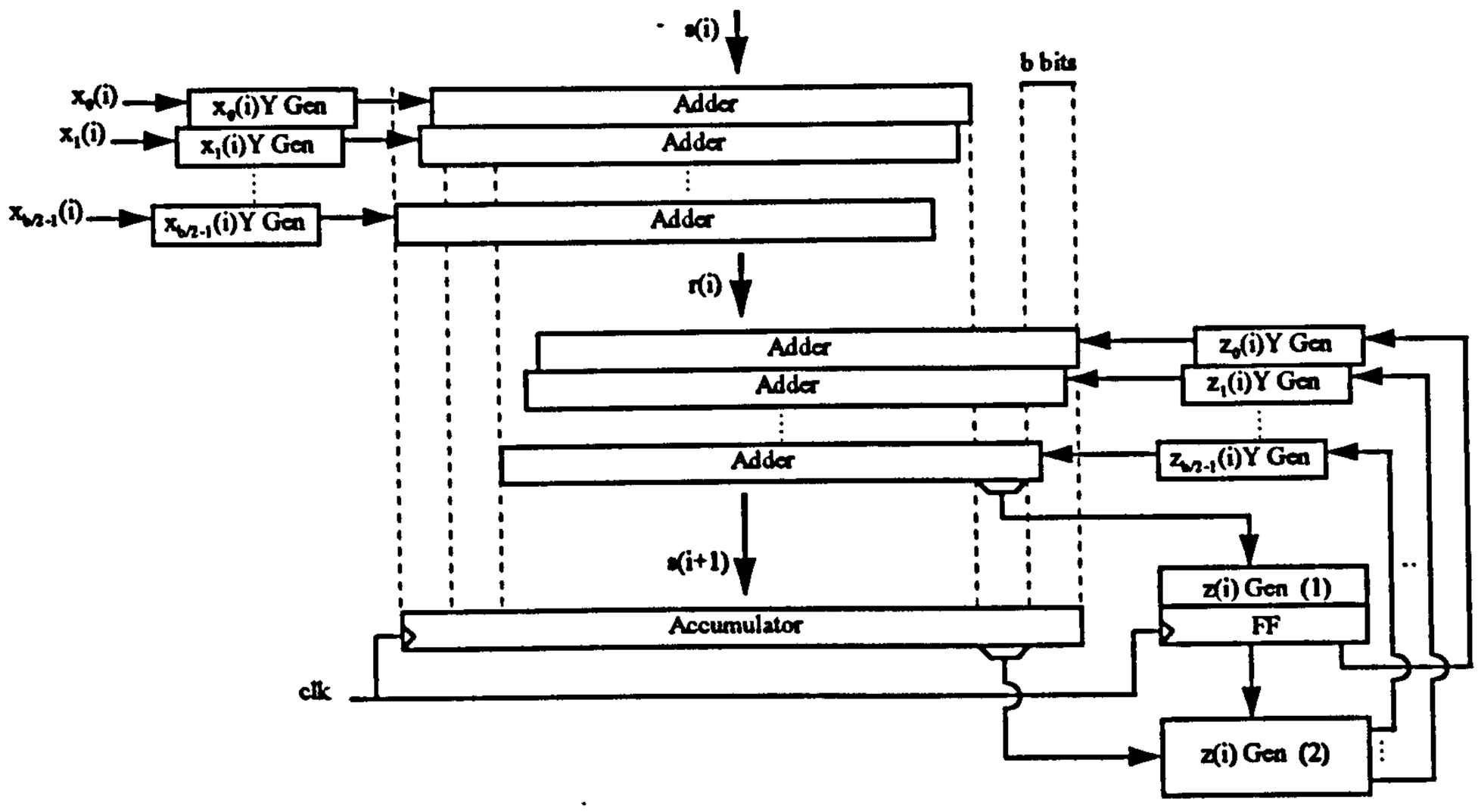


Figure 7.21: Radix-2^b recoded MMDDAMMM multiplier.

7.4.3 Radix-2^b MMDDAMMM Performance Summary

For N a k-bit modulus we need $R \geq 2^{k+b+2}$. Since the MMDDAMMM algorithm requires an extra two iterations to complete, we can summarize the recoded radix-2^b RSD implementation as follows.

$$\begin{aligned} \text{Number of iterations} &= \left\lceil \frac{k+2}{b} \right\rceil + 3 \\ \text{Iteration time} &= \Delta_{MUX} + b \cdot \Delta_{FA} + \Delta_{FF} \\ \text{Number of bitslices} &= k + 3b \\ \text{Bitslice complexity} &= 5 \cdot \Omega_{FF} + b \cdot \Omega_{MUX} + b \cdot \Omega_{FA} \end{aligned}$$

7.5 Summary

In this chapter new, optimised designs for Montgomery multipliers have been presented that allow the multiplier's adder array to operate at full-speed. These designs are for radix-2, radix-4 and general radix-2^b multipliers. The most promising design uses the

MMDDAMMM algorithm together with a recoded RSD architecture.

In the next chapter, technology specific issues will be discussed in determining which of these multipliers will be used to implement the high-speed RSA processor chip called WHiSpER.

Chapter 8

The WHiSpER Chip

The WHiSpER (Wide-word High-Speed Encryption for RSA) chip is an integrated circuit device intended for use within RSA cryptosystems. It is a dedicated long-integer modular exponentiator.

For the device to be particularly useful for implementing RSA cryptosystems over computer and telecommunication networks, then it needs to satisfy two goals,

- security; to accept moduli of at least 500 bits in length, and
- speed; to be able to perform encryption/decryption operations at a rate of not less than 64kbps.

This chapter details the design decisions that had to be made so that the chip was both feasible to manufacture and satisfied its security and speed constraints.

8.1 Technology

The WHiSpER chip is implemented using GEC Plessey Semiconductor's (GPS) CLA70000 gate-array technology [83]. This is a 1 micron twin-well epitaxial CMOS process with two levels of metal interconnect. A range of nine array sizes is available, from the smallest CLA70XXX having 4929 gates, to the largest three,

- CLA76XXX; 110112 gates,
- CLA77XXX; 181260 gates,
- CLA78XXX; 256284 gates.

By a 'gate' is meant the equivalent (in terms of circuit complexity) to a 2-input NAND gate. Using GPS terminology, one NAND gate can be implemented in a single Array Element (AE) which is composed of two p-type/n-type complementary transistor pairs. Thus the largest array in the range contains approximately 1 million transistors.

A range of packaging technologies is available, with the largest arrays available in the following packages:

- Ceramic Leaded Chip Carrier;
- Power Ceramic Leaded Chip Carrier;
- Ceramic Pin Grid Array;
- Power Ceramic Pin Grid Array.

The Power versions of the above package types mount the chip 'cavity-down' with a Cu/W heat-plate. The packages are available with, according to array size and package type, up to 257 pins.

A CLA70000 design is created using design primitives called Cells and Macros. Cells range in complexity from simple 2-input NAND gates to D-type flip-flops and 1-bit full adders. Macros are pre-routed blocks of Cells that make up sub-circuits such as 4-bit counters, 4-bit multipliers etc. A complete design specifies the primitives and their interconnections. Full specifications of GPS Cells and Macros are available in [83].

Design capture was performed on an Apollo workstation using Mentor Graphics version 7 ECAD tools along with GPS CLA70000 technology libraries. Schematic entry and subsequent circuit simulation were performed using a combination of Mentor Graphics

and GPS software. Full details of the GPS Mentor Design Kit can be found in [84], [85] and [86].

A table showing the Δ and Ω characteristics of the four component types used for analysing multiplier performance is shown in Table 8.1. Note that the Δ times are

Component	CLA70000 Cell	Δ (ns)	Ω (AEs)
NAND	NAND2	1	1
MUX	MUX4TO1	3	6
FA	FADD	6	8
FF	MDF	7	6

Table 8.1: CLA70000 Cell characteristics.

maximum delay times that allow for a worst-case manufacturing process scenario.

8.2 The Multiplier

In order to choose an efficient multiplier for the WHiSpER chip, it is first necessary to analyse the time and cost figures given in Chapter 7 for the various different multiplier architectures and algorithms. This analysis is particular to the CLA70000 gate-array technology and is based upon the figures given in Table 8.1.

8.2.1 Efficiency of Recoded Multipliers

In Section 7.2.6 it was stated that the radix-4 recoded multiplier using an RSD architecture with the MMDAMMM algorithm would lead to a design that is almost twice as fast, using only slightly more hardware, than a radix-2 multiplier with a CSA architecture using the DAMMM algorithm. This can be checked as follows.

Let $\mathcal{T}_M$ indicate the time (in nanoseconds) required to perform a multiplication and $\mathcal{R}_M$ the rate of multiplications per second so that

$$\mathcal{R}_M = \frac{10^9}{\mathcal{T}_M}$$

Let $\mathcal{G}_M$ be the total number of gates used in the multiplier design and $\mathcal{E}_M$ a measure of multiplier efficiency such that

$$\mathcal{E}_M = \frac{\mathcal{R}_M}{\mathcal{G}_M}$$

is equal to multiplications per second per gate.

CSA DAMMM $b = 1$

Acknowledging the fact that the AND gate construction for $x_i \cdot Y$ generation of Figure 4.6 can be optimized to use NAND gates, then from Section 7.1.3 we have

$$\begin{aligned} \mathcal{T}_M &= (k+3)(\Delta_{NAND} + 2 \cdot \Delta_{FA} + \Delta_{FF}) \\ \mathcal{G}_M &= (k+4)(2 \cdot \Omega_{NAND} + 2 \cdot \Omega_{FA} + 5 \cdot \Omega_{FF}) \end{aligned}$$

which for $k = 512$ gives

$$\begin{aligned} \mathcal{T}_M &= 10300 \\ \mathcal{R}_M &= 97087 \\ \mathcal{G}_M &= 24768 \\ \mathcal{E}_M &= 3.920 \end{aligned}$$

RSD MMDAMMM $b = 2$

From Section 7.2.6 and assuming k is a multiple of 2, then

$$\begin{aligned} \mathcal{T}_M &= (k/2+3)(\Delta_{MUX} + 2 \cdot \Delta_{FA} + \Delta_{FF}) \\ \mathcal{G}_M &= (k+8)(2 \cdot \Omega_{MUX} + 2 \cdot \Omega_{FA} + 5 \cdot \Omega_{FF}) \end{aligned}$$

which for $k = 512$ gives

$$\begin{aligned} \mathcal{T}_M &= 5698 \\ \mathcal{R}_M &= 175500 \\ \mathcal{G}_M &= 30160 \\ \mathcal{E}_M &= 5.819 \end{aligned}$$

From a simple comparison of the efficiency figures for each of the above implementations we can see that the radix-4 recoded multiplier is better than the unrecoded version. In particular, comparing the multiplication rate and gate count figures we see that the recoded multiplier offers an 81% increase in multiplication speed at the expense of a 22% increase in circuit complexity. This is broadly what was stated in Section 7.2.6.

In constructing performance figures for the general radix-2^b RSD MMDDAMMM recoded multiplier it will be useful also to create figures for a hypothetical radix-2^b CSA MMDDAMMM unrecoded multiplier. The latter figures may then be used as a kind of 'benchmark' for comparison purposes. This may be achieved by noting that, without using recoding techniques, the generation of Z_i will be simpler than the generation of the recoded $z(i)$, and that since twice as many adder levels are used to sum the partial product $X_i \cdot Y$ then the adder array is still able to operate at full-speed.

CSA MMDDAMMM radix-2^b (unrecoded)

Assuming that $k+2$ is a multiple of b we have

$$\mathcal{T}_M = ((k+2)/b+3)(\Delta_{NAND} + 2b \cdot \Delta_{FA} + \Delta_{FF})$$

$$\mathcal{G}_M = (k+3b)(2b \cdot \Omega_{NAND} + 2b \cdot \Omega_{FA} + 5 \cdot \Omega_{FF})$$

which for $k = 512$ and $b = 1, 2, 4, 6, 8$ gives the performance figures of Table 8.2. Note that,

CSA MMDDAMMM (unrecoded) (k=512)				
	T_M (x10^3)	R_M (x10^3)	G_M (x10^3)	E_M
b=1	10.3	97.1	24.7	3.931
b=2	8.3	120.5	34.2	3.523
b=4	7.3	137.0	53.5	2.561
b=6	6.9	144.9	73.1	1.982
b=8	6.8	147.1	93.3	1.577

Table 8.2: CSA MMDDAMMM (unrecoded) performance figures.

as b increases so the efficiency of the multiplier decreases with little speed improvement obtained for high values of b.

RSD MMDDAMMM radix-2^b (recoded)

Assuming that $k+2$ is a multiple of b we have

$$\begin{aligned} \mathcal{T}_M &= ((k+2)/b+3)(\Delta_{MUX} + b \cdot \Delta_{FA} + \Delta_{FF}) \\ \mathcal{G}_M &= (k+3b)(b \cdot \Omega_{MUX} + b \cdot \Omega_{FA} + 5 \cdot \Omega_{FF}) \end{aligned}$$

which for $k = 512$ and $b = 2, 4, 6, 8$ gives the performance figures of Table 8.3.

RSD MMDDAMMM (recoded) (k=512)				
	T_M (x10^3)	R_M (x10^3)	G_M (x10^3)	E_M
b=1	-	-	-	-
b=2	5.7	175.4	30.0	5.847
b=4	4.5	222.2	45.1	4.927
b=6	4.1	243.9	60.4	4.038
b=8	3.9	256.4	76.1	3.369

Table 8.3: RSD MMDDAMMM (recoded) performance figures.

Upon comparison of these two tables we see that the recoded multipliers are both smaller and faster for each implementation $b = 2, 4, 6, 8$. For example, the $b = 2$ recoded multiplier uses 12% less circuitry than the unrecoded $b = 2$ multiplier, but goes 81% faster. Also, because of the reduced circuitry requirements, for a given size of VLSI device it may be possible to implement a recoded multiplier with a higher b value than could be implemented using the unrecoded method. For example, the $b = 8$ recoded multiplier uses only 4% more circuitry than the $b = 6$ unrecoded multiplier (and goes 77% faster).

8.2.2 Multiplier Selection

Assuming a 512-bit modulus and exponent, and also assuming that the exponent is a randomly selected element from the set of all possible 512-bit exponents, then the average number of bits set to a '1' in the exponent is 256. Using either the Right-to-Left or Left-to-Right Montgomery exponentiation algorithms gives

$$\text{Multiplications per exponentiation} \approx 512 + 256 + 3 = 771$$

where the '+3' term accounts for pre- and post-conversion operations.

A 64kbps throughput for RSA requires

$$\text{Exponentiations per second} = \frac{64 \cdot 1000}{512} = 125$$

therefore

$$\text{Multiplications per second} = 125 \cdot 771 = 96375$$

With reference to Table 8.3 we see that using a recoded RSD MMDDAMMM multiplier requires that $b \geq 2$.

The natural choice of which multiplier to use would obviously be the most efficient one that meets our requirements. This implies RSD MMDDAMMM $b = 2$. However, to use this multiplier effectively would require a clock speed of

$$\frac{10^9}{\Delta_{MUX} + 2 \cdot \Delta_{FA} + \Delta_{FF}} = \frac{10^9}{22} \approx 45 \text{ MHz}$$

With the architecture of the multiplier consisting of a very large number of bitslices then the distribution of such a high-speed clock with minimum skew to all bitslices might cause problems. The concern is that the process of mapping the schematic design to silicon (the 'layout' of the VLSI device) will become a critical part of the design, and that considerable manual input will be needed to make the result efficient.

Another cause for concern is that, in analyzing the expected performance of the multiplier, we have used worst-case fabrication tolerances. A common practice in high-performance digital chip manufacture is to measure the fabrication process accuracy for each batch of chips produced. If the accuracy is good then chances are that the chips produced in this batch will be able to be reliably clocked at better than worst-case speeds. In practice several tolerance bands are used, and the chips produced are graded into a number of speed categories. If this technique were used with the $b = 2$ multiplier then the higher clock speeds would be in excess of 50MHz. Generating a 50MHz clock off-chip and then trying to inject it into the chip is, generally speaking, not a good idea. Most manufacturers use a lower frequency off-chip clock combined with an on-chip phase-locked-loop

circuit to create high-frequency on-chip clocks. With CLA70000 gate-array technology such techniques are not available, and so it would be difficult to take advantage of any manufacturing process grading methods.

The next most attractive multiplier is RSD MMDDAMMM $b = 4$. For maximum efficiency this requires a clock-speed of

$$\frac{10^9}{\Delta_{MUX} + 4 \cdot \Delta_{FA} + \Delta_{FF}} = \frac{10^9}{34} \approx 29 \text{ MHz}$$

which is reasonable. On the question of efficiency, it goes 27% faster than the $b = 2$ multiplier for 50% extra circuitry, but the total number of gates is around 45 thousand which makes it acceptable for implementing within the array sizes mentioned in Section 8.1. Therefore this multiplier was chosen as the basis for the WHiSpER chip, with a conservative operational clock speed of 25MHz.

8.2.3 The Carry-Propagate Adder

As was stated in Chapter 7, to make the RSD MMDDAMMM multiplier useful for exponentiations requires that a carry-propagate adder (CPA) be used to assimilate the S^+ and S^- vectors immediately after a multiplication so that the result may be used in further multiplications. However, before we select an appropriate CPA, an alternative architecture that does not require this adder will be investigated.

Full RSD Multiplier

Instead of assimilating the S^+ and S^- vectors after a multiplication, it is possible to construct a multiplier that uses these values directly. Assume Y is the result of a previous multiplication such that

$$Y = Y^+ - Y^-$$

then the partial products of Y can be expressed as

$$x(i) \cdot Y = x(i) \cdot Y^+ - x(i) \cdot Y^-$$

For the RSD MMDDAMMM $b = 4$ multiplier this gives

$$x(i) \cdot Y = 4 \cdot x_1(i) \cdot Y^+ - 4 \cdot x_1(i) \cdot Y^- + x_0(i) \cdot Y^+ - x_0(i) \cdot Y^-$$

and assuming $x(i)$ can be made available directly on the active clock edge then using the adder interconnection optimization technique of Section 4.2.2 leads to the architecture shown in Figure 8.1. Note that the generation of $z(i)$ can still be performed before it is

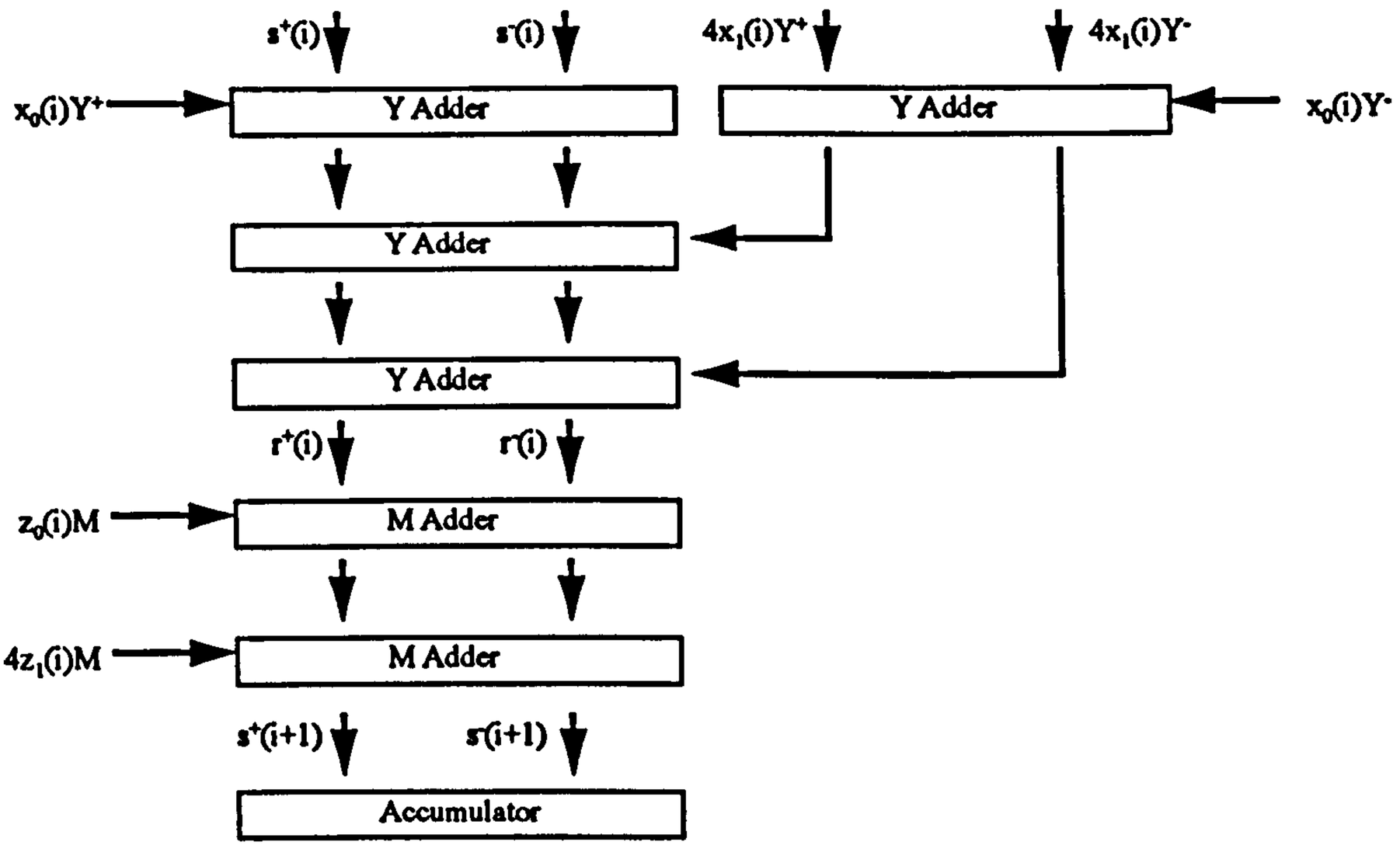


Figure 8.1: Full RSD MMDDAMMM $b = 2$ multiplier.

required since there is a $3 \cdot \Delta_{FA}$ delay in the adder array before the M adders. Performance figures for this multiplier, taking into account the extra adders and multiplexers for the $x(i) \cdot Y$ generation and the extra flip-flops in the X and Y registers, are therefore as follows.

$$\begin{aligned} \mathcal{T}_M &= ((k+2)/b+3) \cdot (\Delta_{MUX} + 5 \cdot \Delta_{FA} + \Delta_{FF}) \\ \mathcal{G}_M &= (k+3b) \cdot (6 \cdot \Omega_{MUX} + 6 \cdot \Omega_{FA} + 7 \cdot \Omega_{FF}) \end{aligned}$$

which gives

$$\begin{aligned} \mathcal{T}_M &= 5260 \\ \mathcal{R}_M &= 190114 \end{aligned}$$

$$\mathcal{G}_M = 66024$$

$$\mathcal{E}_M = 2.879$$

RSD/CPA Multiplier

Assuming that the carry-propagate adder is a simple ripple-adder, then the performance figures for this architecture are as follows.

$$\mathcal{T}_M = ((k+2)/b+3)\cdot(\Delta_{MUX}+4\cdot\Delta_{FA}+\Delta_{FF})+(k+3b)\cdot\Delta_{FA}$$

$$\mathcal{G}_M = (k+3b)\cdot(4\cdot\Omega_{MUX}+4\cdot\Omega_{FA}+5\cdot\Omega_{FF})+(k+3b)\cdot\Omega_{FA}$$

which gives

$$\mathcal{T}_M = 7615$$

$$\mathcal{R}_M = 131320$$

$$\mathcal{G}_M = 49256$$

$$\mathcal{E}_M = 2.666$$

On comparing the Full RSD and RSD/CPA approaches we see that the Full RSD method is more efficient. To be specific, it gives a 45% speed increase for 34% extra circuitry.

Although this seems to imply that the Full RSD approach is better than RSD/CPA, it gives a somewhat false impression. This is because it was assumed that the assimilation adder was a simple ripple-adder. If the type of adder is changed to a more efficient design, then dramatic improvements can be made for little extra cost.

RSD/FCPA Multiplier

An efficient FCPA (Fast Carry-Propagate Adder — for example the carry-completion and carry-select adders of Section 4.1) is the CLA70000 Macro ADT8 8-bit carry-select adder

block. It can add two 8-bit numbers in

$$\Delta_{ADT} \approx 8 \text{ nanoseconds}$$

with a gate count of

$$\Omega_{ADT} = 85 \text{ AEs}$$

This gives it a per-bit addition time of approximately 1ns and a per-bit gate count of approximately 11. Thus it is 600% times faster than the FADD Cell, but only 40% more expensive.

Using the ADT8 Cell as the FCPA gives the following performance figures.

$$\begin{aligned} \mathcal{T}_M &= ((k+2)/b+3)\cdot(\Delta_{MUX}+4\cdot\Delta_{FA}+\Delta_{FF})+(k+3b)\cdot(\Delta_{ADT}/8) \\ \mathcal{G}_M &= (k+3b)\cdot(4\cdot\Omega_{MUX}+4\cdot\Omega_{FA}+5\cdot\Omega_{FF})+(k+3b)\cdot(\Omega_{FA}/8) \end{aligned}$$

which gives

$$\begin{aligned} \mathcal{T}_M &= 4995 \\ \mathcal{R}_M &= 200200 \\ \mathcal{G}_M &= 50632 \\ \mathcal{E}_M &= 3.954 \end{aligned}$$

Thus RSD/FCPA is both faster and smaller than Full RSD. On comparison with RSD/CPA we see that it is 52% faster with only 3% more circuitry. In fact, when compared with the figures of Table 8.3 we see that the carry-propagated addition adds only 11% to the overall multiplication time. Therefore we can conclude that the optimum multipliers do seem to be those of Chapter 7 together with a fast carry-propagate adder.

In summary, the multiplier chosen for the WHiSpER chip is the RSD MMDDAMMM $b = 4$ recoded multiplier with a carry-select adder to perform result assimilation. For a 512-bit modulus, the multiplier is approximately 50 thousand gates in size, and is theoretically capable of performing 200 thousand multiplications per second.

8.3 The Exponentiator

The exponentiator performs calculations of the form $\langle A^E \rangle_M$. Note that this calculation is performed modulo M, that is, using the modified modulus that the multiplier uses. A subsequent reduction of $\langle A^E \rangle_M$ modulo N is assumed to take place after the main exponentiation by a different circuit.

In this section we explore implementations of the alternative exponentiation schemes that were derived in Section 6.2. To simplify the discussion we will use the notation for Montgomery multiplications developed in Section 6.2.2 but using the modulus M. This implies that the results of the Montgomery multiplications are in the range $[0, M-1]$. Whilst this is not strictly true when we use the MMDDAMMMM multiplier of the previous section, it makes little difference when investigating the alternative exponentiation techniques. When it does become important, new notation will be introduced to handle it.

Given that the multiplier has been chosen, the main consideration in designing the exponentiator is that of which algorithm to use. In Section 6.2 we saw that there are two different ways of performing Montgomery exponentiation; either with a post-computation involving a constant derived from R, M and E, or with pre- and post-conversion operations into and out of the M-residue system respectively. The latter technique involving a constant derived from R and M only.

The method chosen is the M-residue technique. This is for the reason that the calculation of the pre-computed constant, $H = \langle R^2 \rangle_M$, depends only on R and M which are both public knowledge. If the other method had been chosen then the computation of its associated constant, $\langle R^{e+1} \rangle_M$, would require knowledge of the exponent, and since the exponent may be secret then performing this calculation may be inconvenient. It would depend upon the particular implementation of the RSA ciphersystem.

The next sections examine the differences between the Right-to-Left and Left-to-Right

M-residue exponentiation schemes.

8.3.1 R-to-L M-residue Exponentiation

The Right-to-Left M-residue exponentiation algorithm was stated in Section 6.2.2. The main loop of this algorithm performs the calculation of $s'(i+1)$ and $t'(i+1)$ as follows.

$$s'(i+1) = \begin{cases} s'(i) & \text{if } e_i = 0 \\ \mathcal{M}_{R,M}(s'(i), t'(i)) & \text{if } e_i = 1 \end{cases}$$

$$t'(i+1) = \mathcal{M}_{R,M}(t'(i), t'(i))$$

From this we can see that the calculations of $s'(i+1)$ and $t'(i+1)$ are independent. That is, each is derived from the $s'(i)$ and $t'(i)$ of the previous iteration. This means that $s'(i+1)$ and $t'(i+1)$ can both be calculated at the same time.

Parallel R-to-L Exponentiation

Using two concurrent parallel multipliers, then the calculations of $s'(i+1)$ and $t'(i+1)$ can proceed as follows.

			Multiplier	Multiplicand
Multiplier 1:	$s'(i+1)$	$\leftarrow$	$t'(i)$	$s'(i)$
Multiplier 2:	$t'(i+1)$	$\leftarrow$	$t'(i)$	$t'(i)$

From this we see that the multiplier operand is the same for both multipliers, therefore the multipliers can be implemented with one X register shared between them.

Defining exponentiation performance figures in the same way as those for multiplication, but using the subscript E, and since the number iterations of the exponentiation algorithm (including pre- and post-conversion operations) is $k+2$, we have

$$\mathcal{T}_E = (k+2) \cdot \mathcal{T}_M$$

$$\mathcal{G}_E = (k+3b) \cdot (2b \cdot \Omega_{MUX} + 2b \cdot \Omega_{FA} + 9 \cdot \Omega_{FF} + 2 \cdot \Omega_{ADT}/8)$$

for $k = 512$ and $b = 4$ this gives

$$\begin{aligned} \mathcal{T}_E &= 2567430 \\ \mathcal{R}_E &= 389.5 \\ \mathcal{G}_E &= 98119 \\ \mathcal{E}_E &= 4.031 \cdot 10^{-3} \end{aligned}$$

Serial R-to-L Exponentiation

Using a single multiplier with the same algorithm yields the following performance figures. Note that, because there are two variables in the algorithm, $s'(i)$ and $t'(i)$, an extra register is needed to hold whichever one of them is not being used.

$$\begin{aligned} \mathcal{T}_E &= (3k/2+3) \cdot \mathcal{T}_M \\ \mathcal{G}_E &= (k+3b) \cdot (b \cdot \Omega_{MUX} + b \cdot \Omega_{FA} + 6 \cdot \Omega_{FF} + \Omega_{ADT}/8) \end{aligned}$$

for $k = 512$ and $b = 4$ this gives

$$\begin{aligned} \mathcal{T}_E &= 3851145 \\ \mathcal{R}_E &= 259.7 \\ \mathcal{G}_E &= 53776 \\ \mathcal{E}_E &= 4.829 \cdot 10^{-3} \end{aligned}$$

Thus the serial multiplier is more efficient than the parallel implementation. The parallel multiplier provides a 50% speedup but at a cost of 82% extra hardware.

8.3.2 L-to-R M-residue Exponentiation

The main loop of the Left-to-Right exponentiation algorithm is

$$s'(i+1) = \begin{cases} \mathcal{M}_{R,M}(s'(i), s'(i)) & \text{if } e_{k-i-1} = 0 \\ \mathcal{M}_{R,M}(\mathcal{M}_{R,M}(s'(i), s'(i)), A') & \text{if } e_{k-i-1} = 1 \end{cases}$$

and so, when two multiplications are required, we see that they have to be performed sequentially. Noting that an extra register is required to hold the pre-converted number A', then the performance figures for this method are

$$\begin{aligned} \mathcal{T}_E &= (3k/2+3)\cdot\mathcal{T}_M \\ \mathcal{G}_E &= (k+3b)\cdot(b\cdot\Omega_{MUX}+b\cdot\Omega_{FA}+6\cdot\Omega_{FF}+\Omega_{ADT}/8) \end{aligned}$$

for $k = 512$ and $b = 4$ this gives

$$\begin{aligned} \mathcal{T}_E &= 3851145 \\ \mathcal{R}_E &= 259.7 \\ \mathcal{G}_E &= 53776 \\ \mathcal{E}_E &= 4.829\cdot 10^{-3} \end{aligned}$$

which are identical to the serial Right-to-Left method.

The main difference between the Right-to-Left and Left-to-Right techniques is that, in the former there are two variables, $s'(i)$ and $t'(i)$, that are continually updated, whereas in the latter there is only one updated variable, $s'(i)$, together with a pre-converted number, A', that is constant for most of the exponentiation. Under the assumption that keeping track of just one variable will simplify the control circuitry and reduce register-to-register communications, then the Left-to-Right exponentiation scheme is the one chosen for the WHiSpER chip.

8.3.3 Optimizing L-to-R Exponentiation

Using the notation that

$$a = [\,b\,]_M$$

means

$$a \equiv b \pmod{M}$$

but with a limited to some range not necessarily equal to $[0, M-1]$, then we can define the operation of the RSD MMDDAMMM multiplier, call it $\lambda_{R,M}(X,Y)$, as

$$\lambda_{R,M}(X,Y) = \left[X \cdot Y \cdot R^{-1}\right]_M$$

where the range is defined as

$$\lambda_{R,M}(X,Y) \in (-R/2, R/2)$$

Pre-Conversion

With $H = \langle R^2 \rangle_M$, then looking at the first three multiplications of the Left-to-Right M-residue exponentiation algorithm we have

$$\begin{aligned} A' &= \lambda_{R,M}(A,H) &&= [A \cdot R]_M \\ s'(0) &= \lambda_{R,M}(1,H) &&= [R]_M \\ s'(1) &= \lambda_{R,M}(\lambda_{R,M}(s'(0),s'(0)),A') &&= [A \cdot R]_M \end{aligned}$$

where the last equality is due to the most-significant-bit of the exponent being a '1'. Examining these three identities we see that it is possible to collapse the calculations into a single Montgomery multiplication. Thus

$$A' = s'(1) = \lambda_{R,M}(A,H)$$

Post-Conversion

From the Left-to-Right M-residue exponentiation post-conversion operation we have

$$s(k) = \lambda_{R,M}(1, s'(k))$$

now since

$$s'(k) \in (-R/2, R/2)$$

then according to the recoded MMDDAMMM algorithm (Section 7.4) we have

$$|s(k)| < \frac{\frac{R}{2} \cdot 1 + 2^{2b} \cdot R \cdot M}{2^{2b} \cdot R} \leq M$$

But we can show that $|s(k)| \neq M$ as follows.

If $s(k) = \pm M$ then obviously

$$s(k) \equiv 0 \pmod{M}$$

which, since $A \in [0, N-1]$ where $M = N \cdot \langle N' \rangle_{2^b} \geq N$, would mean that

$$A = 0$$

Since the M-residue representation of zero is zero, therefore

$$A' = 0$$

and by the last section

$$s'(1) = 0$$

since the calculation of $s'(i+1)$ involves the Montgomery product of $s'(i)$ and A' which would be zero, therefore

$$s'(k) = 0$$

moving back to standard residue representation we have

$$s(k) = 0$$

and so therefore $s(k) \neq \pm M$.

This means that the result range of an exponentiation is

$$s(k) \in [-(M-1), M-1]$$

and so to perform a reduction to the range $[0, M-1]$ involves just a possible addition of M.

The WHiSpER Exponentiation Algorithm

The exponentiation algorithm of the WHiSpER chip is as follows.

Algorithm 20 (Left-to-Right WHiSpER Exponentiation) *Given an integer A, a positive k-bit exponent E, modulus M, constant R and pre-computed constant $H = \langle R^2 \rangle_M$, then calculating $A^E \pmod M$ in the range $[-(M-1), M-1]$ is a 3-stage process.*

1. *Pre-conversion*

$$A' = s'(1) = \lambda_{R,M}(A, H)$$

2. *Processing (for $i = 1 \ldots k-1$)*

$$s'(i+1) = \begin{cases} \lambda_{R,M}(s'(i), s'(i)) & \text{if } e_{k-i-1} = 0 \\ \lambda_{R,M}\left(\lambda_{R,M}(s'(i), s'(i)), A'\right) & \text{if } e_{k-i-1} = 1 \end{cases}$$

3. *Post-conversion*

$$s(k) = \lambda_{R,N}(1, s'(k))$$

8.4 Register Variable Analysis

In this section we will examine the whole process of performing an RSA exponentiation using the multiplier and exponentiator of the previous sections. Looking at the flow of data through these devices will allow us to derive an efficient VLSI design for the WHiSpER chip.

It is assumed that we are given a modulus N, an exponent E, and successive integers

$$A \in [0, N-1]$$

so that we must compute the results

$$D = \langle A^E \rangle_N$$

with as high a throughput as possible.

Assuming that the constant H was pre-calculated once and for all when the RSA key-pair was generated, then the steps involved in performing an exponentiation are

1. Derive the modified modulus for our radix-2^4 MMDDAMMM multiplier

$$M = N \cdot \langle N' \rangle_{2^4}$$

2. Pre-convert the input, A, to M-residue format, A'. Call this value B here so that

$$B = \lambda_{R,M}(A, H)$$

3. Setting $s'(1) = B$ perform the main exponentiation loop for the next-to-most-significant-bit of the exponenent, e_{k-2}, to the least-significant-bit, e_0, so that

$$s'(i+1) = \begin{cases} \lambda_{R,M}(s'(i), s'(i)) & \text{if } e_{k-i-1} = 0 \\ \lambda_{R,M}\left(\lambda_{R,M}(s'(i), s'(i)), B\right) & \text{if } e_{k-i-1} = 1 \end{cases}$$

will give $s'(k) = \left[A^E \cdot R\right]_M$ in the range $(-R/2, R/2)$.

4. Post-convert $s'(k)$ to a value $C = \left[A^E\right]_M$ in the range $[-(M-1), M-1]$ by

$$C = \lambda_{R,M}(1, s'(k))$$

5. Finally, reduce C to the range $[0, N-1]$ so that

$$D = \langle C \rangle_N = \left\langle A^E \right\rangle_N$$

Looking at the way each of the variables is used during the exponentiation process, we find the following.

- N: For the conversion of $N \rightarrow M$ and for reducing $C \rightarrow D$.
- M: Used by the multiplier for all multiplication operations.
- E: Examined one bit at a time between multiplications.
- H: Used by the multiplier but only in the first multiplication.
- A: Used by the multiplier but only in the first multiplication.
- B: Used by the multiplier during any iteration where $e_{k-i-1} = 1$.

- C: For reducing $C \rightarrow D$.

- D: Stored until read.

This implies that each variable has its own 'bandwidth' requirement. i.e. how frequently and in what way it is used. This can be used to advantage by storing the low-bandwidth variables off-chip, in a seperate RAM device, that is mounted alongside but under the direct control of the WHiSpER chip.

Referring back to Chapter 7 we know that the multiplier has registers X, Y and M. The first two are the multiplier and multiplicand registers respectively, whilst the third always holds the modified modulus M. In Section 8.3.2 we noted that the Left-to-Right exponentiator requires an extra register to hold the M-residue representation of its input operand. Therefore call this register B. Since it is assumed that, when the WHiSpER chip is operating, many exponentiations will be performed for the same modulus and exponent, this suggests the following 4 procedures.

- Load: Load the M register with the modified modulus, $N \cdot \langle N' \rangle_{2^t}$, and determine the most-significant bit of the exponent.

- Transfer: Store the contents of the X register (assumed to contain the result of a previous exponentiation) to C. Load the X and Y registers with A and H.

- Exponentiate: Perform pre-conversion (storing M-residue representation of A in the B register), main loop processing and post-conversion.

- Reduce: Reduce C to D.

Note that there is an enforced serialization in the first three processes, but that the reduction process can be overlapped with the exponentiation process. i.e. the exponentiation of the next input operand can be started before the current output has been fully reduced. Also, the load process need only be performed when the RSA key is changed.

The WHiSpER chip control circuitry is based around these four processes, with hardware 'double-buffering' techniques that allow the exponentiator to operate continuously whilst device i/o and the reduction process operate independently and in parallel.

8.5 Architecture

In this section we will explore the general architecture of the WHiSpER chip. As has already been mentioned, the WHiSpER chip employs a recoded RSD MMDDAMMM $b = 4$ multiplier to perform RSA exponentiation. For ease of implementation, the multiplier is constructed so that $R = 2^{512}$. This allows for very simple implementations of control circuitry and counters within the WHiSpER chip. This means that the maximum size of RSA modulus, N, that can be accepted by WHiSpER, according to Section 7.4.3, is 506 bits.

The WHiSpER chip and its associated static RAM (SRAM) chip are shown in Figure 8.2. Communication from the host device to the WHiSpER chip is via WHiSpER's

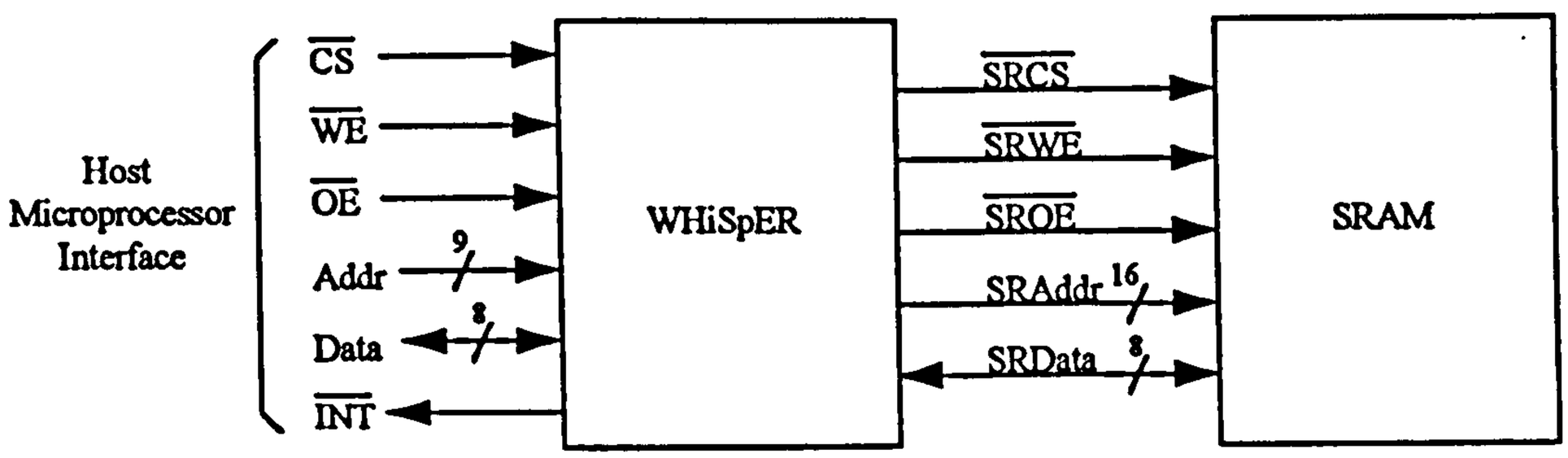


Figure 8.2: The WHiSpER and SRAM devices.

microprocessor interface port. This interface is designed to resemble a 512-byte RAM device, so that, together with the interrupt facility, the WHiSpER chip may be connected to almost any standard microprocessor based system.

8.5.1 The SRAM Device

The SRAM chip of Figure 8.2 is a CMOS static RAM device of up to 64k x 8-bit in size. The SRAM is used to hold RSA keys (modulus and exponent) together with their

pre-computed constants, H. The organization of the SRAM is as follows.

The 64-kbyte space of the SRAM is partitioned into 256 lots of 256-byte spaces. Each 256-byte space is then partitioned into 4 lots of 64-byte spaces. Note that 64 bytes = 512 bits. The four 64-byte spaces are then used to store an RSA *key-pair* (modulus N and exponents E_p and E_s) and pre-computed constant $H = \langle R^2 \rangle_M$ where $R = 2^{512}$ and M is the modified modulus $M = N \cdot \langle N' \rangle_{2^4}$. This is shown in Figure 8.3. Thus the external

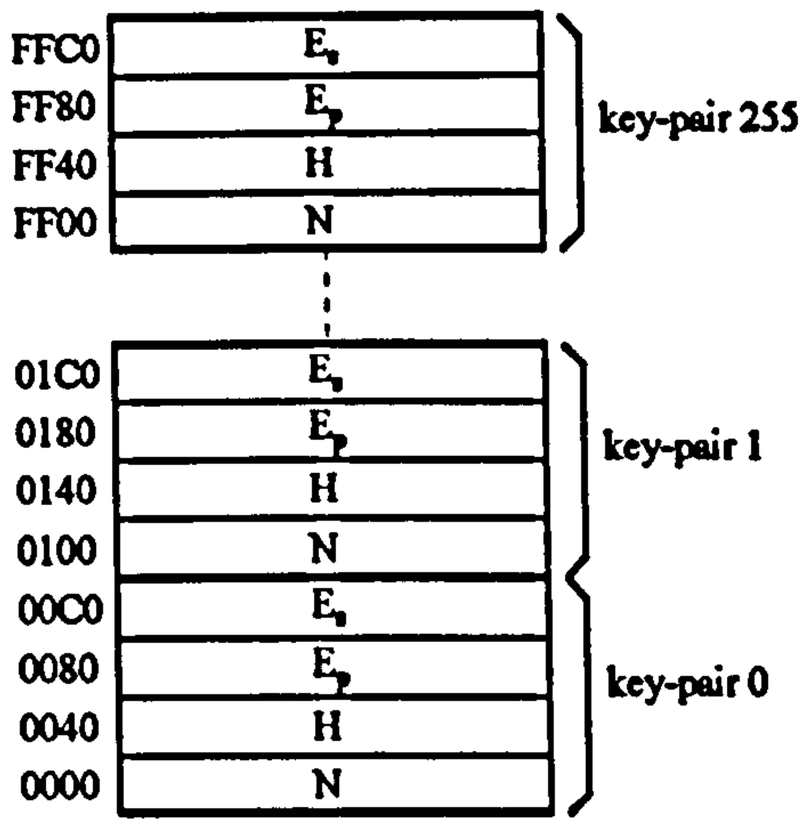


Figure 8.3: SRAM memory map.

SRAM may range in size from 256 bytes to 64 kbytes and may hold from 1 to 256 RSA key-pairs.

Access to the SRAM is available only through the WHiSpER microprocessor interface and limits the host to write-only access. This prevents unauthorized software running on the host system from reading any of the keys.

8.5.2 WHiSpER

A top-level block diagram of the WHiSpER chip is shown in Figure 8.4. The function of each block is as follows.

MME – Montgomery Modular Exponentiator

The MME is an implementation of the exponentiator discussed in Section 8.3. Its architecture is shown in Figure 8.5. It consists of the three main multiplier registers X, Y and

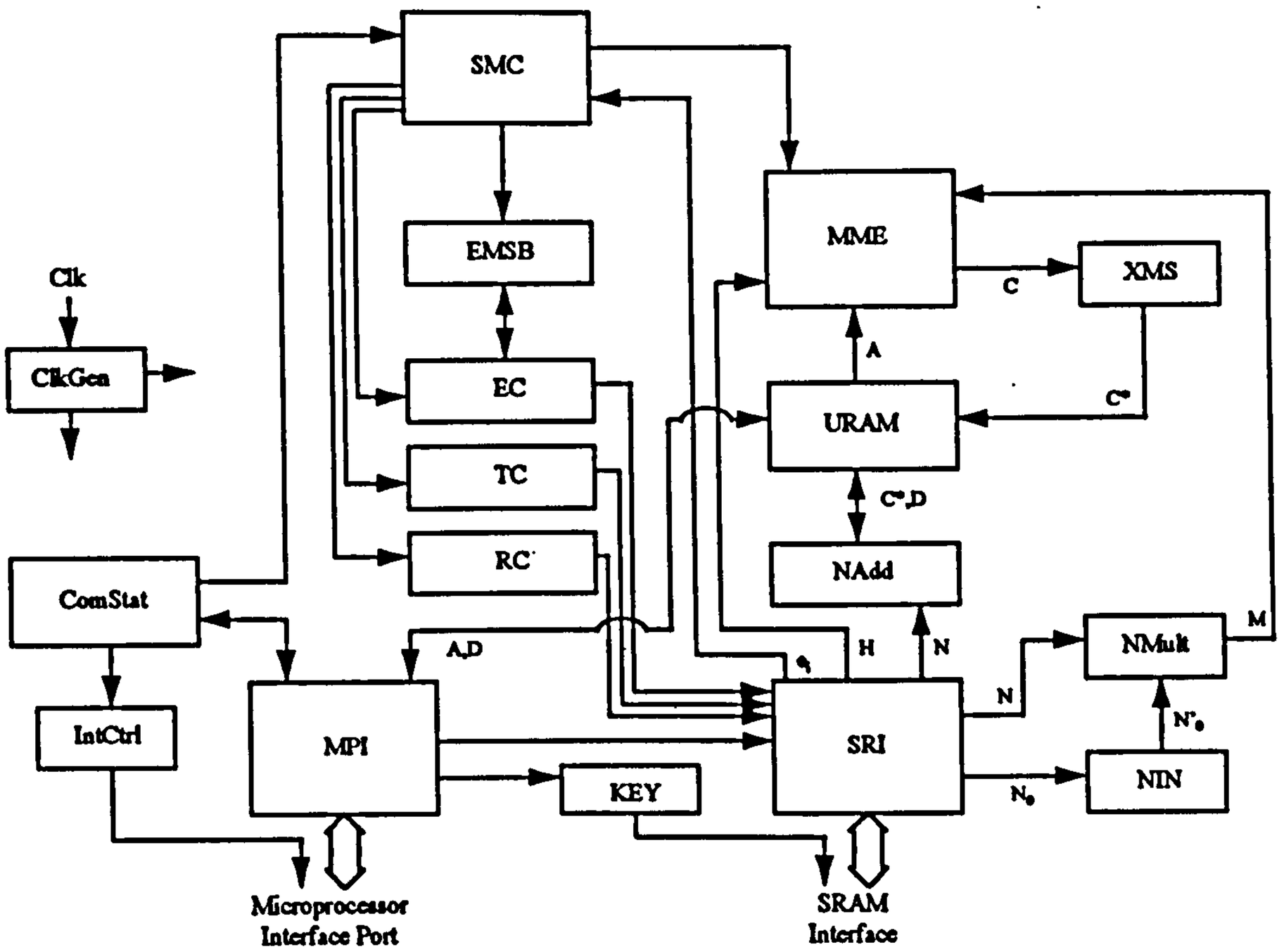


Figure 8.4: The WHiSpER chip.

M, and the M-residue operand register B, together with the $x(i) \cdot Y$ and $z(i) \cdot M$ multiple generation circuits, the RSD adder array, the accumulator and the carry-propagate adder. Parallel 512-bit wide data-paths are shown as broad arrows, whilst serial 4-bit wide data-paths are shown as thin arrows.

To perform an exponentiation, the operation of the MME is as follows (assuming the M register has been pre-loaded with the modified modulus M).

1. The X and Y registers are serially loaded with the A and H values respectively.

2. The first multiplication is performed to yield the value $B = [A \cdot R]_M$. This value corresponds to the calculation that 'collapses' down the first three calculations of the Left-to-Right Montgomery exponentiation as detailed in Section 8.3.3.

3. The B value just generated is parallel loaded into the X and Y registers.

4. The second multiplication is performed. This corresponds to the squaring operation for the exponent bit e_{k-2}. During this multiplication the B register is serially loaded

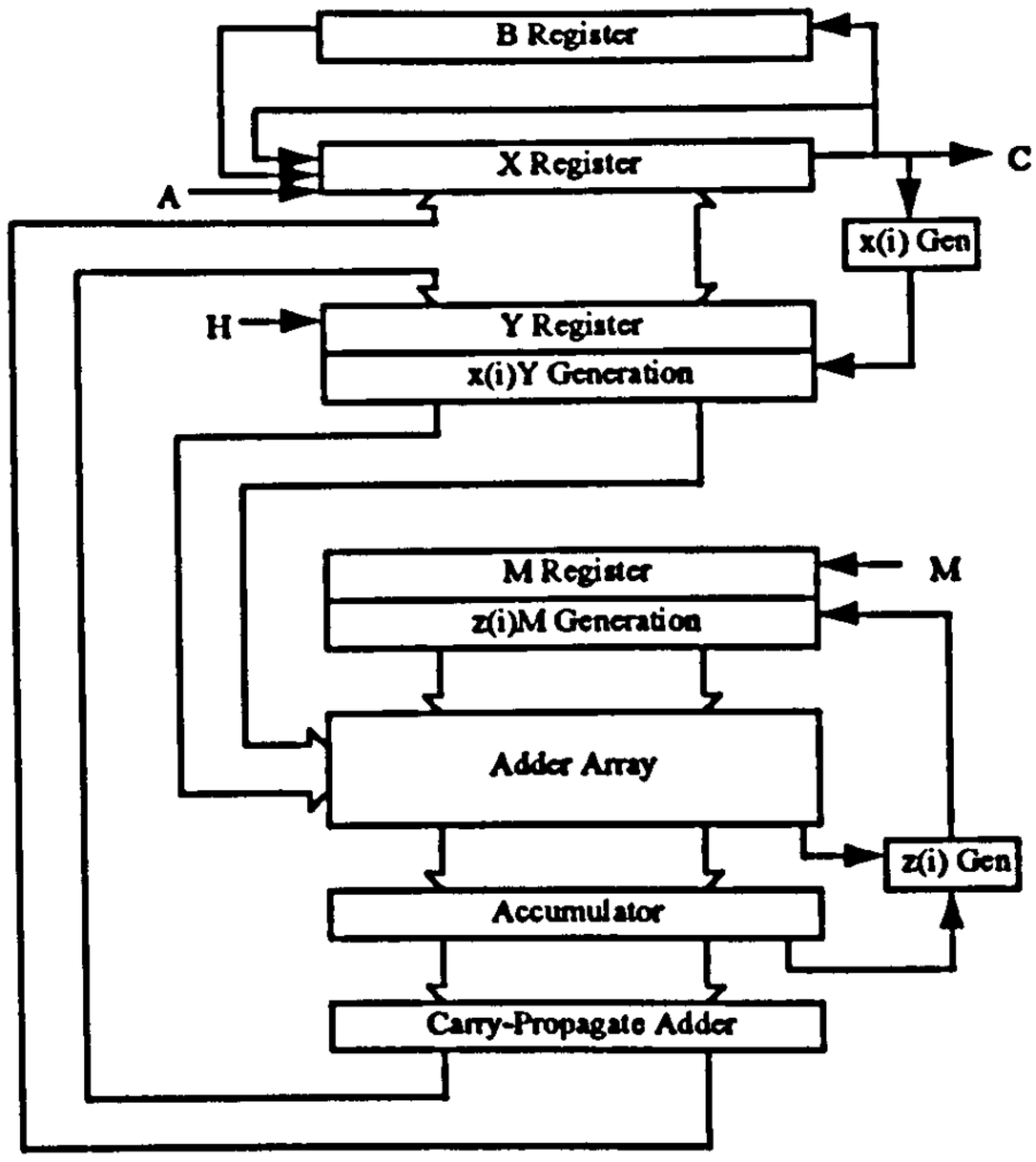


Figure 8.5: MME – Montgomery Modular Exponentiator.

with the B value as it emerges from the X register. The X register also reloads itself with the B value. Thus at the end of the multiplication all three register, B, X and Y, contain the B value.

5. If the exponent bit $e_{k-2} = 1$ then the Y register is parallel loaded with the previous result and a multiplication is performed.

6. The following is now performed for exponent bits e_{k-3} to e_0.

   Parallel load the X and Y registers with the result of the previous multiplication. Perform the multiplication – serially loading the B value from the B register into the X register as the X register is consumed during the multiplication.

   If $e_i = 1$ then parallel load the Y register with the result of the previous multiplication. Perform the multiplication.

7. The Y register is parallel loaded with the result of the previous multiplication. A multiplication is performed with the $x(i)$ generation circuitry switched to generate $X = 1$.

8. The X register is parallel loaded with the result of the previous multiplication.

9. The X register serially unloads the value $C = \left[A^E\right]_M$ as it is loaded with a new A value. The Y register is serially loaded with the H value.

SMC – State Machine Controller

The SMC provides all of the control signals necessary for the operation of WHiSpER. A block diagram of the SMC is shown in Figure 8.6. The SMC consists of the four separate

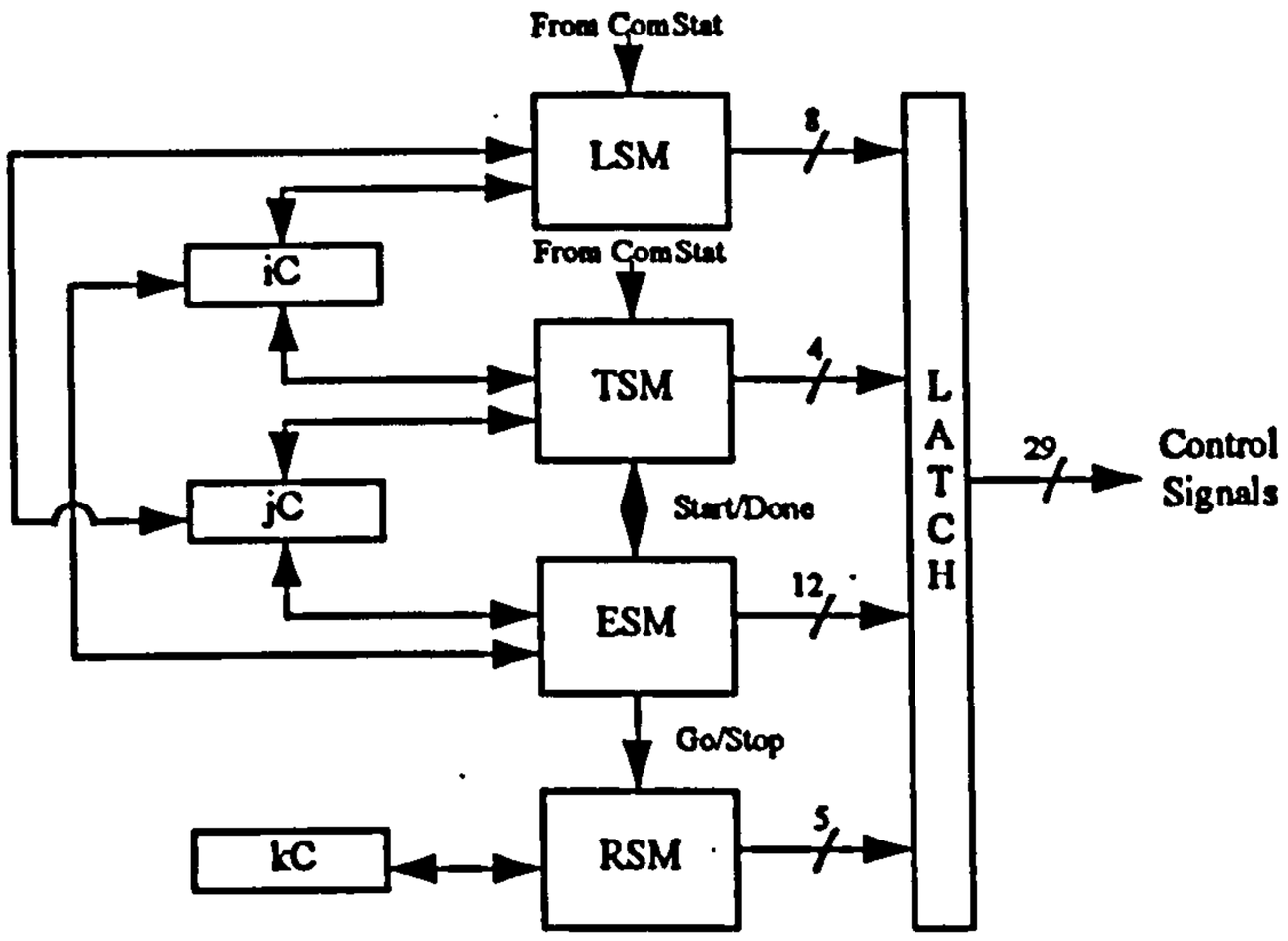


Figure 8.6: SMC – State Machine Controller.

semi-autonomous state machines, LSM, TSM, ESM and RSM, together with three general purpose counters, iC, jC and kC, and an output latch. The purpose of each state machine is as follows (they are closely related to the four processes of Section 8.4).

- LSM – Load State Machine. The LSM is in charge of loading the multiplier M register with the modified modulus $M = N \cdot \langle N' \rangle_{2^4}$. It also finds the most-significant-bit of the current exponent.

- TSM – Transfer State Machine. The TSM handles the serial transfers into and out of the multiplier's X and Y registers before each exponentiation starts. Once the transfer is complete, it will start the ESM and RSM state machines.

- ESM – Exponentiation State Machine. The ESM handles the operation of the MME. It will signal the TSM when an exponentiation is complete. The ESM can temporarily halt the operation of the RSM via stop/go signals.

- RSM – Reduction State Machine. The RSM performs the final reduction of a result from the MME to the range $[0, N-1]$. It informs the TSM when it has completed.

The reason that the ESM has stop/go control over the RSM is because both the exponentiation and reduction processes need access to the SRAM as they are working. The ESM needs to access the exponent bits whilst the RSM needs to access the modulus N. The ESM has control over the RSM so that the exponentiation is not delayed. The reduction process has plenty of time in which to complete, so that a few halts will not effect overall performance.

XMS – X, M Subtractor

When the value C is serially transferred out of the MME then if $C \geq 0$ this circuit will serially subtract M from it. Thus a 2's complement value C^* is produced in the range $[-(M-1), -1]$. This is done so that the final reduction process can simply keep adding N to C^* until a positive value is produced. This simplifies the reduction circuitry so that completion is detected from the sign bit of C^*, that is, no long-integer comparison is required.

URAM – U RAM

The URAM is used to hold incoming A values and also to hold the C^* value while it is being reduced to D in the range $[0, N-1]$. The D value is held here until the host microprocessor reads it.

NAdd – N Adder

The NAdd is an 8-bit multi-precision adder that is used to reduce C^* to the range $[0, N-1]$.

It does this by adding N until C^* goes positive.

SRI – SRAM Interface

The SRI generates the necessary control signals to operate the external SRAM device.

NIN – Negative Inverse of N

The NIN is a 4-bit register that holds the value $N'_0 = \langle N' \rangle_{2^4}$.

NMult – N Multiplier

The NMult is a 4-bit multi-precision multiplier used to calculate $M = N \cdot \langle N' \rangle_{2^4}$.

MPI – MicroProcessor Interface

The MPI performs microprocessor control signal translation and address decoding.

IntCtrl – Interrupt Control

Handles enabling/disabling and setting/resetting the external $\overline{\text{INT}}$ signal.

ComStat – Command and Status

Interprets the host microprocessor commands and provides a device status register.

KEY – Key-pair Selection

An 8-bit register that defines the current SRAM key-pair being used.

EMSB – Exponent Most-Significant-Bit

A 9-bit register holding the bit-position of the current exponent's most-significant-bit.

EC – Exponent Counter

A 9-bit counter used for keeping track of the current exponent bit under examination during an exponentiation.

TC – Transfer Counter

A 7-bit counter used when performing transfers between URAM/SRAM and multiplier registers.

RC – Reduction Counter

A 6-bit counter used when performing the multi-precision reduction of $C^* \rightarrow D$.

ClkGen – Clock Generation

Generates all necessary clock signals for the device.

8.6 Operation

As stated in the previous section, the WHiSpER chip appears as a 512-byte area of memory to the host microprocessor system. A map of this memory is shown in Figure 8.7. As can be seen from this diagram, the memory is partitioned into four main areas.

- SRAM – A 256-byte window on the SRAM memory.
- URAM – Access to the 64-byte URAM.
- Registers – Access to the 8-bit read/write KEY register, and the 8-bit read-only STATUS register.
- Commands – The WHiSpER operational commands. A write operation to a command's address will cause that command to execute.

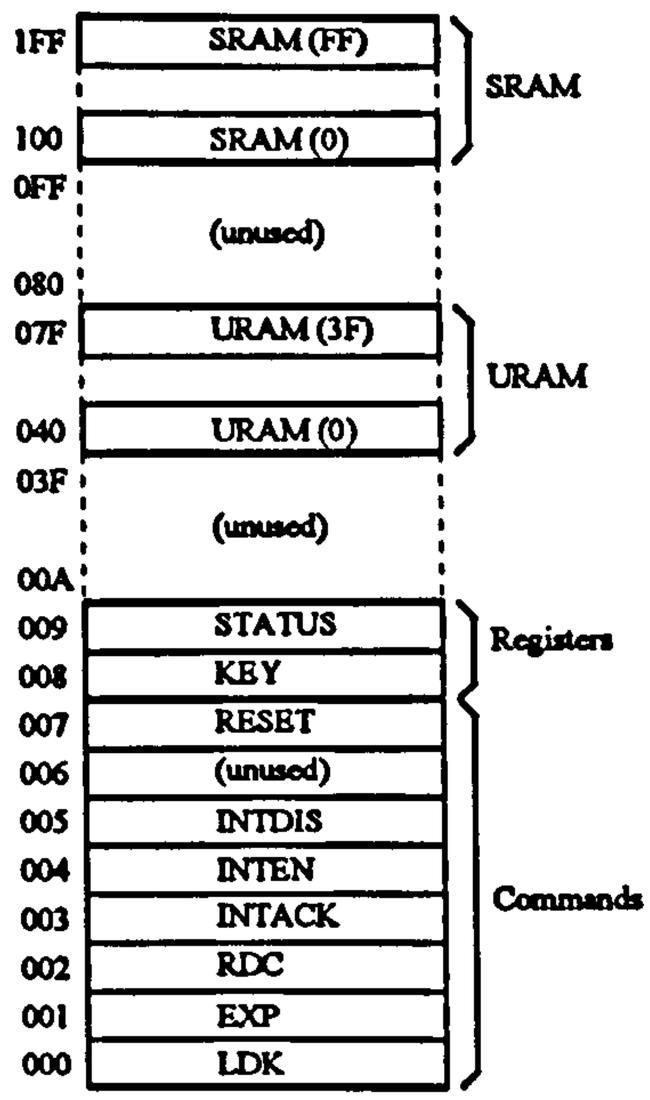


Figure 8.7: WHiSpER memory map.

8.6.1 RAM

The SRAM Window: Address 0x100 - 0x1FF

As was stated in Section 8.5.1 the SRAM is partitioned into 256 lots of 256-byte blocks. Each block holds an RSA key-pair and its associated constant. The SRAM window provides access to one of these blocks. The particular block in question is determined by the KEY register (see below).

Stored within this block are N, H, E_p and E_s at microprocessor interface port addresses 0x100, 0x140, 0x180 and 0x1C0 respectively. Each number is stored 'little-endian', that is, starting with the least-significant-byte first. For example, if

$$N = [N_{l-1}, N_{l-2}, \ldots, N_0]$$

where each N_i is the i-th byte of N such that

$$N = \sum_{i=0}^{l-1} 2^{8i} \cdot N_i$$

then N_0 will be stored at 0x100, N_1 will be stored at 0x101 and so on.

The URAM: Address 0x040 - 0x07F

The URAM is a 64-byte block of RAM for writing and reading exponentiation operands and results respectively. As in the SRAM case, numbers are written 'little-endian'.

8.6.2 Registers

The KEY Register: Address 0x008

The KEY register is an 8-bit read/write register that determines which 256-byte block of SRAM, holding $\{N, H, E_p, E_s\}$, will be used for subsequent WHiSpER operations. It also determines the 256-byte block that will be addressed by the microprocessor port's SRAM window. Simply stated, the contents of this register provide the high 8 bits of the SRAM address lines.

The STATUS Register: Address 0x009

The STATUS register is an 8-bit read-only register that allows the host microprocessor to determine the current state of the WHiSpER chip. It has three valid status bits.

- Bit 7 – UBUSY. When this bit is active-high it signals that the URAM is unavailable for reading/writing by the host microprocessor.
- Bit 6 – SRBUSY. When this bit is active-high it signals that the SRAM is unavailable for writing to by the host microprocessor.
- Bit 0 – ES. Along with the KEY register this bit signals which exponent will be used when performing an exponentiation. ES=0 means E_p will be used. ES=1 means E_s will be used.

The UBUSY and SRBUSY bits will only go active as a direct result of the host microprocessor issuing a WHiSpER command. Furthermore, they go active *during* the command write operation. This means that there is no danger of the host microprocessor's being

blocked while part-way through reading/writing the URAM or SRAM areas. All it need do is make sure that the appropriate bit is inactive before proceeding.

The ES bit is both set and reset with the LDK command detailed below.

8.6.3 Commands

The details of each command are presented next. The action of each command is summarized, followed by a block-level description of the command's operation. Refer to Figure 8.4 for the WHiSpER block diagram.

The LDK Command: Address 0x000

The LDK command is executed by performing a microprocessor write operation to this address. It loads the multiplier's M register with the modified modulus and finds the most-significant-bit of the selected exponent. Selection of either E_p or E_s is effected by the microprocessor writing either a '0' or a '1' respectively.

On issuing this command the SRBUSY status bit will go active. Following this, the ComStat circuit will issue a signal to the SMC that starts up the LSM state machine. LSM will now load NIN with N_0' and proceed to transfer N, using the TC, from the SRAM via the NMult circuit so that the modified modulus M is loaded into the multiplier's M register. After this, the LSM will preset EC to 511 and then, counting down, will examine each bit of the selected exponent in turn until the first '1' is found. When this happens, the EMSB register will be loaded with the EC count. The command will terminate by clearing the SRBUSY status bit and activating the external $\overline{\text{INT}}$ signal, if enabled.

The EXP Command: Address 0x001

The EXP command is executed by performing a microprocessor write operation to this address. It takes the current A value that the microprocessor has previously placed into the URAM and performs an exponentiation on it. At the same time that it reads the A

value, it will place a D value from the previous exponentiation into the URAM so that the microprocessor can read it.

On issuing this command the UBUSY and SRBUSY status bits will both go active. ComStat then issues a signal to the SMC that starts up the TSM state machine. The TSM will now transfer the next exponentiation operand A from the URAM to the multiplier's X register, and the constant, H, from the SRAM to the multiplier's Y register. At the same time, the previous exponentiation result, C, will be read from the X register and transferred via the XMS to produce C^* which will be written into the URAM. The TSM will now start up the ESM and RSM state machines.

The ESM controls the MME to perform an exponentiation as discussed in Section 8.5.2. The RSM controls the URAM, NAdd and SRI circuits to perform the reduction of C^* to D. It does this by using multi-precision (8-bit) additions to repeatedly add N to C^*, and stops when the result of an addition is positive.

Once the RSD signals completion, the UBUSY status bit goes inactive and $\overline{\text{INT}}$ is asserted. The URAM is now available to the host microprocessor, which may read the result, D, of the just-reduced previous exponentiation, and then write the next exponentiation operand, A, into this memory. If the microprocessor does this, then it should issue another EXP or RDC command immediately, without waiting for the current EXP operation to complete. The command will be latched by ComStat (with the UBUSY status bit activated once more) and acted upon as soon as the current EXP operation has completed.

When the ESM signals completion, the TSM resumes control once more. It checks to make sure that the RSM has completed, and if so then it will wait for another EXP or RDC command. If the command is already waiting (latched by ComStat) then it will be executed immediately. Otherwise it will wait indefinitely (with SRBUSY active and UBUSY inactive) for one of these commands to arrive.

The RDC Command: Address 0x002

The RDC command is executed by performing a microprocessor write operation to this address. It places a D value from the previous exponentiation into the URAM so that the microprocessor can read it. The RDC command should be used to terminate a string of EXP commands.

This command should only be issued after a previous EXP command. From the above description of the EXP command we know that the TSM will be expecting this command to arrive, and that the STATUS register will have SRBUSY active and UBUSY inactive.

On recieving this command the UBUSY status bit will go active. The result of the previous exponentiation, C, will be transferred from the MME via the XMS to produce C^* which will be written to the URAM. The RSM will now be invoked to reduce C^* to D. Upon completion, both the UBUSY and SRBUSY status bits will go inactive and the $\overline{\text{INT}}$ signal will be asserted. The TSM will now return to its idle state and so the entire chip will be at rest.

The INTACK Command: Address 0x003

The INTACK command is executed by performing a microprocessor write operation to this address. If the external $\overline{\text{INT}}$ signal is active then this command will deactivate it.

The INTEN Command: Address 0x004

The INTEN command is executed by performing a microprocessor write operation to this address. It enables the operation of the external $\overline{\text{INT}}$ signal. After a device reset, the $\overline{\text{INT}}$ signal is disabled, and this command must be used to enable it.

The INTDIS Command: Address 0x005

The INTDIS command is executed by performing a microprocessor write operation to this address. It disables the operation of the $\overline{\text{INT}}$ signal. If the $\overline{\text{INT}}$ signal is active then it

will be deactivated.

The RESET Command: Address 0x007

The RESET command is executed by performing a microprocessor write operation to this address. It performs a device reset. After a reset, the chip will be in an idle state awaiting further commands. The KEY and STATUS registers will be cleared. The $\overline{\text{INT}}$ signal will be disabled. The SRAM device should still contain its pre-reset information. The URAM should be assumed to contain garbage. All other device registers should be assumed to contain garbage.

8.6.4 Operation Examples

Single Exponentiation

The following example shows how to perform the single exponentiation $D = \left\langle A^E \right\rangle_N$ using STATUS register polling (non-interrupt operation).

1. Power-up reset. Interrupts disabled, KEY = 0, STATUS = 0.

2. Write N, H and E to the SRAM window.

$$N \rightarrow \text{0x100}$$

$$H \rightarrow \text{0x140}$$

$$E \rightarrow \text{0x180}$$

3. Write LDK(0) command.

4. Poll STATUS until SRBUSY goes inactive low.

5. Write A to URAM.

6. Write EXP command.

7. Poll STATUS until UBUSY goes inactive low.

8. Write RDC command.

9. Poll STATUS until UBUSY goes inactive low.

10. Read *D* from URAM.

Multiple Exponentiation

The diagram of Figure 8.8 is a state-transition diagram showing how to perform the exponentiation of a sequence of *A*s to yield their respective sequence of *D*s. The sequence of *A*s is denoted as

$$A[1], A[2], \ldots, A[n]$$

The process is interrupt driven, and it is assumed that the SRAM has been previously filled with keys.

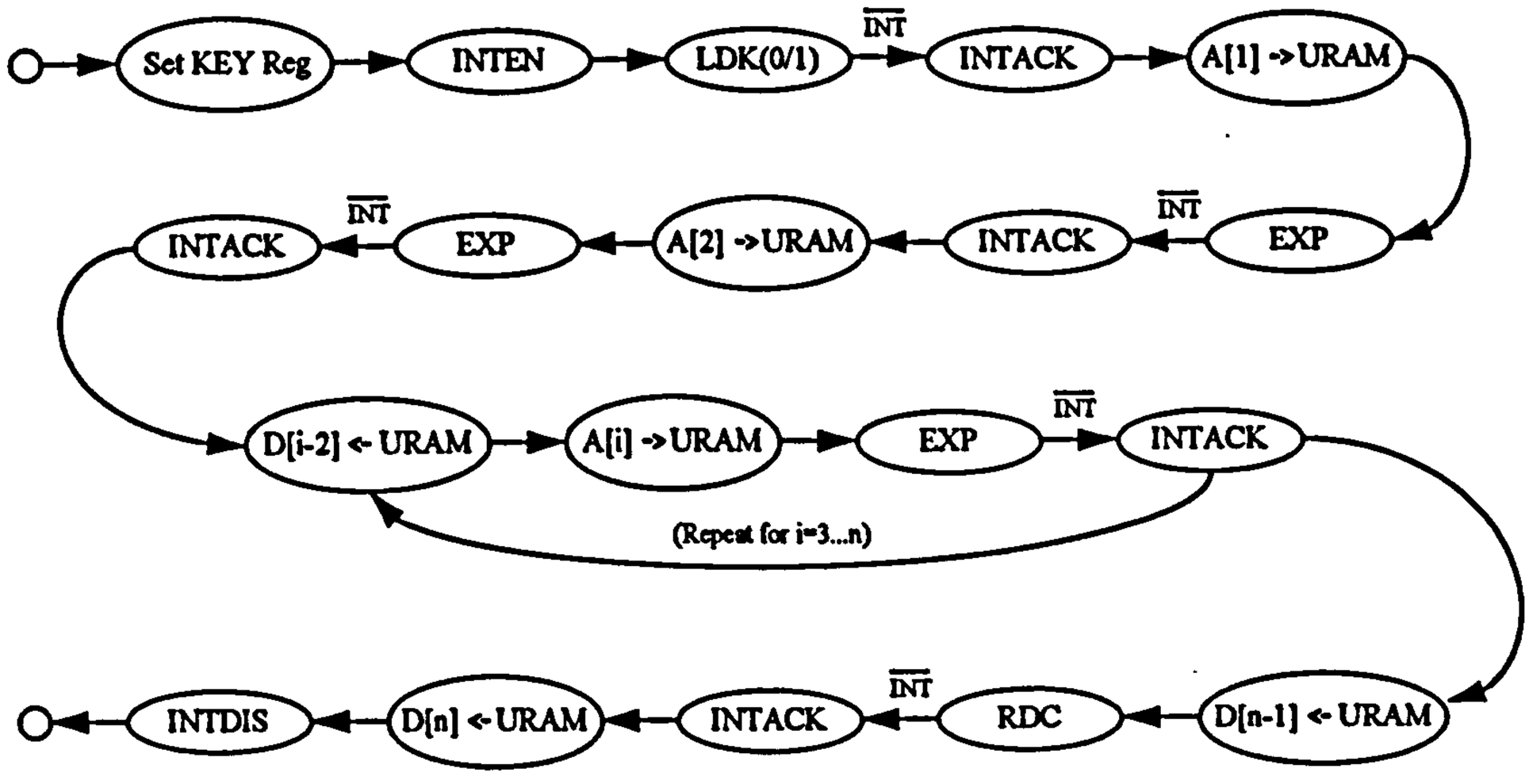


Figure 8.8: Multiple exponentiation state transition diagram.

8.7 Performance

Based on the WHiSpER schematics of Appendix B and the LSM, TSM, ESM and RSM state machine diagrams of Appendix A (noting that the SMC clock signal oscillates at half the frequency of the main clock signal), then the expected performance of the WHiSpER chip can be summarized as follows.

8.7.1 Key Load Process

The number of clock cycles required by the LDK command is the sum of the cycles required to load M into the multiplier and to find the most-significant-bit of the selected exponent. They are

$$M \text{ load cycles} = 1048$$

$$e_{msb} \text{ search cycles} = 12 \text{ to } 5102$$

8.7.2 Transfer Process

Assuming the RSM does not require more cycles than the ESM to complete, then the number of clock cycles required by the TSM to transfer operands and results into and out of the multiplier's registers is

$$\text{Transfer cycles} = 1036$$

8.7.3 Exponentiation Process

The number of multiplications that have to be performed for each exponentiation is

1. Pre-conversion and e_{msb}: 1 multiplication.

2. Processing for $e_{msb-1} \ldots e_0$: average of $1\frac{1}{2}$ multiplications per bit.

3. Post-conversion: 1 multiplication.

For a 506-bit exponent this gives an average number of 760 multiplications. With the number of cycles per multiplication equal to 148, this gives

$$\text{Exponentiation cycles} = 11248$$

8.7.4 Reduction Process

The multi-precision addition of N to C^* by the RSM state machine is performed in 16-byte blocks. It therefore takes 4 of these blocks to add a single N. Since we have a maximum

of $15 \cdot N$ to add, then a maximum of 60 blocks will be needed. Simulations have confirmed that a minimum reduction rate of 1 block per multiplication is achieved by the RSM, and so therefore any exponentiation where the exponent is greater than 60 bits will allow the reduction to take place in parallel with the exponentiation. i.e. the reduction process will not add to the exponentiation time.

8.7.5 RSA Throughput

Assuming that the key has been loaded by the LDK command, then the throughput of the WHiSpER chip is defined to be the time taken to execute consecutive EXP commands. The number of cycles required to complete an EXP command is the sum of the cycles required by the transfer and exponentiation processes.

With a clock rate of 25MHz, then the clock period is 40ns and we have

$$\text{Average EXP time} = (1036 + 112480) \cdot 40\text{ns} = 4.541\text{ms}$$

which gives

$$\text{Average EXPs per second} = 220$$

which, for a 506-bit key, gives a throughput of

$$\text{Average throughput} = 111\text{kbps}$$

For the worst-case exponent (where all exponent bits are '1') we need 1012 multiplications and the figures are

$$\text{Maximum EXP time} = 6.032\text{ms}$$

$$\text{Minimum EXPs per second} = 165$$

$$\text{Minimum throughput} = 83\text{kbps}$$

and so we see that this is still greater than the threshold rate of 64kbps.

8.7.6 Gate-Array Selection

From Section 8.3.2 we have the size of the MME circuit set at approximately 54000 gates. Assuming the remaining control circuitry occupies less than 30% of the area of the MME, then the total number of gates in the WHiSpER chip will be approximately 70000.

The conventional rule for gate-array design is that, for the design to be easily mapped into the array, it should not use more than 50% of the array's available gates. Applying this rule to the WHiSpER design suggests a gate-array with at least 140000 gates. Such an array is the CLA77XXX which has 181260 available gates. Thus the WHiSpER design would use approximately 40% of the array's gates, allowing room for manouevre in the layout process.

8.7.7 Power Consumption

In order to calculate the expected power consumption of the WHiSpER chip we need to estimate the number of gates that change state during each cycle of the exponentiation process. Looking at the MME we see that the Y and M registers will remain constant but the X and B registers will change every cycle. It is also likely that the $x(i) \cdot Y$ and $z(i) \cdot M$ generation circuits, the adder array, the accumulator and the carry-propagate adder will all change state on each cycle. This gives

$$4 \cdot \Omega_{MUX} + 4 \cdot \Omega_{FA} + 4 \cdot \Omega_{FF} + \Omega_{ADT}/8 = 91$$

gates per bitslice changing state on each cycle. With 518 bitslices, this gives approximately 47000 gates changing state on each cycle. Using the figure of 7uW/MHz power dissipation per gate for the CLA70000 gate-array, and also using the rule-of-thumb from [83] that 15% of the control circuitry changes state on each cycle, then we have a total power dissipation of

$$\text{Power Dissipation @ 25MHz} \approx 9 \text{ Watts}$$

a not inconsiderable figure.

The packaging technology used for the WHiSpER chip therefore needs to be of the Power type (see Section 8.1). Also, it would seem advisable that any implementation of the chip use a package mounted heatsink and fan combination, similar to the units used by modern high-performance microprocessors.

8.8 Testability

Post-fabrication testing of an integrated circuit involves applying a set of test stimuli (the 'testvectors') to the device's inputs and monitoring the state of the device's outputs. The idea is that the testvectors will cause a state-change on the output of every gate in the device, and that these changes will, ultimately, be detectable on the device outputs. i.e. if there is a fault in the device such that a particular gate does not change state when it should do, then either it is immediately detectable, or else its effect will be propagated to other parts of the circuit where, sooner or later, it will show itself as an unexpected state-change on a device output.

The goal of full testability is to create both a design and a set of testvectors such that all possible fabrication faults can be detected in as short a time as possible. In practice, the percentage of detectable faults (the fault coverage) is rarely 100%, but a minimum coverage level of 95% is often quoted.

The testing of the WHiSpER chip is twofold.

- A signature analysis technique is used for testing the MME circuit.
- All other circuitry is tested by exhaustive stimuli.

The signature analysis technique lets the MME perform a small exponentiation with known input operands and then examines the result. The justification behind this approach is that,

1. it can easily be ensured that all MME gates change state during the exponentiation,

and

2. due to the nature of modular arithmetic, a single fault affecting one bit of the exponentiator circuit will very likely affect other bits too. This is simply a consequence of the diffusion property of modular arithmetic that makes it attractive for cryptologic purposes.

Thus we see that it is very likely that a fault in the MME will show up in the result of an exponentiation.

The remaining circuitry of the WHiSpER chip can be tested by creating a set of test vectors that toggle each node within the design. i.e. registers are loaded with complementary values, state machines have all possible state transitions exercised, etc.

To increase the observability of potential faults within the design, several auxilliary device inputs and outputs have been included. For full details see Appendix B, but in essence the extra facilities offered are controlled by a TEST input signal which, when made active high, allows the examination of internal control and data signals on device output pins.

Although not directly concerned with fault-testing, another facility provided by the WHiSpER design is an externally programmable delay time for the MME's carry-propagation adder. The rationale behind this is that, since the carry-propagation adder is constructed from a chain of 64 8-bit ADT Macros, then a slight deviation in Δ_{ADT} from that expected could lead to large difference in total addition time. The RAADsel(1:0) signal selects between four alternative delay times (the default is the shortest with RAADsel = 00) and its presence in the design is purely a precautionary measure that will also allow full evaluation of the prototypes. i.e. allow the performance of the RSD adder array to be determined separately from that of the carry-propagate adder.

8.9 The WHiSpER PC-Card

Using the WHiSpER chip in conjunction with an IBM PC compatible requires that an interface card be built that slots into the PC's XT-bus. Specifications for the XT-bus can be found in [87].

The XT-bus address map is shown in Figure 8.9. Due to the fact that early interface

XT-bus Address Lines														
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
High Address Decoding 64 Slots						CARD SLOT	Low Address Decoding 512 Ports								

Figure 8.9: XT-bus address map.

cards used only the lower 10 bits for address decoding (ignoring the high 6 bits) then for interface cards that require more than a few port addresses it is necessary to implement the interface card address decoding in a non-obvious manner.

The WHiSpER chip needs a 512-byte address space which requires 9 address bits. We construct these 9 bits from the top 6 bits and the bottom 3 bits of the XT-bus address. Card access is then decoded from XT-bus address bits 9 downto 3 by DIP-switches. Thus, when looking at just the lower 10 bits of XT-bus address, the WHiSpER PC-Card will appear as a block of 8 consecutive port addresses at some location defined by the DIP-switch settings. However, access to the entire WHiSpER memory map is possible because the interface card also decodes the upper 6 bits of address. In summary, if $A_{15} \ldots A_0$ refer to the XT-bus address lines, then address decoding is performed as follows,

- A_9 must be a '1',
- $A_8 \ldots A_3$ compared with DIP-switch,
- $A_{15} \ldots A_{10} A_2 \ldots A_0$ map to WHiSpER address lines Addr(9:0).

Access to the card from software is performed by X86 assembler 'in' and 'out' instructions

(or HLL library functions).

The circuit diagram for the WHiSpER PC-Card is shown in Figure 8.10.

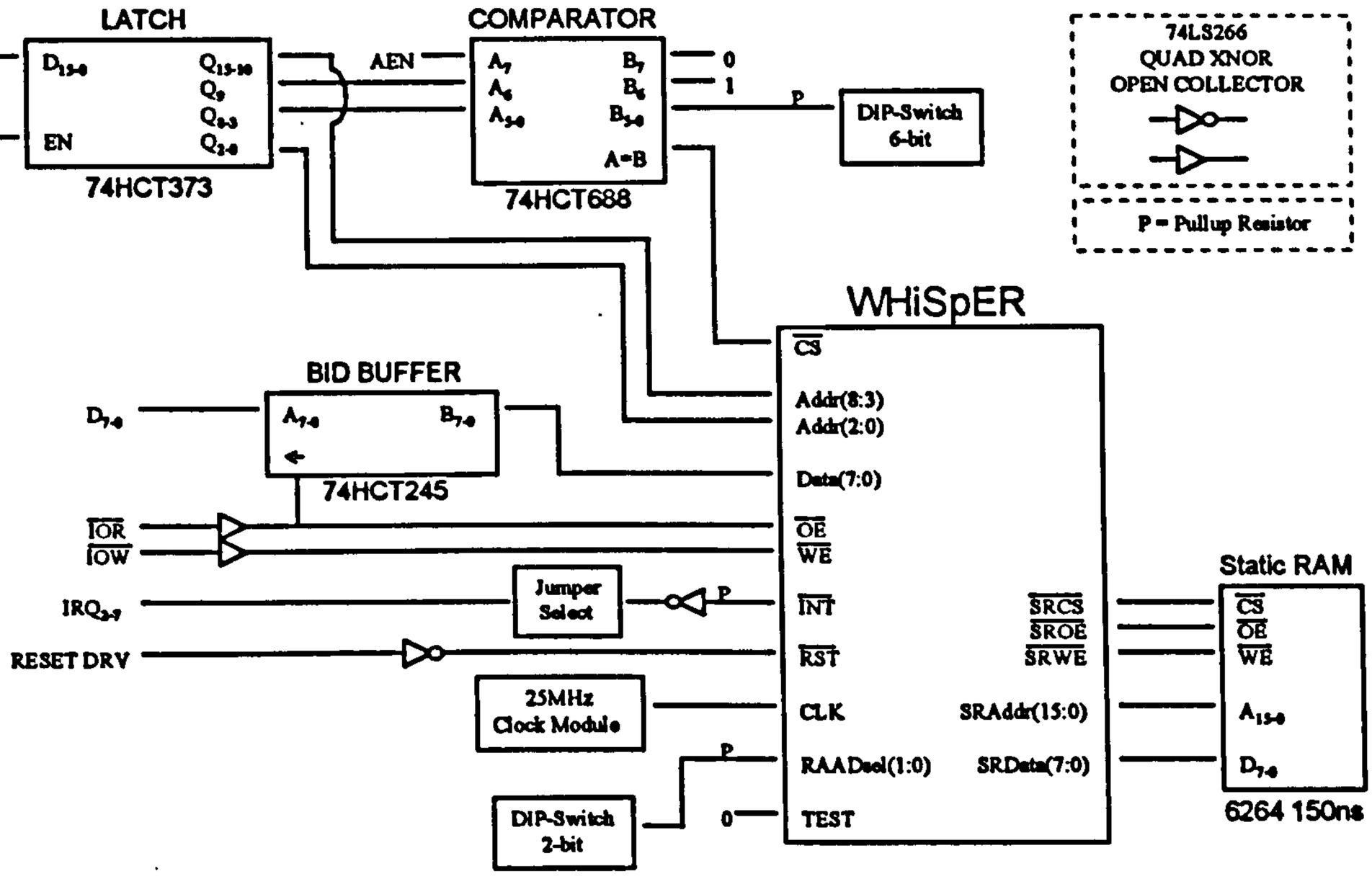


Figure 8.10: The WHiSpER PC-Card schematic.

8.10 Summary

This chapter has presented the design of the WHiSpER chip. Selection of the appropriate Montgomery multiplier circuit was performed based on the technology issues of the CLA70000 series gate array. Furthermore, the exponentiation algorithm chosen favoured reduced circuit area and efficiency over raw speed. The architecture of the chip was described, showing how the use of operand/result double-buffering techniques and an external SRAM device leads to an efficient, high-throughput device. This was followed by a description of the device's operation and an analysis of its expected performance. Finally, the details of a simple WHiSpER based IBM PC card were given.

Chapter 9

Conclusions

In the last chapter the WHiSpER chip was presented, and it was shown that this chip could perform RSA exponentiations at a rate of over 100kbps for moduli of up to 506 bits in length.

In this chapter the thesis is brought to a conclusion by discussing

1. how the WHiSpER chip can be used in RSA cryptosystems with moduli of around 1000 bits in length,

2. how the WHiSpER chip can be used to generate RSA keys, and finally

3. a discussion of the achievements of the work presented here followed by ideas for further work.

9.1 The WHiSpER Chip and Extended Moduli

A method for speeding up the RSA cryptosystem that has been presented in many publications (see for example [79]) relies upon the use of the Chinese Remainder Theorem (CRT) for exponentiation with the secret key, and using a small preset exponent in the public key.

The use of the CRT requires knowledge of the prime factors of N. However, since the

CRT is used in the secret exponentiation then it is not unreasonable to assume that the prime factors of N are also available to the entity that has access to the secret exponent. Indeed, in [10] it is shown that the prime factors of N can be easily calculated if the public and secret exponents are known.

The use of a small public exponent, such as the number 3, is acceptable [88] so long as certain measures are incorporated in the cryptosystem protocol (see [1]). For example, a small field in the pre-exponentiation framing operation is required to ensure that the probability of two messages being identical is sufficiently low.

9.1.1 CRT Exponentiation Using The WHiSpER Chip

The RSA modulus N is the product of two prime numbers P and Q. As we saw in Section 2.6.5 we can perform an exponentiation modulo N by performing exponentiations modulo P and Q followed by two multiplications modulo N. That is,

$$\left\langle A^E \right\rangle_N = \left\langle \left\langle (A_P)^E \right\rangle_P \cdot \left\langle Q^{-1} \right\rangle_P \cdot Q + \left\langle (A_Q)^E \right\rangle_Q \cdot \left\langle P^{-1} \right\rangle_Q \cdot P \right\rangle_N$$

where

$$\begin{aligned} A_P &= \langle A \rangle_P \\ A_Q &= \langle A \rangle_Q \end{aligned}$$

Now, since P and Q are prime, then from Fermat's Little Theorem (Theorem 4) we have

$$\begin{aligned} \left\langle (A_P)^E \right\rangle_P &= \left\langle (A_P)^{\langle E \rangle_{P-1}} \right\rangle_P \\ \left\langle (A_Q)^E \right\rangle_Q &= \left\langle (A_Q)^{\langle E \rangle_{Q-1}} \right\rangle_Q \end{aligned}$$

So defining

$$\begin{aligned} E_P &= \langle E \rangle_{P-1} \\ E_Q &= \langle E \rangle_{Q-1} \end{aligned}$$

and

$$D_P = \left\langle (A_P)^{E_P} \right\rangle_P$$

$$D_Q = \left\langle (A_Q)^{E_Q} \right\rangle_Q$$

then we have

$$D = \left\langle A^E \right\rangle_N = \left\langle D_P \cdot Q'_P + D_Q \cdot P'_Q \right\rangle_N$$

where

$$Q'_P = \left\langle Q^{-1} \right\rangle_P \cdot Q$$

$$P'_Q = \left\langle P^{-1} \right\rangle_Q \cdot P$$

are pre-computed constants. For N and E around 1000 bits, and so assuming P and Q of around 500 bits, then the above calculation of D can roughly halve the number of multiplications involved compared to non-CRT methods. Furthermore, since the calculations of D_P and D_Q involve only 500-bit multiplications, then each multiplication takes approximately half the time of non-CRT multiplications (half the time for hardware multipliers. In software, using multi-precision arithmetic techniques, the multiplications would take approximately one quarter of the time).

Thus, with a WHiSpER and host computer combination, the CRT method could be used with moduli whose prime factors are each less than 506 bits in length, as follows.

1. Pre-calculate on the host

$$M_P = P \cdot \left\langle -P^{-1} \right\rangle_{2^4}$$

$$M_Q = Q \cdot \left\langle -Q^{-1} \right\rangle_{2^4}$$

$$E_P = \langle E \rangle_{P-1}$$

$$E_Q = \langle E \rangle_{Q-1}$$

$$H_P = \left\langle 2^{1024} \right\rangle_{M_P}$$

$$H_Q = \left\langle 2^{1024} \right\rangle_{M_Q}$$

$$Q'_P = \left\langle Q^{-1} \right\rangle_P \cdot Q$$

$$P'_Q = \left\langle P^{-1} \right\rangle_Q \cdot P$$

2. Load P, H_P and E_P into the first SRAM key storage bank. Load Q, H_Q and E_Q into the second SRAM key storage bank.

3. To perform the exponentiation $D = \left\langle A^E \right\rangle_N$ then

   (a) calculate A_P and A_Q on the host,

   (b) write A_P to WHiSpER and issue LDK, EXP and RDC commands for first key store,

   (c) read D_P from WHiSpER,

   (d) write A_Q to WHiSpER and issue LDK, EXP and RDC commands for second key store,

   (e) read D_Q from WHiSpER,

   (f) calculate $D = \left\langle D_P \cdot Q'_P + D_Q \cdot P'_Q \right\rangle_N$ on the host.

For maximum throughput the host should perform the calculations of A_P, A_Q and D in parallel with the WHiSpER exponentiations. To see that this is possible we can examine the software modular multiplication implementations of Comba [34] and Bong et al. [35]. Bong shows that an 8MHz 80286 PC can perform approximately 200 512-bit modular multiplications per second. Since doubling the modulus size approximately quadruples the calculation time of multi-precision arithmetic routines, this corresponds to approximately 50 1024-bit multiplications per second. Using a mixture of 512-bit and 1024-bit operations, then it should be possible to perform 100 512-bit multiplications plus 25 1024-bit multiplications per second. Assuming that a 100MHz Pentium processor is at least 10-times faster than an 8MHz 80286, then a modern PC should be able to perform approximately 1000 512-bit and 250 1024-bit multiplications in one second. Referencing

Section 8.7 we see that the WHiSpER chip performs approximately 220 exponentiations per second. Therefore there should be ample time for the host device to perform two 500-bit reductions and two 1000-bit multiplications in the time it takes the WHiSpER chip to perform two exponentiations. Thus the throughput of the CRT method will be limited by the WHiSpER chip.

The time taken by the WHiSpER chip to perform two exponentiations with different moduli is equal to twice the time required to execute the LDK, EXP and RDC commands. Approximating the figures of Section 8.7 we have

$$\begin{aligned} \text{key load} &= 5000 \text{ cycles} \\ \text{transfer} &= 1000 \text{ cycles} \\ \text{exponentiation} &= 112000 \text{ cycles} \\ \text{reduction} &= 7000 \text{ cycles} \end{aligned}$$

which gives a total of approximately 125000 cycles. With the WHiSpER chip clocked at 25MHz this corresponds to approximately 200 exponentiations per second. Since we need two WHiSpER exponentiations per modulo N exponentiation this corresponds to approximately 100 1000-bit exponentiations per second. In other words, for keys around 1000 bits in size,

$$\text{RSA throughput with CRT} = 100\text{kbps}$$

9.1.2 Host Exponentiation with a Small Exponent

As we saw above, it should be possible to perform 500 1000-bit modular multiplications per second on a fast PC. To achieve a 100kbps throughput rate for RSA with a small public exponent, then the exponent, E_p will be limited to the following values (remembering that the exponent must be odd)

$$E_p \in \{3, 5, 7, 9, 11, 13, 17\}$$

Recalling from Section 2.7 that the exponents must be coprime to $\Phi(N)$, then the choice of which exponent to use depends on which of them is coprime to both $P-1$ and $Q-1$. To ensure that one of them is coprime it may be necessary to create the primes P and Q with this condition in mind. This can be done quite simply using Euclid's algorithm with each candidate prime in the key generation process.

Thus it is possible to use the WHiSpER chip and a fast PC to implement the RSA cryptosystem with key lengths around 1000 bit and still achieve encryption rates of approximately 100kbps.

9.2 The WHiSpER Chip and Key Generation

As was shown in Section 2.7, RSA key generation is essentially prime number generation. The technique most commonly used to generate the large prime numbers needed for RSA keys is the primality testing of large odd random numbers. An efficient primality testing algorithm is that of Rabin in [89]. The algorithm is probabilistic in nature, which means that there is a small probability that the algorithm will declare an integer to be prime when it is not. However, this probability can be reduced to an arbitrarily small level with enough applications of the algorithm.

9.2.1 Rabin's Primality Test

In [89] Rabin shows how to test an arbitrary odd integer P for primality. First, write P as

$$P = 2^f \cdot E + 1$$

where E is an odd number and $f \geq 1$. A random number $A \in [0, P-1]$, called a witness to the primality or otherwise of P is then used according to the pseudo-code algorithm shown in Figure 9.1. The algorithm may return FALSE or TRUE, and their meanings are as follows,

```
1.  S := < A^E >_P
2.  IF (S = 1) OR (S = P-1)
3.    RETURN TRUE
4.  ENDIF
5.  FOR j = 1 to f
6.    S := S^2
7.    CASE OF
8.      S = 1   : RETURN FALSE
9.      S = P-1 : RETURN TRUE
10.   ENDCASE
11. ENDFOR .
12. RETURN FALSE
```

Figure 9.1: Rabin's primality test.

- FALSE: P is definitely composite,

- TRUE: P is probably prime.

The probability that the algorithm returns TRUE when P is really composite is 1 in 4.

In practice, a sequence of randomly chosen witnesses

$$A[1], A[2], \ldots, A[n]$$

are used to test the primality of P. If Rabin's test returns TRUE for every $A[i]$, then the probability that we are wrong in declaring P to be a prime is 2^{-2n}.

For example, with 10 randomly generated witnesses $A[1], A[2], \ldots, A[n]$ in the range $[0, P-1]$ then if, when applied to each $A[i]$ in turn, the algorithm returns TRUE in every case, then the probability that P is actually composite is approximately one in a million.

9.2.2 Primality Testing on WHiSpER

Using Rabin's primality test with the WHiSpER chip it is possible to test odd integers of up to 506 bits for primality.

With $P = 2^f \cdot E + 1$ then Rabin's test requires an exponentiation and f squarings modulo P. To do this on the WHiSpER chip requires pre-calculating

$$M = P \cdot \left\langle -P^{-1} \right\rangle_{2^4}$$

and loading an SRAM key bank with

$$\begin{aligned} N &= P \\ H &= \left\langle 2^{1024} \right\rangle_M \\ E_p &= E \\ E_s &= 2 \end{aligned}$$

The first exponentiation in Rabin's test is then started, after $A[1]$ has been loaded into the URAM, by issuing the commands LDK(0), EXP and RDC. Further squarings are then performed by the commands LDK(1), EXP, EXP,...,RDC until the algorithm terminates. After each exponentiation the result can be read from the URAM.

If the algorithm returns TRUE for $A[1]$ then $A[2]$ is tested, if this returns true then $A[3]$ is tested and so on until either the algorithm returns FALSE for some $A[i]$ or the sequence of witnesses is exhausted. In the latter case P is declared to be prime.

To estimate the time required to generate a k-bit prime number we must first estimate the time required to execute Rabin's test for one witness A. To do this we need to know the average number of squarings that will be performed for randomly chosen P.

Consider a randomly chosen odd number P, then

- there is a probability of $\frac{1}{2}$ that the bit-vector of P will end in '11',
- there is a probability of $\frac{1}{4}$ that the bit-vector of P will end in '101',
- there is a probability of $\frac{1}{8}$ that the bit-vector of P will end in '1001',

and so on. For Rabin's test this means that

- there is a probability of $\frac{1}{2}$ that the test will require one exponentiation and one squaring operation,
- there is a probability of $\frac{1}{4}$ that the test will require one exponentiation and two squaring operations,

- there is a probability of $\frac{1}{8}$ that the test will require one exponentiation and three squaring operations,

and so on. In general this implies that for randomly chosen k-bit odd integers the number of squaring operations required by the algorithm will be

$$\sum_{i=1}^{k-1} i \cdot 2^{-i}$$

which quickly converges to the value 2.

Thus, on average, we will have to perform one exponentiation and two squaring operations when applying Rabin's test to randomly chosen integers. If these operations are performed by the WHiSpER chip for many prime candidates during the search for a prime number, and with the calculation of M and H for each candidate performed by the host in parallel with WHiSpER operation, then the time required to test each candidate will be the time required by the commands LDK(0), EXP, RDC, LDK(1), EXP, EXP and RDC. For P around 500-bit this requires approximately 120000 cycles of WHiSpER's clock, which at 25Mhz corresponds to approximately 5ms. Therefore approximately 200 primality tests can be performed per second.

If it is assumed that all composite numbers fail the primality test on at most the second witness (as is most often seen in practice), and using the prime distribution approximation that $\log_e P$ is the probability that numbers of the size of P are likely prime, then we should be able to discover a 500-bit prime with fewer than 350 primality tests. This corresponds to less than 2 seconds using the WHiSpER chip. A 1000-bit RSA key requires two such primes, and therefore it should be possible to generate a 1000-bit RSA key in less than 4 seconds.

For 500-bit RSA keys the time required by the WHiSpER chip to perform 250-bit primality tests reduces to less than 3ms. Thus over 350 such tests can be performed per second. Since 250-bit primes are about twice as common as 500-bit primes, then fewer than 175 tests should be needed to find a 250-bit prime. Thus it should be possible to

generate a 500-bit RSA key in 1 second.

9.3 Achievements

The achievements of this project are threefold; a new algorithm, an efficient architecture and the WHiSpER chip.

9.3.1 A New Algorithm

The MMDDAMMM algorithm allows fast and efficient Montgomery multipliers to be realised in VLSI hardware. It has two clear advantages over other algorithms,

- the calculation of the modulus multiple, $Z_i \cdot M$, that has to be added to the partial result during each iteration – traditionally the bottleneck to processing speed in modular multiplication – has been simplified to such an extent that it no longer limits multiplier performance, and

- the range of Z_i and X_i is the same, allowing for efficient high-radix implementations of the algorithm.

9.3.2 An Efficient Architecture

Using an RSD approach to the design of an iterative Montgomery multiplier enables the use of string recoding techniques to further improve the efficiency and speed of the multiplier. The optimizations incorporated into the MMDDAMMM algorithm allow the recoding scheme to be used to full advantage. The performance bottleneck in this new architecture is now the architecture itself, i.e. the delay of signals through the adder array.

9.3.3 The WHiSpER Chip

A high-speed 506-bit RSA processor, the WHiSpER chip, has been designed and simulated and is expected to be able to perform RSA encryption/decryption at an average rate

of 111kbps. At the time of writing, the WHiSpER chip is awaiting final layout and fabrication. Negotiations with GEC Plessey are in progress to achieve this end.

The fastest known RSA processor using a similar technology is the Cryptech chip (see Chapter 5). This chip can perform a one-off 512-bit exponentiation at an equivalent rate of 32kbps. Sustained encryption rates, however, will be lower than this since the chip does not incorporate operand/result input and output buffering. Therefore it is expected that the WHiSpER chip will show, approximately, a fourfold increase in throughput compared to its nearest rival.

Applications of the WHiSpER chip include its use as an RSA processor in high-throughput, computationally intensive, crypto-processing engines. These are found in the security service providers of computer security systems such as [90] and [91]. Also, because encryption rates of over 64kbps are guaranteed by the WHiSpER chip, then it could be used by moderate-speed communication networks (such as ISDN, see [92]) to provide a transparent security function for users.

9.4 Further Work

To achieve greater throughput for RSA hardware would require investigations into the following areas.

9.4.1 Exponentiation Algorithms

Although the exponentiation algorithms discussed in Chapter 3 are efficient when implemented in hardware, they are not the fastest available.

Addition Chain Exponentiation

An addition chain for a given number can be defined, from [93], as a list of numbers such that,

- the first number is one,
- every number is the sum of two earlier numbers, and
- the given number is the last in the list.

To perform an exponentiation, the numbers can be viewed as the intermediate exponents that are calculated during the exponentiation process.

The Right-to-Left and Left-to-Right exponentiation algorithms given in Chapter 3 are special case addition chains where each number in the chain is either the sum of the previous number with itself (the squaring operation), or the sum of the previous number and one (the multiply operation).

To show that generalized addition chains can permit faster exponentiation then consider raising a number to the power 15. Using the standard Right-to-Left technique, the addition chain becomes

$$1, 2, 3, 6, 7, 14, 15$$

which requires 3 squarings and 3 multiplications. A different addition chain for the number 15 is

$$1, 2, 3, 6, 12, 15$$

which still requires 3 squarings but only 2 multiplications.

The use of addition chains for fast exponentiation has been investigated by various authors, see for example [25], [94] and [95]. The main problem is that computing the shortest addition chain for a given number is known to be an NP-complete problem. However, algorithms are available that can compute addition chains that are up to 20% shorter [93] than the Right-to-Left technique.

With respect to modular exponentiation hardware, the main problem with using addition chains is that several intermediate results may have to be stored. This is because, as the exponentiation is proceeding, the current multiplication is allowed to be the product of any two of the previous intermediate results.

Further work would allow the WHiSpER chip to be modified in such a way that would allow extra register storage, for intermediate exponentiation results, to be made available within the device, and to implement increased flexibility in the exponentiation algorithm. This increased flexibility would allow pre-computed addition chains to be somehow encoded within the exponent space of the SRAM, defaulting to standard binary when no efficient chain can be found.

Exponentiation with Table-lookup

In [96] and [97] exponentiation algorithms are proposed which allow the number of multiplications used in the calculation of $\langle A^E \rangle_N$ to be reduced. The technique relies on computing a table of powers of A.

This idea could be used by the WHiSpER chip by reserving an area of the SRAM for the table. The table could be computed by WHiSpER in a first-stage exponentiation process, and then used subsequently to calculate $\langle A^E \rangle_N$.

9.4.2 Improved Technology

High-performance VLSI integrated circuits, such as modern RISC microprocessors, use silicon-efficent full-custom design techniques and advanced manufacturing process technologies. Current state-of-the-art technology uses 3.3v 4-level metal processes allowing sub-0.5 micron features with die sizes that permit more than 3 million transistors on a chip. This leads to clock frequencies in excess of 200MHz and power dissipations of over 20W. See, for example, the MIPS Technologies' R10000 microprocessor [98], DEC's Alpha AXP microprocessor [99] and IBM/Motorola's PowerPC microprocessors [100] [101].

Using 0.5 micron technology to implement the $b = 2$ MMDDAMMM recoded RSD multiplier of Chapter 7 would surely allow this multiplier to be clocked at 100MHz. For moduli of around 500 bits then approximately 200 thousand clock cycles are required per exponentiation. This gives an RSA throughput of approximately 250kbps. From

Chapter 8 we see that the size of the multiplier is around 30 thousand gates or 120 thousand transistors.

Assuming that pipelining techniques similar to those of Section 6.4.3 can be applied to this multiplier, then we see that a four-stage pipelined architecture would probably use no more than 500 thousand transistors and yet be capable of performing RSA exponentiations in the region of 1Mbps. More pipelining and parallelism would produce still higher rates.

Therefore, more work needs to be done in investigating pipelined implementations of the optimized Montgomery multipliers presented in Chapter 7 with regard to large full-custom designs using fast sub-micron technologies.

9.4.3 Towards a New Architecture

Recent proposals [102] [103] suggest using a Residue Number System (RNS) approach to long-integer modular arithmetic. The RNS system has the great advantage that when performing modular arithmetic modulo an integer constructed from many small primes (called the base of the RNS system, see [104]), addition and multiplication can be performed by many small calculations that proceed completely in parallel. The time required to complete one of these operations is thus very short.

The main disadvantage of RNS however, is that, when used to perform arithmetic modulo an integer coprime to the base of the RNS system (such as will happen with RSA exponentiation), then this will involve a division-like operation and this is very difficult to do in RNS. The basic problem is that the RNS number system is a non-weighted system, and so comparison and division are not easily achievable.

In summary, it is not known whether RNS systems will ever provide an efficient alternative to weighted number systems when it comes to implementing high-speed RSA hardware.

9.5 Summary

A new Montgomery multiplication algorithm and an efficient architecture have yielded the WHiSpER chip. In combination with the Chinese Remainder Theorem this chip can be used to implement the RSA cryptosystem with keys of up to 1000 bits in length and ciphering rates of approximately 100kbps. Primality testing can also be performed by this chip, with RSA key generation typically taking only a few seconds.

Bibliography

[1] Gustavus J. Simmons, editor. *Contemporary Cryptology - The Science of Information Integrity.* IEEE Press, 1992.

[2] Jennifer Seberry and Josef Pieprzyk. *Cryptography - An Introduction to Computer Security.* Prentice-Hall, 1989.

[3] D. W. Davies. *Security for Computer Networks.* Wiley-Interscience, 1984.

[4] C.E. Shannon. Communication theory of secrecy systems. *Bell Systems Technical Journal*, 28:656–715, 1949.

[5] Bruce Schneier. The IDEA encryption algorithm. *Dr. Dobb's Journal*, December 1993.

[6] Eli Biham and Adi Shamir. Differential cryptanalysis of DES-like cryptosystems. In *Advances in Cryptology: CRYPTO 90.* Springer-Verlag, 1991.

[7] Adi Shamir. Differential cryptanalysis of the full 16-round DES. In *Advances in Cryptology: CRYPTO 92.* Springer-Verlag, 1993.

[8] G. Carter, A. Clark, E. Dawson, and L. Nielsen. Analysis of DES double key mode. In *Information Security – The Next Decade.* Chapman and Hall, 1995.

[9] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4), 1985.

[10] Neal Koblitz. *A Course in Number Theory and Cryptography.* Springer-Verlag, 1987.

[11] L. Harn. Public-key cryptosystem design based on factoring and discrete logarithms. *IEE Proceedings: Computers and Digital Techniques*, 141(3), May 1994.

[12] Chih-Chwen Chuang and James George Dunham. Matrix extensions of the RSA algorithm. In *Advances in Cryptology: CRYPTO 90.* Springer-Verlag, 1991.

[13] Paul C. van Oorschot. A comparison of practical public-key cryptosystems based on integer factorization and discrete logarithms. In *Advances in Cryptology: CRYPTO 90.* Springer-Verlag, 1991.

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

[15] Wiebren de Jonge and David Chaum. Attacks on some RSA signatures. In *Advances in Cryptology: CRYPTO 85.* Springer-Verlag, 1986.

[16] Y. Desmedt. A chosen text attack on the RSA cryptosystem and some discrete logarithm schemes. In *Advances in Cryptology: CRYPTO 85.* Springer-Verlag, 1986.

[17] John M. DeLaurentis. A further weakness in the common modulus protocol for the RSA cryptoalgorithm. *Cryptologia*, July 1984.

[18] James A. Davis and Diane B. Holdridge. Factorization of large integers on a massively parallel computer. In *Advances in Cryptology: EUROCRYPT 88.* Springer-Verlag, 1989.

[19] Kurt Kleiner. Squeamish ossifrage dents electronic armour. *New Scientist*, 7th May 1994.

[20] Alan Baker. *A Concise Introduction to the Theory of Numbers.* Cambridge University Press, 1984.

[21] M. R. Schroeder. *Number Theory in Science and Communication.* Springer-Verlag, 1984.

[22] Kenneth H. Rosen. *Elementary Number Theory and its Applications.* Addison-Wesley, 1984.

[23] Charles C. Pinter. *A Book of Abstract Algebra.* McGraw-Hill Publishing Company, 2 edition, 1990.

[24] Keith Devlin. *Microchip Mathematics - Number Theory for Computer Users.* Shiva Publishing Ltd, 1984.

[25] Donald E. Knuth. *The Art of Computer Programming*, volume 2: Semi-Numerical Algorithms. Addison-Wesley, 2 edition, 1981.

[26] Kai Hwang. *Computer Arithmetic - Principles, Architecture and Design.* John Wiley and Sons, 1979.

[27] Akhilesh Tyagi. A reduced-area scheme for carry-select adders. *IEEE Transactions on Computers*, 42(10), October 1993.

[28] Mark. R. Santoro and Mark A. Horowitz. SPIM: A pipelined 64x64-bit iterative multiplier. *IEEE Journal of Solid-State Circuits*, 24(2), April 1989.

[29] Masoto Nagamatsu, Shigeru Tanaka, Junji Mori, Katsusi Hirano, Tatsuo Noguchi, and Kazuhisa Hatanaka. A 15-ns 32x32-b cmos multiplier with an improved parallel structure. *IEEE Journal of Solid-State Circuits*, 25(2), April 1990.

[30] Stamatis Vassiliadis, Eric M. Schwarz, and Boik M. Sung. Hard-wired multipliers with encoded partial products. *IEEE Transactions on Computers*, 40(11), November 1991.

[31] G. R. Blakley. A computer algorithm for calculating the product AB modulo M. *IEEE Transactions on Computers*, C-32(5):497–500, May 1983.

[32] Per Brinch Hansen. Multiple-length division revisited: A tour of the minefield. *Software - Practice and Experience*, 24(6), June 1994.

[33] A. Selby and C. Mitchell. Algorithms for software implementations of RSA. *Proceedings of the IEE*, 136(3), May 1989.

[34] P. G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4), 1990.

[35] Dieter Bong and Cristoph Ruland. Optimized software implementations of the modular exponentation on general purpose microprocessors. *Computers and Security*, 8:621–630, 1989.

[36] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology: CRYPTO 86*. Springer-Verlag, 1987.

[37] Stephen R. Dusse and Burton S. Kaliski Jr. A cryptographic library for the motorola DSP56000. In *Advances in Cryptology: EUROCRYPT 90*. Springer-Verlag, 1991.

[38] Dominique de Waleffe and Jean-Jacques Quisquater. CORSAIR: A smart card for public key cryptosystems. In *Advances in Cryptology: CRYPTO 90*. Springer-Verlag, 1991.

[39] P. A. Findlay and B. A. Johnson. Modular exponentation using recursive sums of residues. In *Advances in Cryptology: CRYPTO 89*. Springer-Verlag, 1990.

[40] Andre Vandemeulebroecke, Etienne Vanzieleghem, Tony Denayer, and Paul G. A. Jespers. A new carry-free division algorithm and its application to a single-chip 1024-b RSA processor. *IEEE Journal of Solid-State Circuits*, 25(3), June 1990.

[41] Paolo Montuschi and Luigi Ciminiera. Over-redundant digit sets and the design of digit-by-digit division units. *IEEE Transactions on Computers*, 43(3), March 1994.

[42] Tony M. Carter and James E. Robertson. Radix-16 signed-digit division. *IEEE Transactions on Computers*, 39(12), December 1990.

[43] David M. Mandelbaum. A systematic method for division with high average bit skipping. *IEEE Transactions on Computers*, 39(1), January 1990.

[44] Eric M. Schwarz and Michael J. Flynn. Parallel high-radix nonrestoring division. *IEEE Transactions on Computers*, 42(10), October 1993.

[45] Milos D. Ercegovac and Tomas Lang. Simple radix-4 division with operands scaling. *IEEE Transactions on Computers*, 39(9), September 1990.

[46] Milos D. Ercegovac, Tomas Lang, and Paolo Montuschi. Very-high radix division with prescaling and selection by rounding. *IEEE Transactions on Computers*, 43(8), August 1994.

[47] Peter A. Ivey, Alan L. Cox, John R. Harbridge, and John K. Oldfield. A single-chip public key encryption subsystem. *IEEE Journal of Solid-State Circuits*, 24(4), August 1989.

[48] A. Tomlinson. Modulo multiplier to enhance encryption rates. *Electronic Engineering*, April 1990.

[49] A. Tomlinson. Bit-serial modular multiplier. *Electronics Letters*, 25(24):1664, 23rd November 1989.

[50] Keiichi Iwamura, Tsutomu Matsumoto, and Hideki Imai. High-speed implementation methods for RSA scheme. In *Advances in Cryptology: EUROCRYPT 92*. Springer-Verlag, 1993.

[51] Che Wun Chiou. A fast logic for modular multiplication. *International Journal of Electronics*, 74(6), 1993.

[52] Frank Hoornaert, Marc Decroos, Joos Vandewalle, and Rene Govaerts. Fast RSA-hardware: Dream or reality. Technical report. Cryptech NV/SA, Av. Lloyd George 7, 1050 Brussels, Belgium.

[53] C.W. Chiou and T.C. Yang. Iterative modular multiplication algorithm without magnitude comparison. *Electronics Letters*, 30(24), 24th November 1994.

[54] Ernest F. Brickell. A fast modular multiplication algorithm with application to two-key cryptography. In *Advances in Cryptology: CRYPTO 82*. Springer-Verlag, 1983.

[55] P. W. Baker. Fast computation of A*B modulo N. *Electronics Letters*, 23(15), 16th July 1987.

[56] Naofumi Takagi and Shuzo Yajima. Modular multiplication hardware algorithms with a redundant representation and their application to rsa cryptosystem. *IEEE Transactions on Computers*, 41(7), July 1992.

[57] Naofumi Takagi. A radix-4 modular multiplication hardware algorithm for modular exponentiation. *IEEE Transactions on Computers*, 41(8), August 1992.

[58] Hikaru Morita. A fast modular multiplication algorithm based on a higher radix. In *Advances in Cryptology: CRYPTO 89*. Springer-Verlag, 1990.

[59] Holger Orup, Erik Svendsen, and Erik Andreasen. VICTOR: An efficient RSA hardware implementation. In *Advances in Cryptology: EUROCRYPT 90*. Springer-Verlag, 1991.

[60] Glenn Orton, Lloyd Peppard, and Stafford Tavares. A design of a fast pipelined modular multiplier based on a diminished-radix algorithm. *Journal of Cryptology*, 6:183–208, 1993.

[61] Ernest F. Brickell. A survey of hardware implementations of RSA. In *Advances in Cryptology: CRYPTO 89*. Springer-Verlag, 1990.

[62] Gordon Rankine. THOMAS - a complete single chip RSA device. In *Advances in Cryptology: CRYPTO 86*. Springer-Verlag, 1987.

[63] G. A. Orton, M. P. Roy, P. A. Scott, L. E. Peppard, and S. E. Tavares. VLSI implementation of public-key encryption algorithms. In *Advances in Cryptology: CRYPTO 86*. Springer-Verlag, 1987.

[64] C. K. Koc and C. Y. Hung. Multi-operand modulo addition using carry-save adders. *Electronics Letters*, 26(6):361–363, 15th March 1990.

[65] Peter A. Ivey, Simon N. Walker, Jon M. Stern, and Simon Davidson. An ultra-high speed public-key encryption processor. In *Proceedings of the IEEE Integrated Circuits Conference*, 1992.

[66] Holger Sedlak. The RSA cryptography processor. In *Advances in Cryptology: CRYPTO 87*. Springer-Verlag, 1988.

[67] Martin Kochanski. Developing an RSA chip. In *Advances in Cryptology: CRYPTO 85*. Springer-Verlag, 1986.

[68] B.S. Prasanna and P.V. Ananda Mohan. Fast VLSI architectures using nonredundant multibit recoding for computing $a^y \pmod{N}$. *IEE Proceedings: Circuits Devices and Systems*, 141(5), October 1994.

[69] Giuseppe Alia and Enrico Martinelli. A VLSI modulo m multiplier. *IEEE Transactions on Computers*, 40(7), July 1991.

[70] Stanislaw J. Piestrak. Design of residue generators and multioperand modulo adders using carry-save adders. *IEEE Transactions on Computers*, 43(1), January 1994.

[71] Thomas Beth and Dieter Gollmann. Algorithm engineering for public key algorithms. *IEEE Journal on Selected Areas in Communications*, 7(4), May 1989.

[72] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170), April 1985.

[73] S. J. Shepherd. A high-speed cryptographic engine. Electrical Engineering Department, University of Bradford, UK.

[74] Dan Zuras. More on squaring and multiplying large integers. *IEEE Transactions on Computers*, 43(8), August 1994.

[75] M. Shand, P. Bertin, and J. Vuillemin. Hardware speedups in long integer multiplication. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, July 1990.

[76] Shimon Even. Systolic modular multiplication. In *Advances in Cryptology: CRYPTO 90*. Springer-Verlag, 1991.

[77] Jorg Sauerbrey. A modular exponentiation unit based on systolic arrays. In *Advances in Cryptology: AUSCRYPT 92*. Springer-Verlag, 1993.

[78] Keiichi Iwamura, Tsutomu Matsumoto, and Hideki Imai. Systolic-arrays for modular exponentiation using montgomery method. In *Advances in Cryptology: EUROCRYPT 92*. Springer-Verlag, 1993.

[79] M. Shand and J. Vuillemin. Fast implementations of RSA cryptography. In *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, 1993.

[80] Colin D. Walter. Systolic modular multiplication. *IEEE Transactions on Computers*, 42(3), March 1993.

[81] Stephen E. Eldridge and Colin D. Walter. Hardware implementation of montgomery's modular multiplication algorithm. *IEEE Transactions on Computers*, 42(6), June 1993.

[82] C.D. Walter. Still faster modular multiplication. *Electronics Letters*, 31(4), 16th February 1995.

[83] *CMOS Semi-Custom CLA70000 ASIC Handbook.* GEC Plessey Semiconductors, July 1992.

[84] GEC Plessey Semiconductors. *Mentor Design Kit: User Manual*, November 1991.

[85] GEC Plessey Semiconductors. *Mentor Design Kit: Volume One*, November 1991.

[86] GEC Plessey Semiconductors. *Mentor Design Kit: Volume Two*, November 1991.

[87] Lewis C. Eggebrecht. *Interfacing to the IBM Personal Computer.* Howard W. Sams and Company, 1983.

[88] Uyless Black. *The X Series Recommendations.* McGraw-Hill, 1995.

[89] Michael O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12:128–138, 1980.

[90] S. J. Shepherd, P. W. Sanders, and A. Patel. A comprehensive security system - the concepts, agents and protocols. *Computers and Security*, 9:631–643, 1990.

[91] Warwick Ford and Brian O'Higgins. Public-key cryptography and open systems interconnection. *IEEE Communications Magazine*, July 1992.

[92] William Stallings. *Data and Computer Communications.* MacMillan Publishing Company, 1991.

[93] Jurjen Bos and Matthijs Coster. Addition chain heuristics. In *Advances in Cryptology: CRYPTO 89.* Springer-Verlag, 1990.

[94] Y. Yacobi. Exponentiating faster with addition chains. In *Advances in Cryptology: EUROCRYPT 90*. Springer-Verlag, 1991.

[95] Jorg Sauerbrey and Andreas Dietel. Resource requirements for the application of addition chains in modulo exponentiation. In *Advances in Cryptology: EUROCRYPT 92*. Springer-Verlag, 1993.

[96] L.C.K. Hui and K.Y. Lam. Fast square-and-multiply exponentiation for RSA. *Electronics Letters*, 30(17), 18th August 1994.

[97] K.Y. Lam and L.C.K. Hui. Efficiency of SS(l) square-and-multiply exponentiation algorithms. *Electronics Letters*, 30(25), 8th December 1994.

[98] MIPS R10000 microprocessor product overview. Technical report, MIPS Technologies Incorporated, October 1994.

[99] Digital 21064-AA microprocessor product brief. Technical report, Digital Equipment Corporation, February 1992.

[100] Charles R. Moore. PowerPC 601 microprocessor. Technical report, IBM Corporation, 1993.

[101] James Kahle and Deene Ogden. PowerPC 603 microprocessor. Technical report, IBM Corporation, 1994.

[102] Mahdi Abdelguerfi, Andrea Dunham, and Wayne Patterson. MRA: A computational technique for security in high-performance systems. *Computer Security*, A-37:401–417, 1993.

[103] K. C. Posch and R. Posch. Residue number systems: A key to parallelism in public-key cryptography. In *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, 1-4 December 1992.

[104] Nicholas S. Szabo and Richard I. Tanaka. *Residue Arithmetic and its Applications to Computer Technology*. McGraw-Hill Book Company, 1967.

Appendix A

The WHiSpER SMC

The SMC is a state-machine that controls the operation of the WHiSpER chip. It is composed of four semi-autonomous smaller state-machines called LSM, TSM, ESM and RSM. Figures A.1, A.2, A.3 and A.4 show state-transition diagrams for each of these state-machines. See the schematics of Appendix B for their circuit diagrams.

A.1 SMC Input Signals

The following signals are all active high and control the operation of the SMC.

- cLDK: LDK command signal from ComStat.
- cEXP: EXP command signal from ComStat. Perform first exponentiation in sequence.
- cEXP_RDC: EXP_RDC command signal from ComStat. Perform intermediate exponentiation and reduction in sequence.
- cRDC: RDC command signal from ComStat. Perform final reduction at end of sequence.
- e_i: current exponent bit indexed by EC counter. Used by ESM during the exponentiation process.

- ECQ00: active when EC counter is at 0x000. Used by ESM during the exponentiation process.

- RCQ3F: active when RC counter is at 0x3F. Used by the RSM during the reduction process.

- Upos: active when currently accessed URAM byte has most-significant-bit of '0'. Used by RSM during the reduction process.

A.2 SMC Output Signals

The control signals produced by the SMC are all active high and have the following functions.

- SRI: the SRAM is being used internally by the WHiSpER chip. Switch SRAM address, data and control lines to internally generated signals. No host microprocessor access to the SRAM is allowed.

- SRT: the SRAM is being used by the transfer process. Switch SRAM address lines to the transfer counter.

- URI: the URAM is being used internally by the WHiSpER chip. No host microprocessor access allowed.

- URT: the URAM is being used by the transfer process. Switch URAM address lines to the transfer counter.

- RCA: the SRAM and URAM are being used by the reduction process. Switch SRAM and URAM address lines to the reduction counter.

- IntSet: set the $\overline{\text{INT}}$ to active low.

- EMSBload: load the EMSB register.

- NINload: load the NIN register.
- ECen: enable EC counter operation.
- ECload: load the EC counter with preset value 0x1FF.
- TCload: load the TC counter with preset value 0x7F.
- Txfer_H: transfer H from SRAM.
- Txfer_N: transfer N from SRAM.
- Txfer_HN: active whenever transfer operation is in progress (Txfer_H OR Txfer_N).
- TxferClkEn: enable TxferClk.
- RCload: load RC counter with preset value 0x3F.
- RedcClkEn: enable RedcClk.
- ESM_XclkEn: enable MME X register clock.
- ESM_YclkEn: enable MME Y register clock.
- ESM_AclkEn: enable MME accumulator clock.
- ESM_BclkEn: enable MME B register clock.
- ESM_Xpar: enable parallel loading of MME X register.
- ESM_Ypar: enable parallel loading of MME Y register.
- ESM_Asrst: enable synchronous reset of MME accumulator.
- ESM_Xone: override MME $x(i)$ generation circuitry to generate $X = 1$.
- ESM_Xs2X: sign extend the MME X register as it is consumed during a multiplication.
- ESM_X2X: refill MME X register as it is consumed during a multiplication.

- ESM_X2B: fill MME B register from X register as X register is consumed during a multiplication.

- ESM_B2X: fill MME X register from B register as X register is consumed during a multiplication.

A.3 SMC Internal Signals

The following signals are all active high and are used internally by the SMC for output signal generation, for LSM, TSM, ESM and RSM inter-state-machine communications and for iC, jC and kC counter control.

- SRIset, SRIrst: Set SRI = 1 or 0 respectively.

- SRTset, SRTrst: Set SRT = 1 or 0 respectively.

- URIset, URIrst: Set URI = 1 or 0 respectively.

- URTset, URTrst: Set URT = 1 or 0 respectively.

- RCAset, RCArst: Set RCA = 1 or 0 respectively.

- ESM_Go: Issued by the TSM to start the ESM.

- EXP_Done: Issued by the ESM to inform the TSM of its completion.

- Reduce: Issued by the TSM to start the RSM reduction process.

- NoReduce: Issued by the TSM so that the RSM just issues an interrupt but does not perform a reduction.

- RDC_Done: Issued by the RSM to inform the TSM of its completion.

- RSM_Go: Issued by the ESM to enable RSM operation.

- RSM_Stop: Issued by the ESM to temporarily halt RSM operation.

- i4, i64, i512: iC count signals.
- j4, jRAAD: jC count signals. jRAAD is programmable via RAADsel(1:0) device inputs.
- k4, k64: kC count signals.

A.4 State-Transition Diagrams

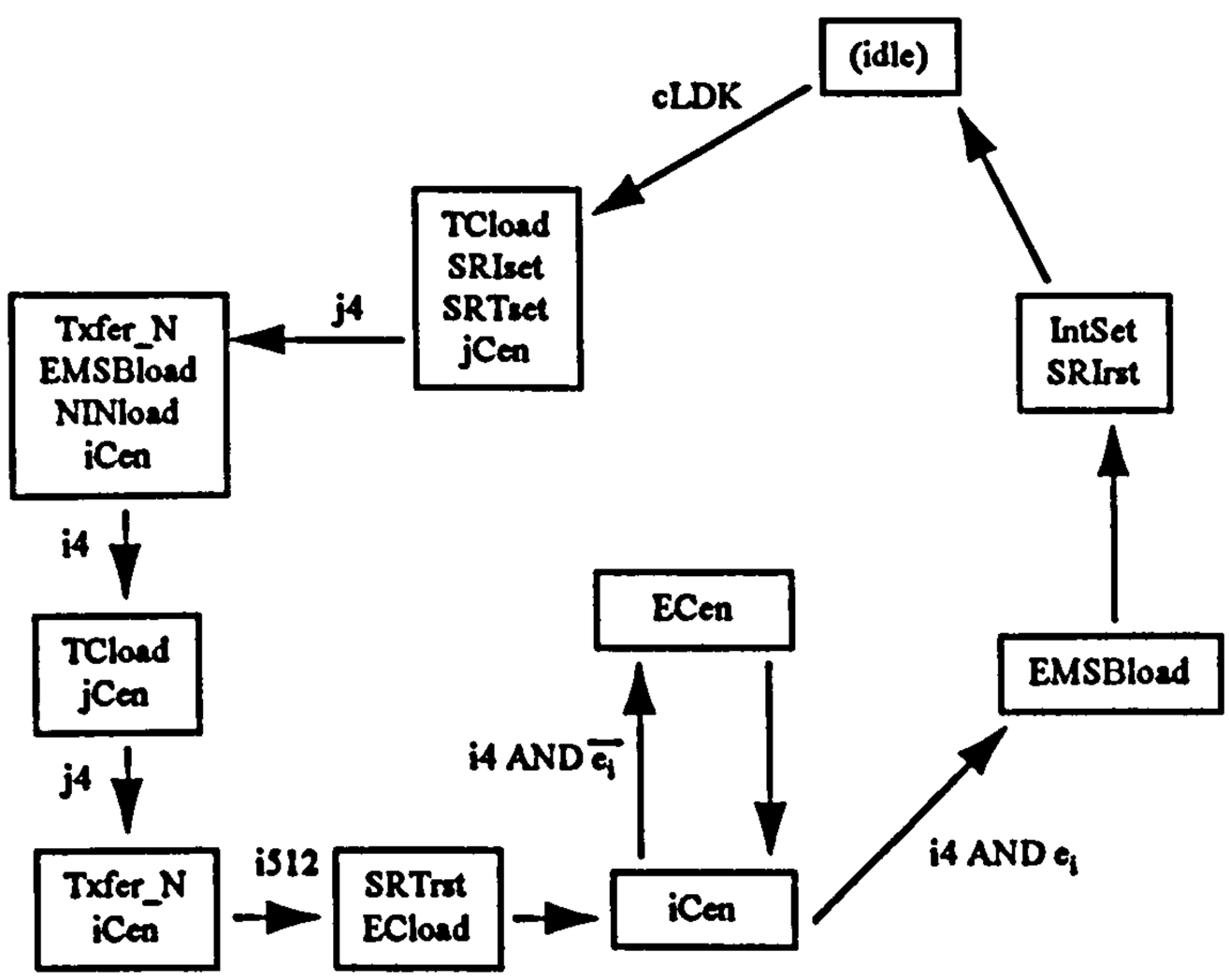


Figure A.1: LSM state-transition diagram.

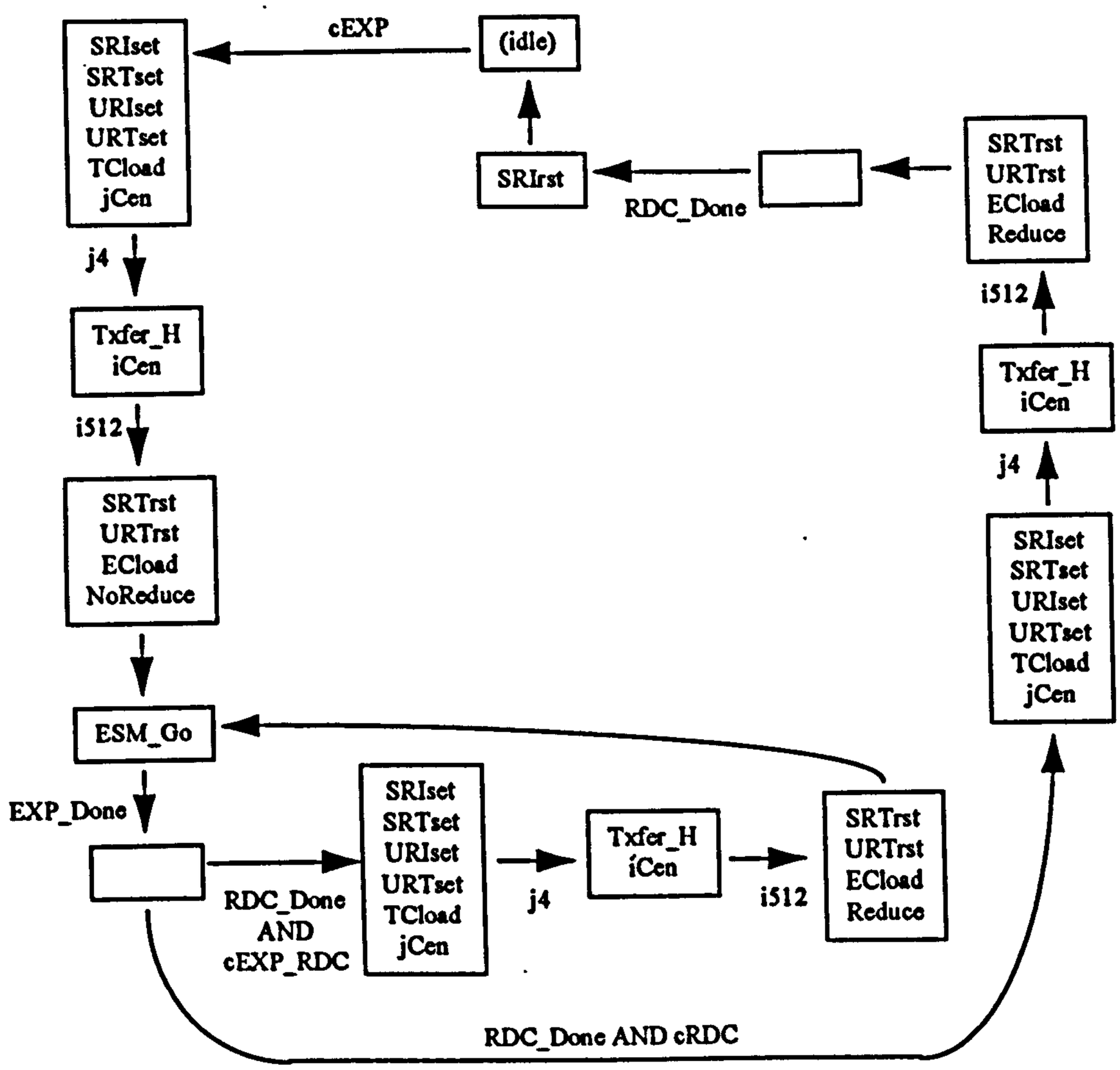


Figure A.2: TSM state-transition diagram.

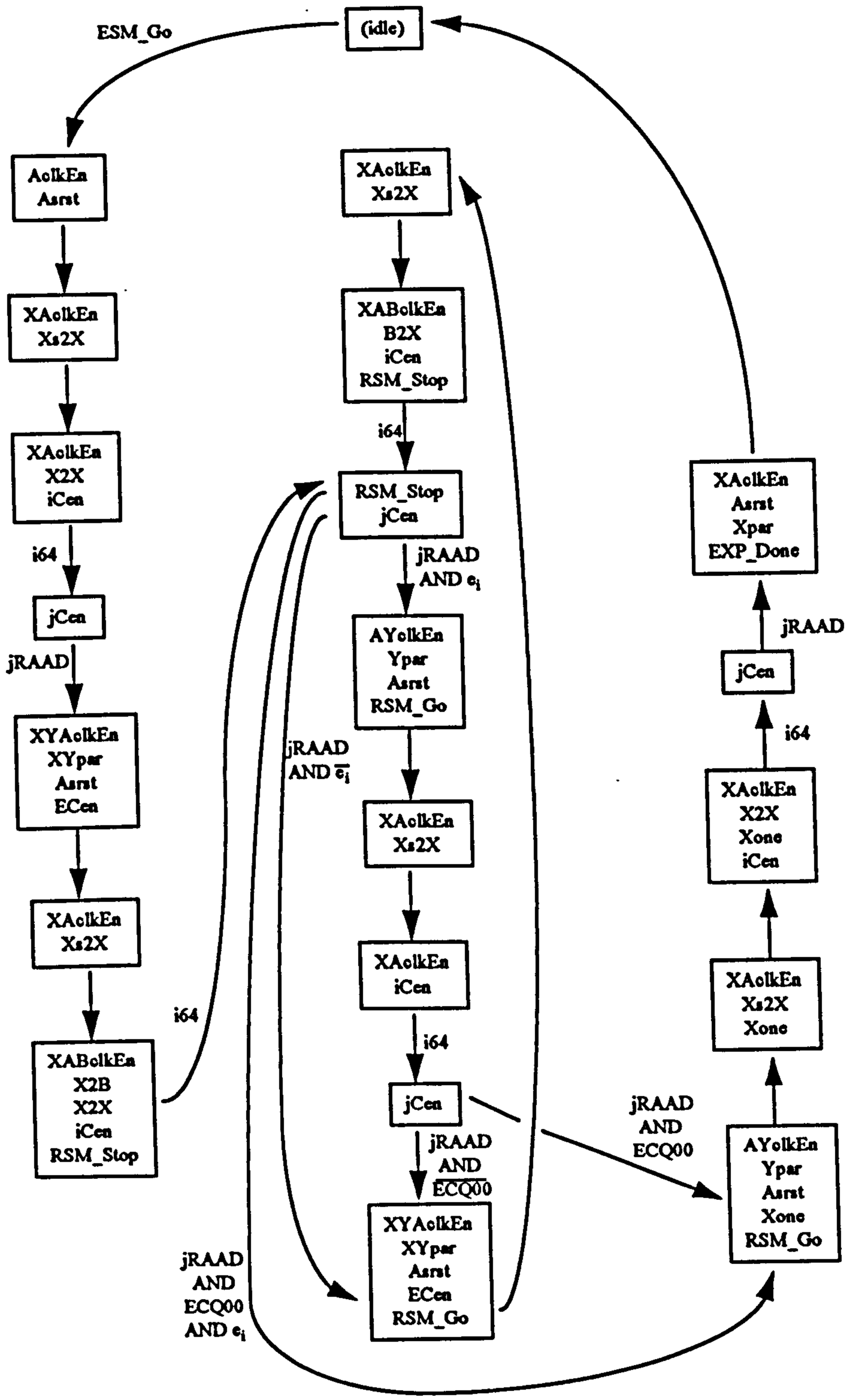


Figure A.3: ESM state-transition diagram.

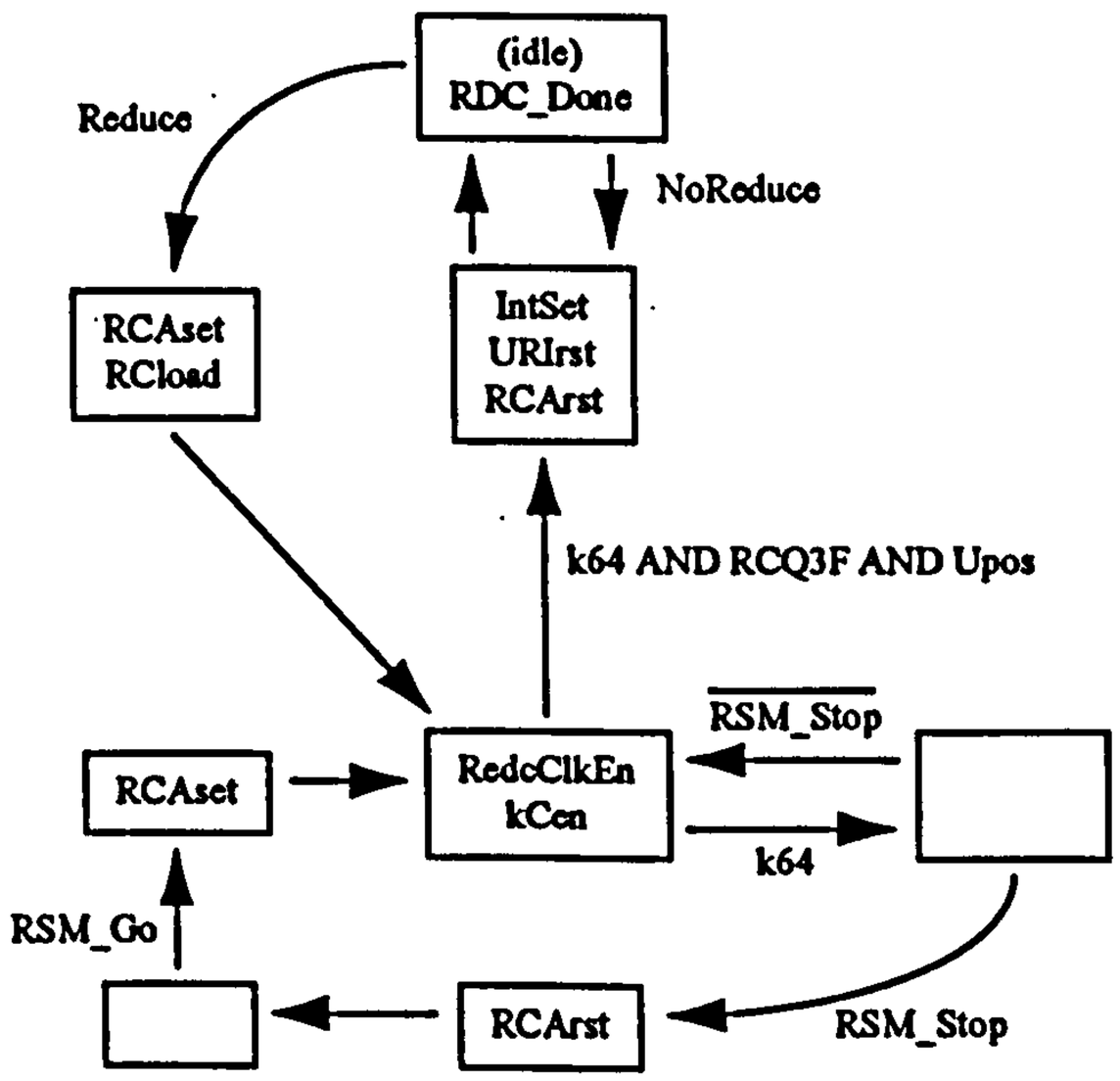
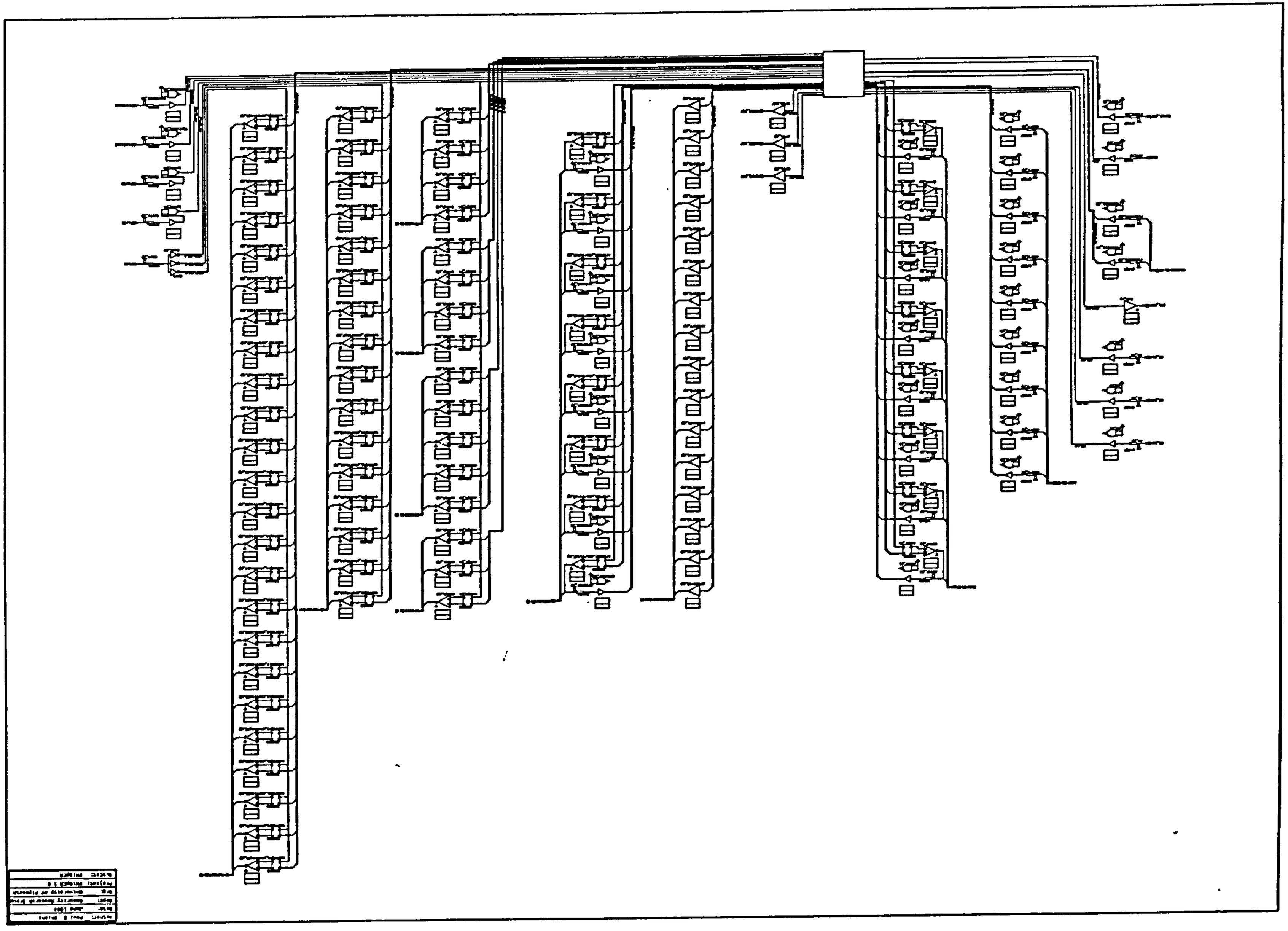


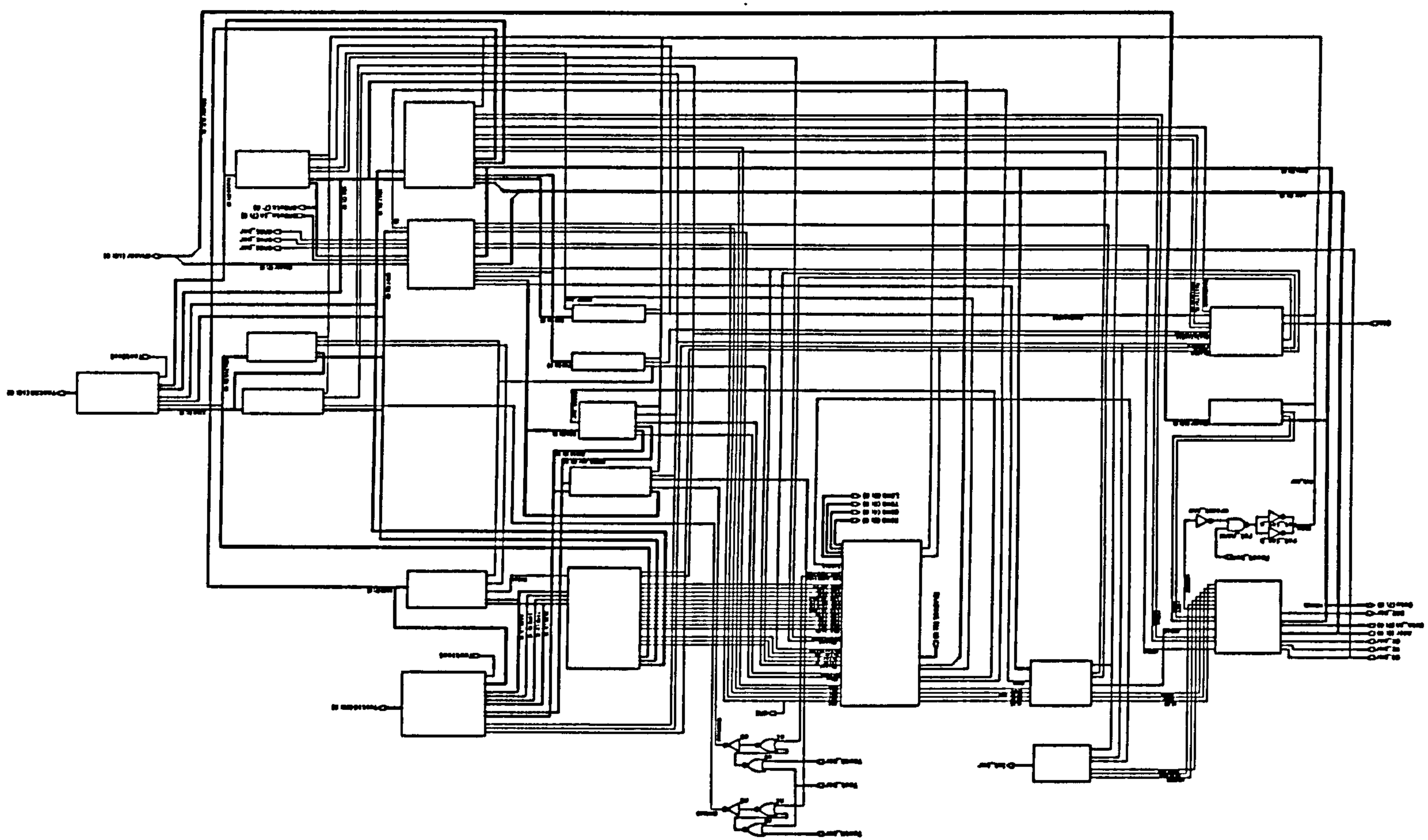
Figure A.4: RSM state-transition diagram.

Appendix B

The WHiSpER Schematics

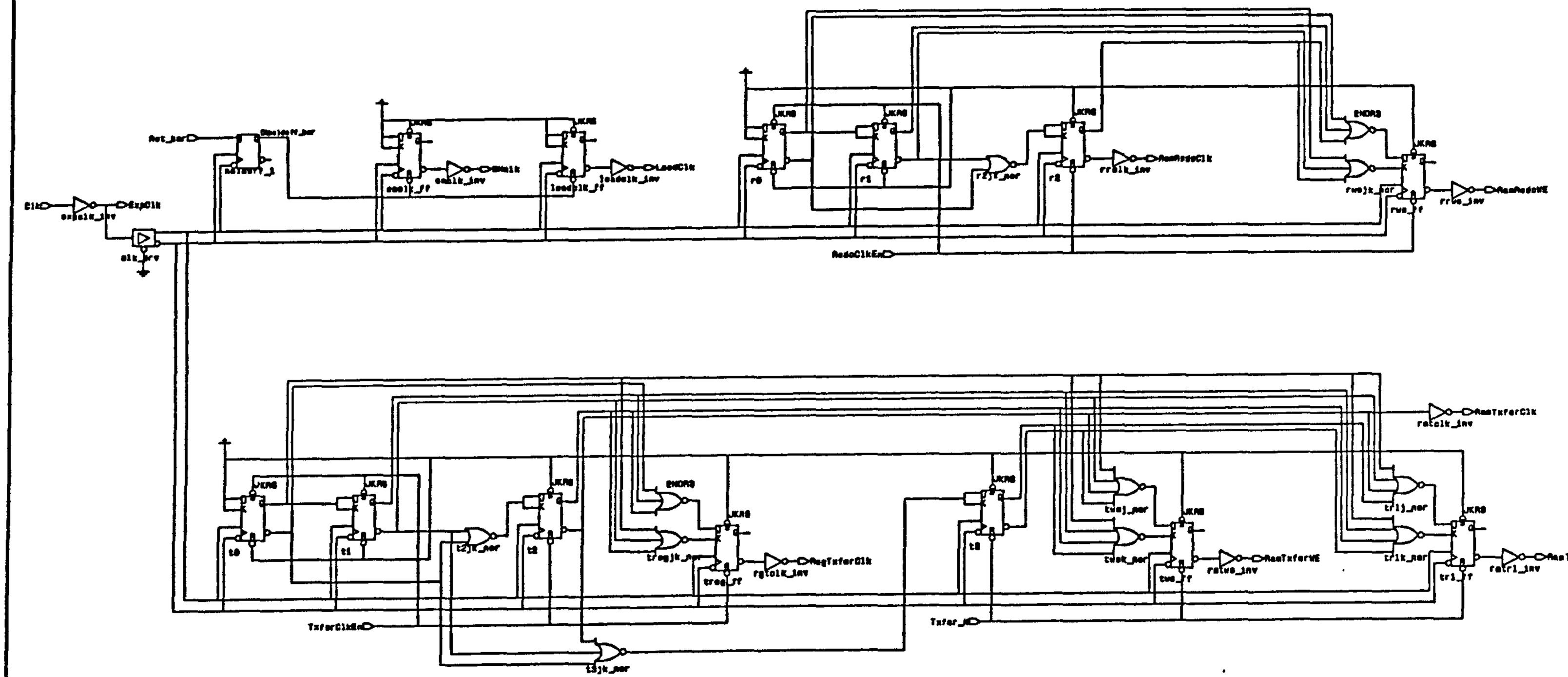
The following pages contain full circuit diagrams for the WHiSpER chip.

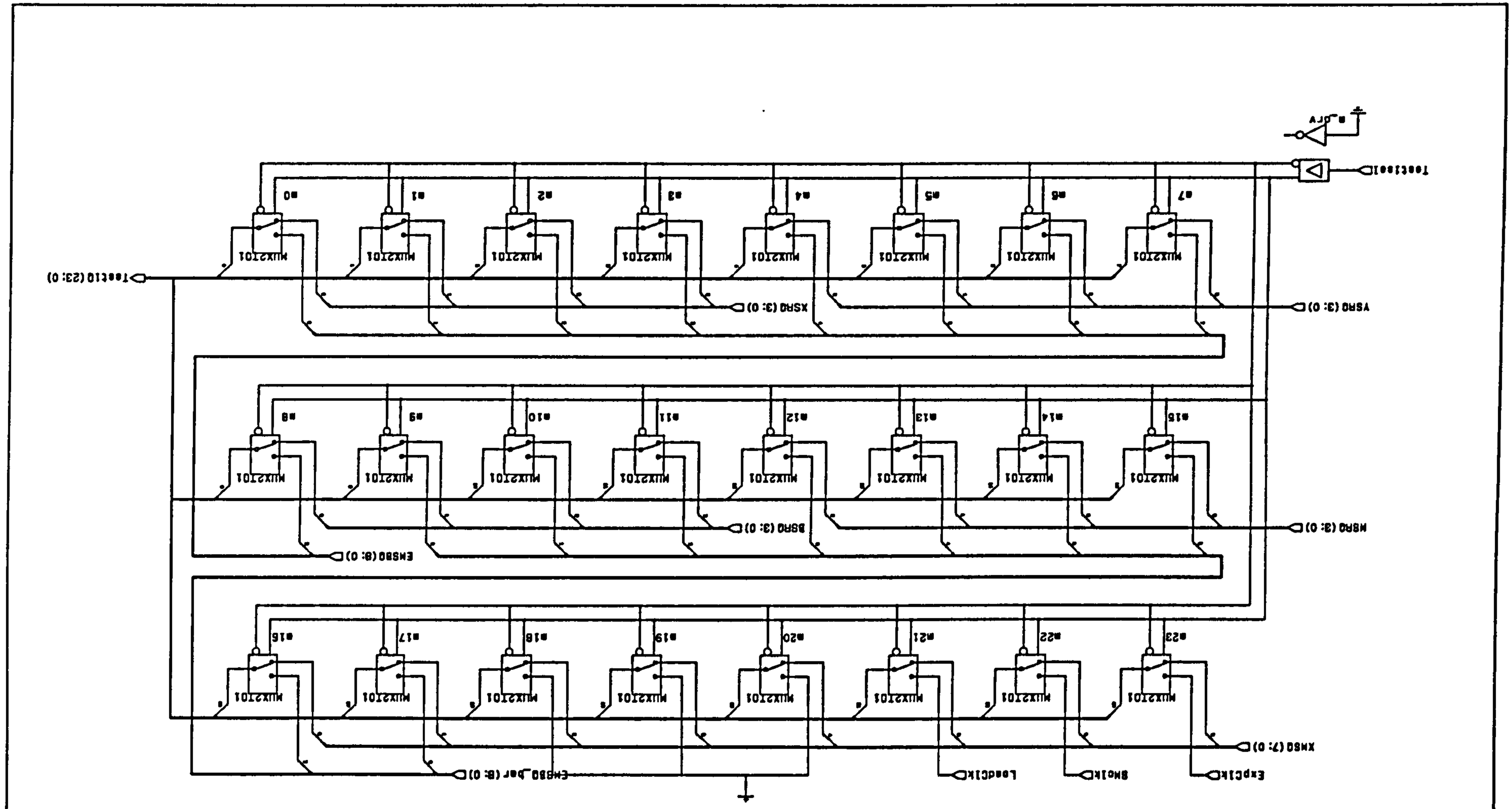




Author:	Paul B. Delone
Date:	June 1994
Dept:	Security Assessment Group
Org:	University of Plymouth
Project:	WHISPER 1.0
Subject:	Core

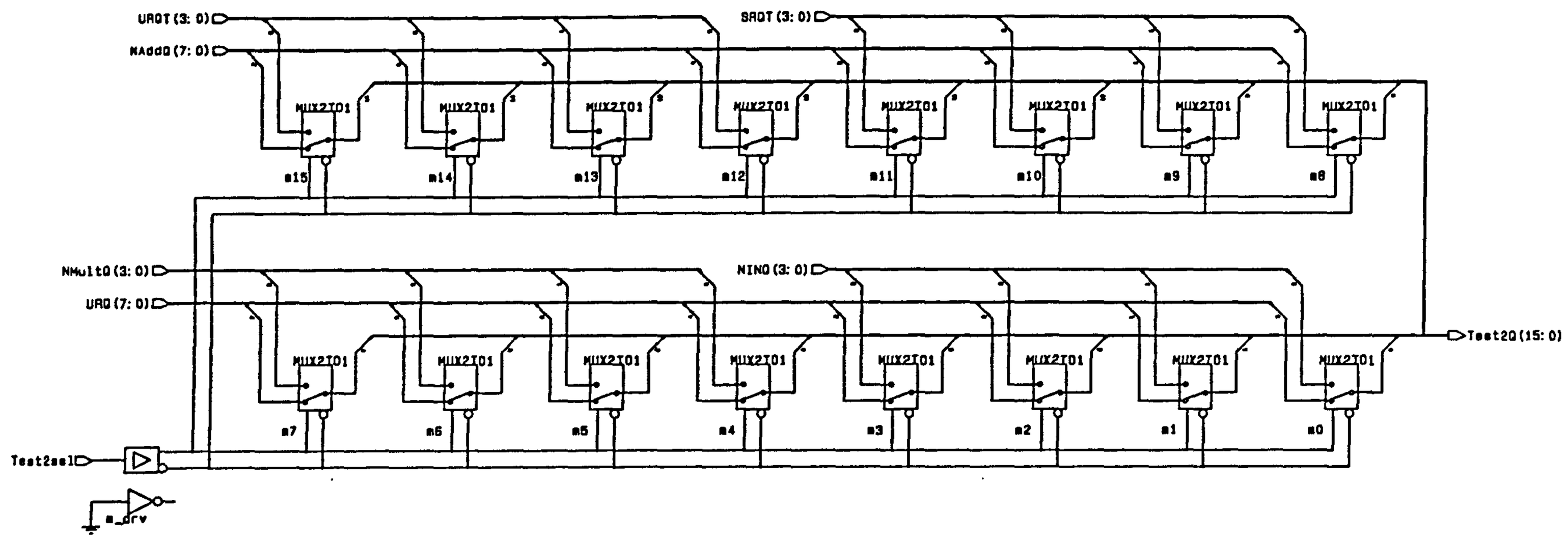
Author:	Paul D. Onione
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHISPER 1.0
SubCct:	ClkGen



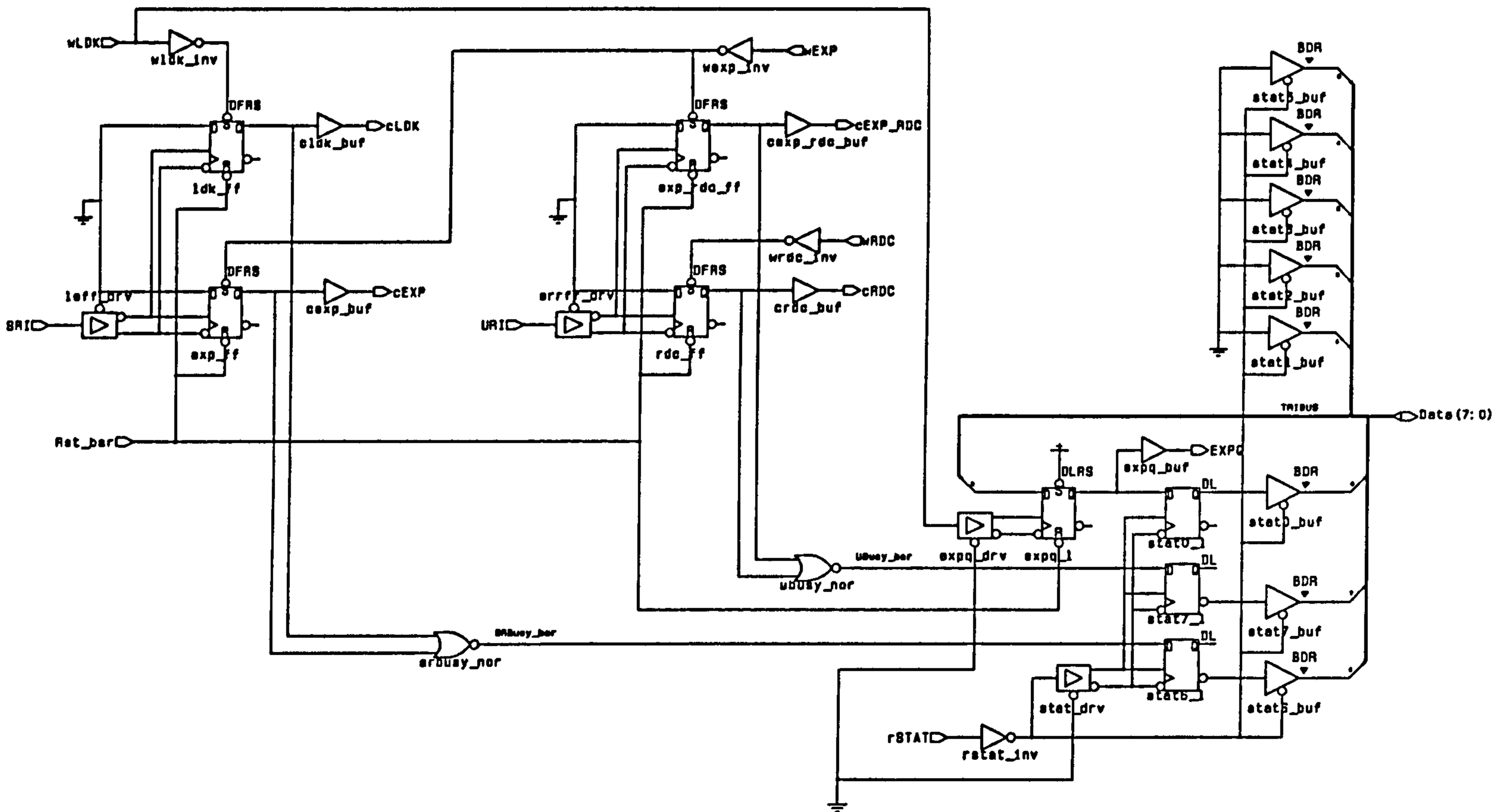


Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHISPER 1.0
SubCct:	TestOP1

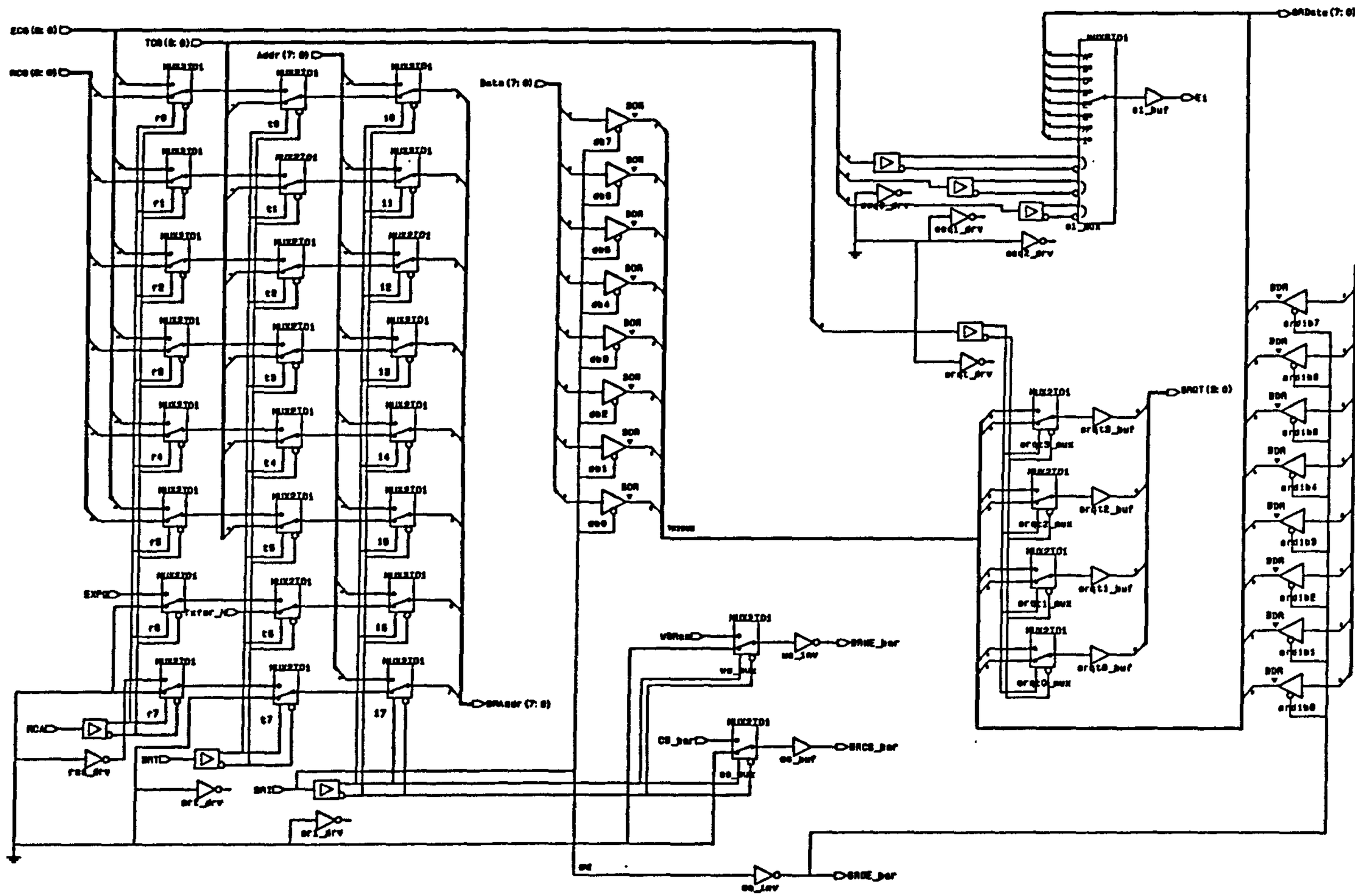
Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHiSpER 1.0
SubCct:	TestOP2



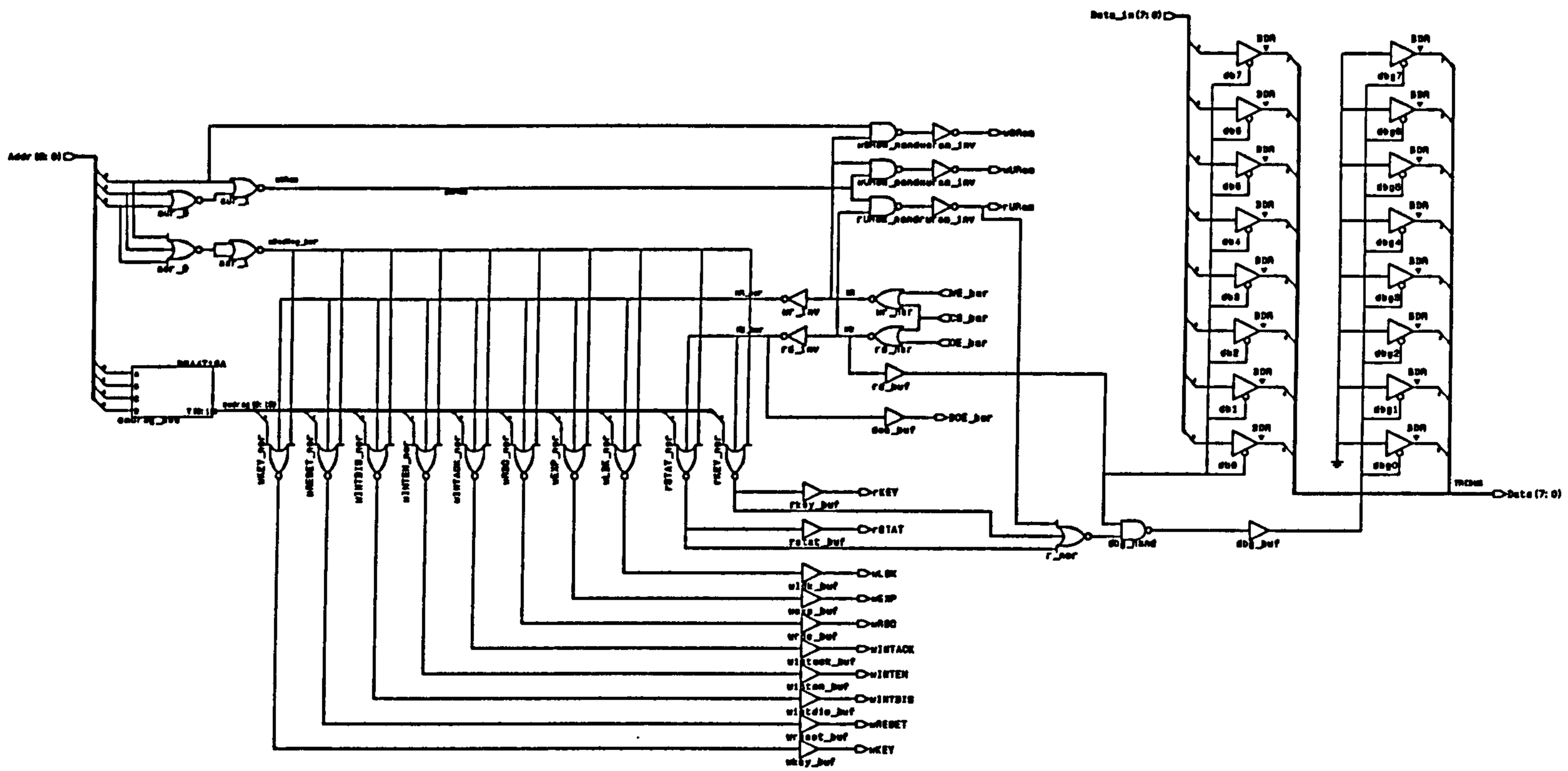
Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHiSpER 1.0
SubCct:	ComStat



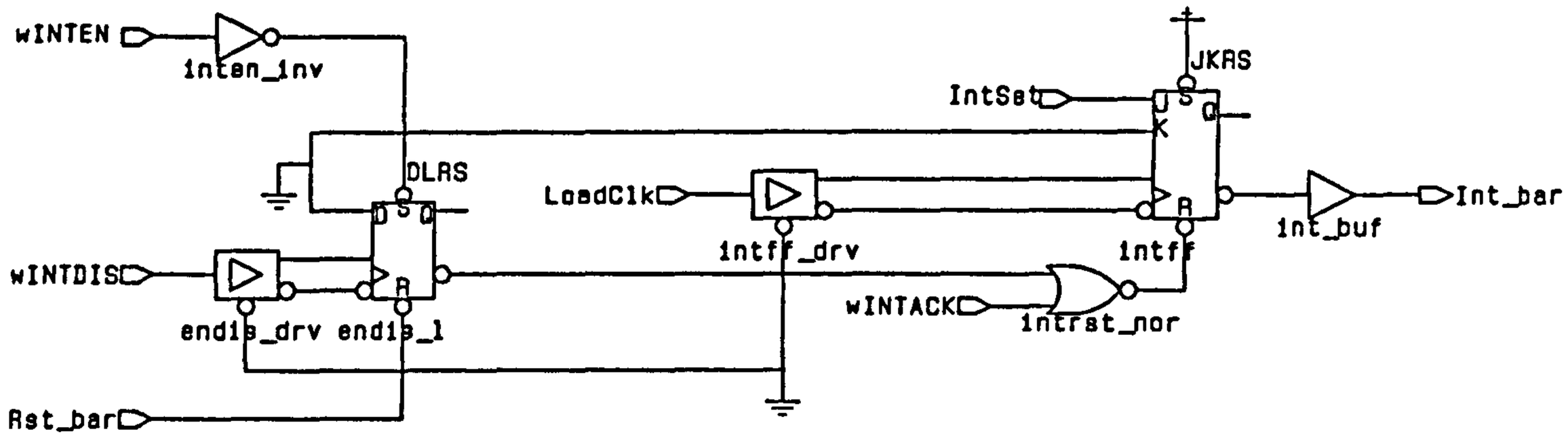
Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHISpER 1.0
SubCct:	SRI



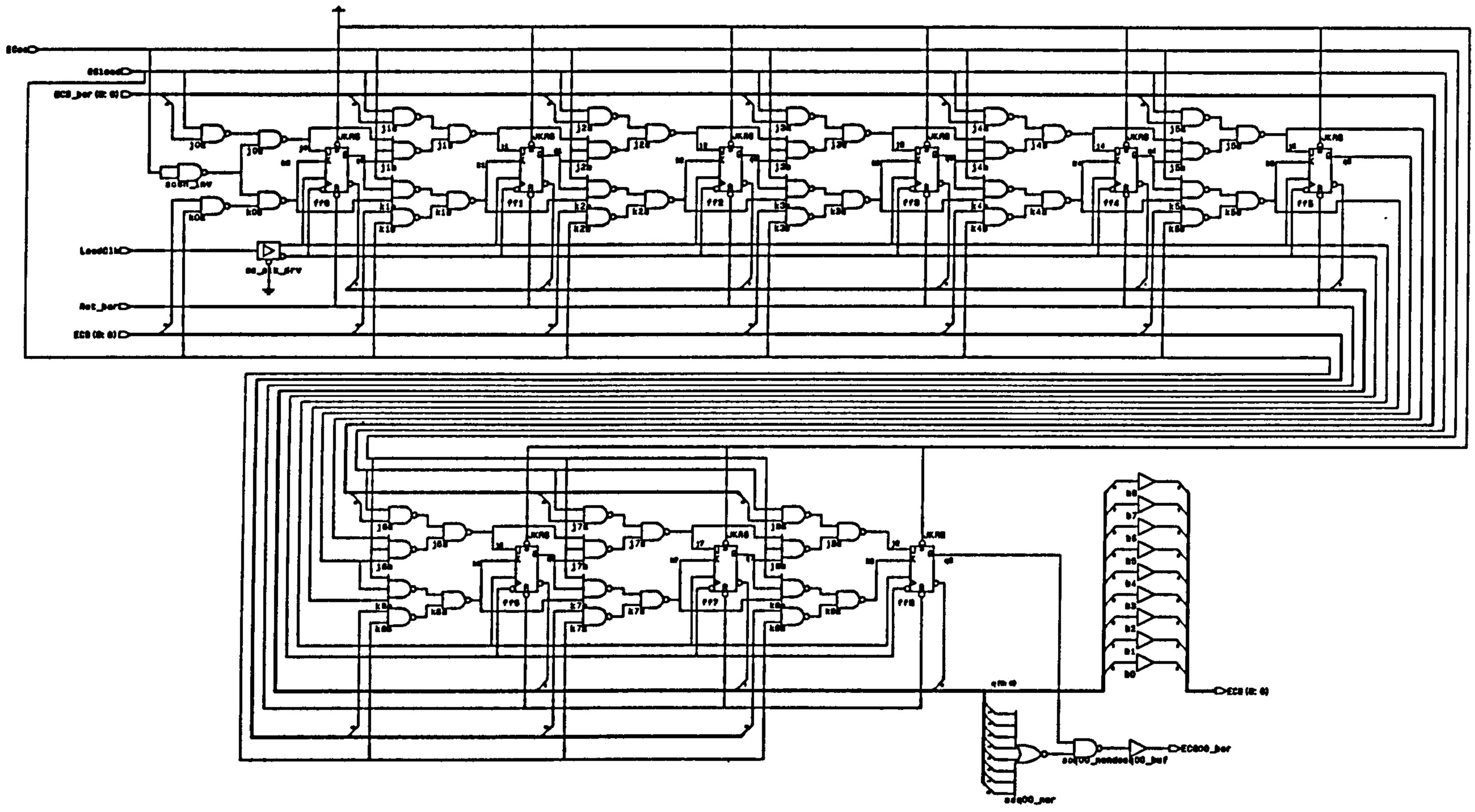
Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHISPER 1.0
SubCct:	MPI



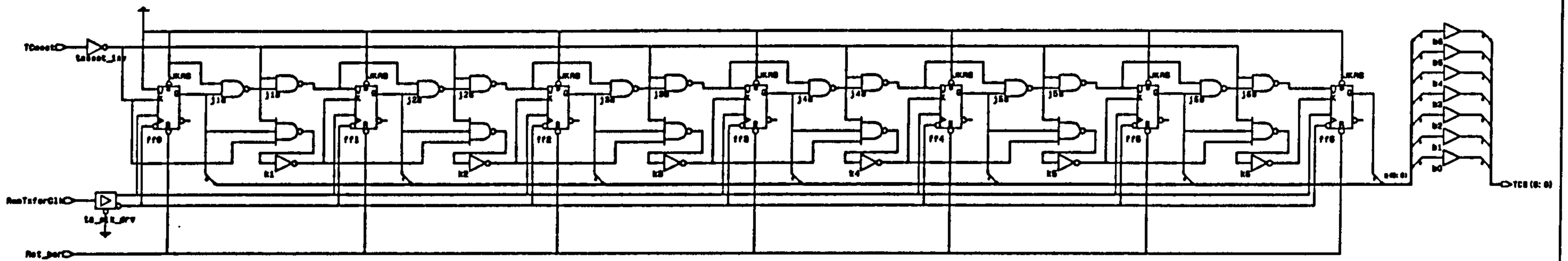
Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WH1SpER 1.0
SubCct:	IntCtrl



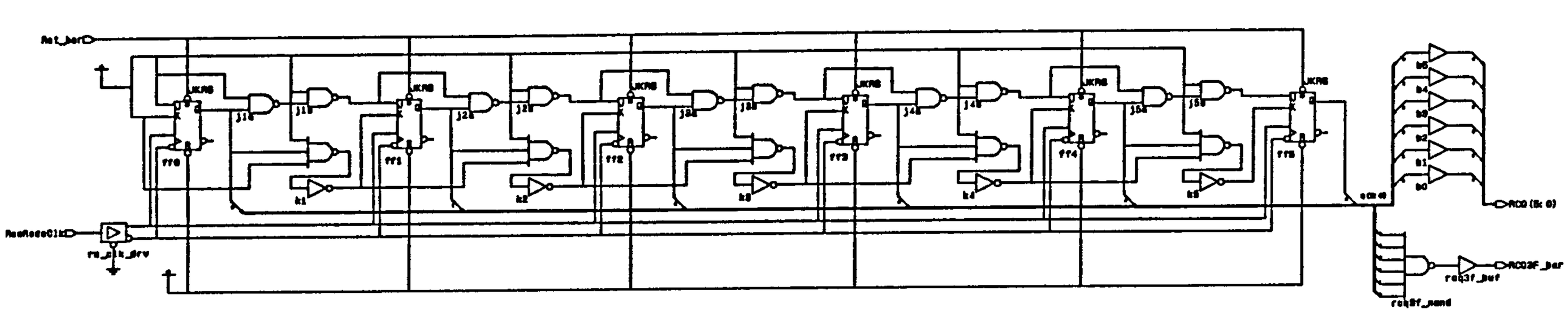
Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	Univereity of Plymouth
Project:	WHiSpER 1.0
SubCot:	EC

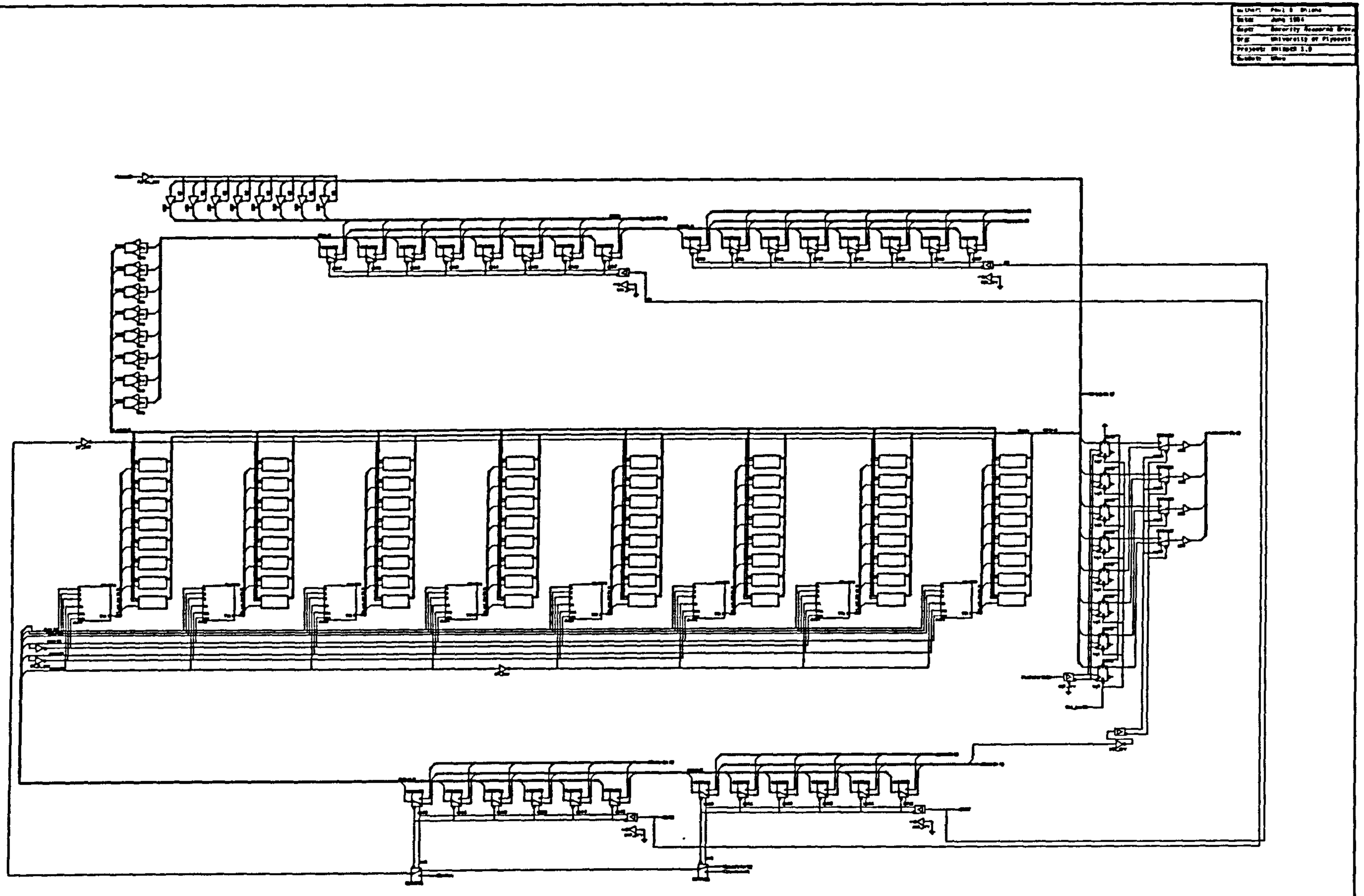


Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHiSpER 1.0
SubCat:	TC

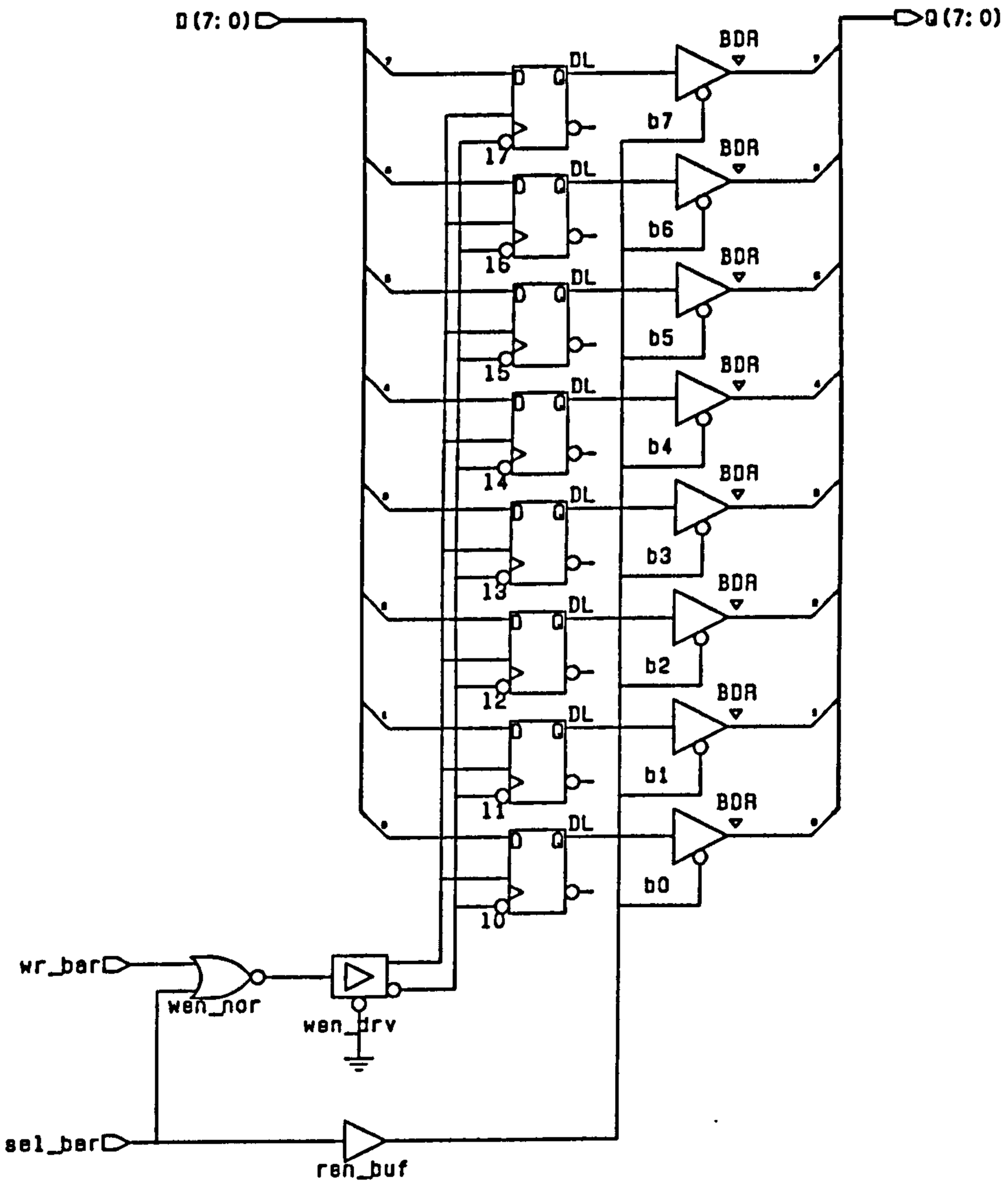


Author:	Paul D. Onione
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHISPER 1.0
SubCct:	RC

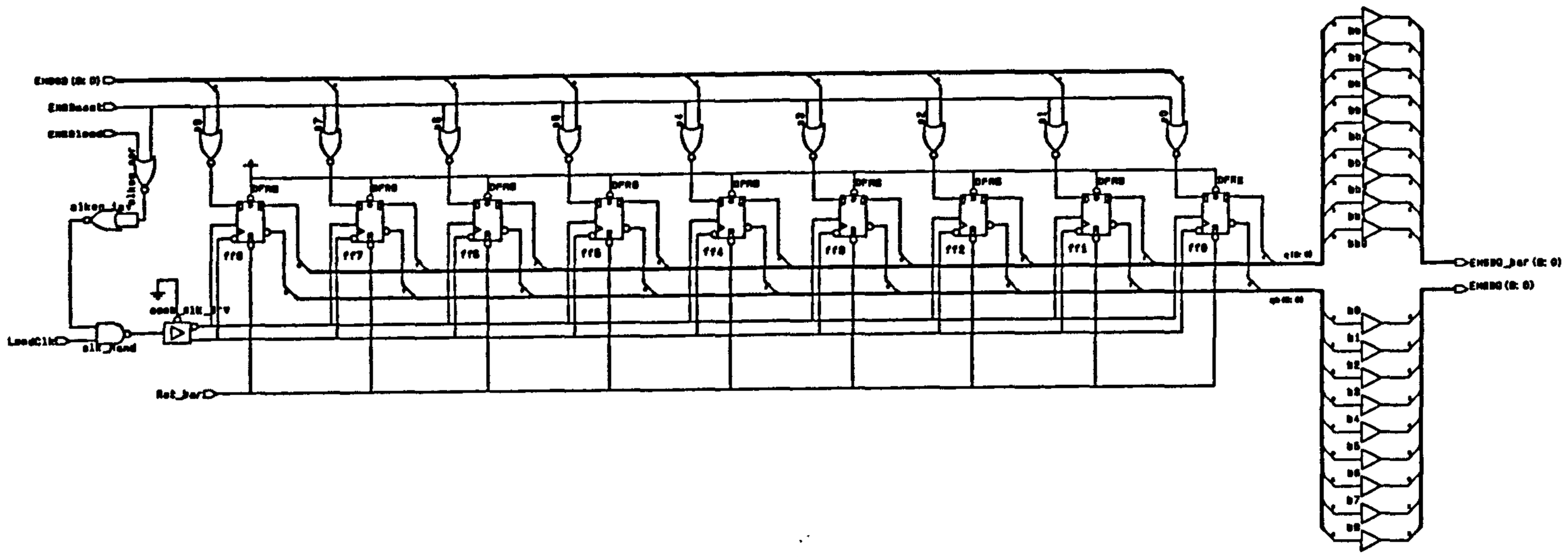




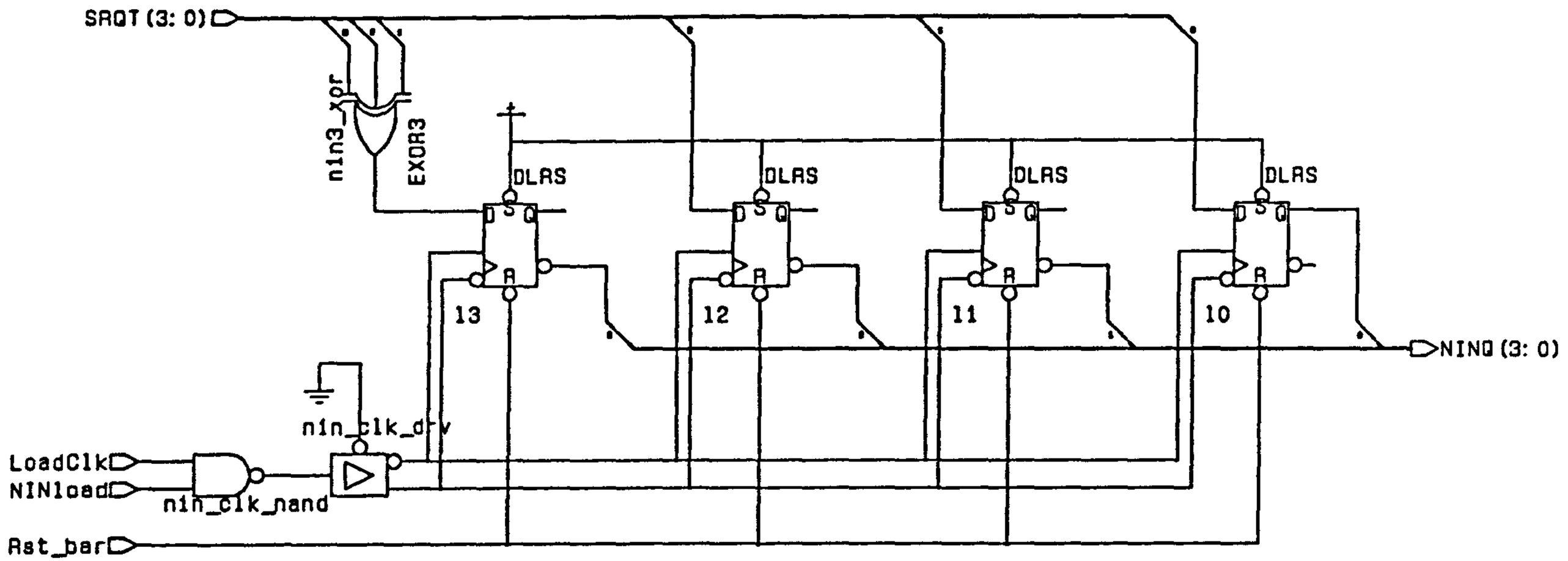
Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WH1SpER 1.0
SubCct:	URamReg



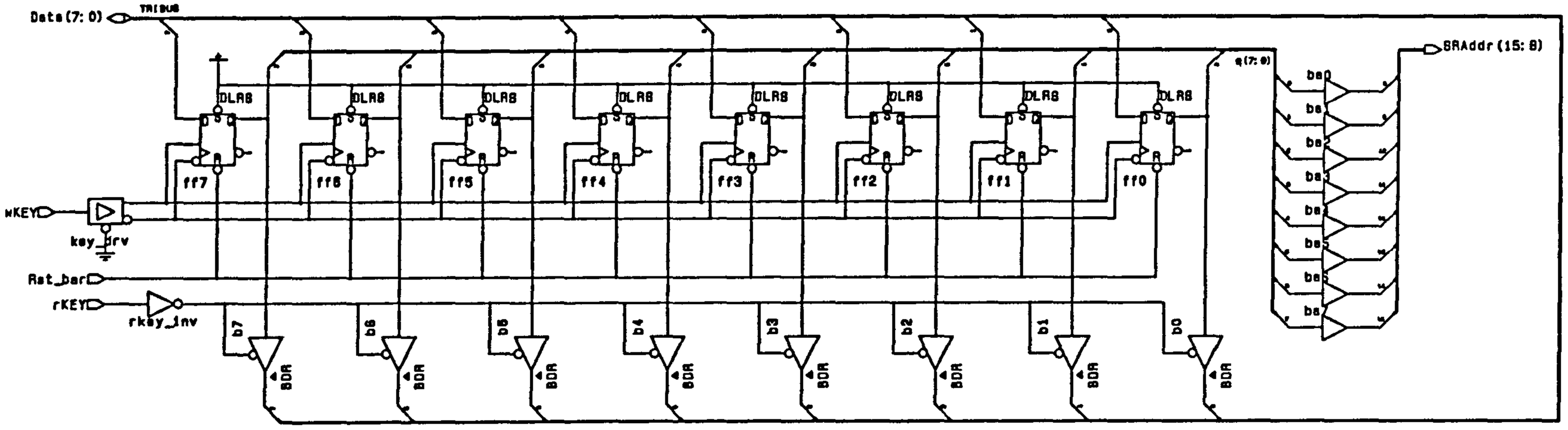
Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHISPER 1.0
SubCct:	EMSB



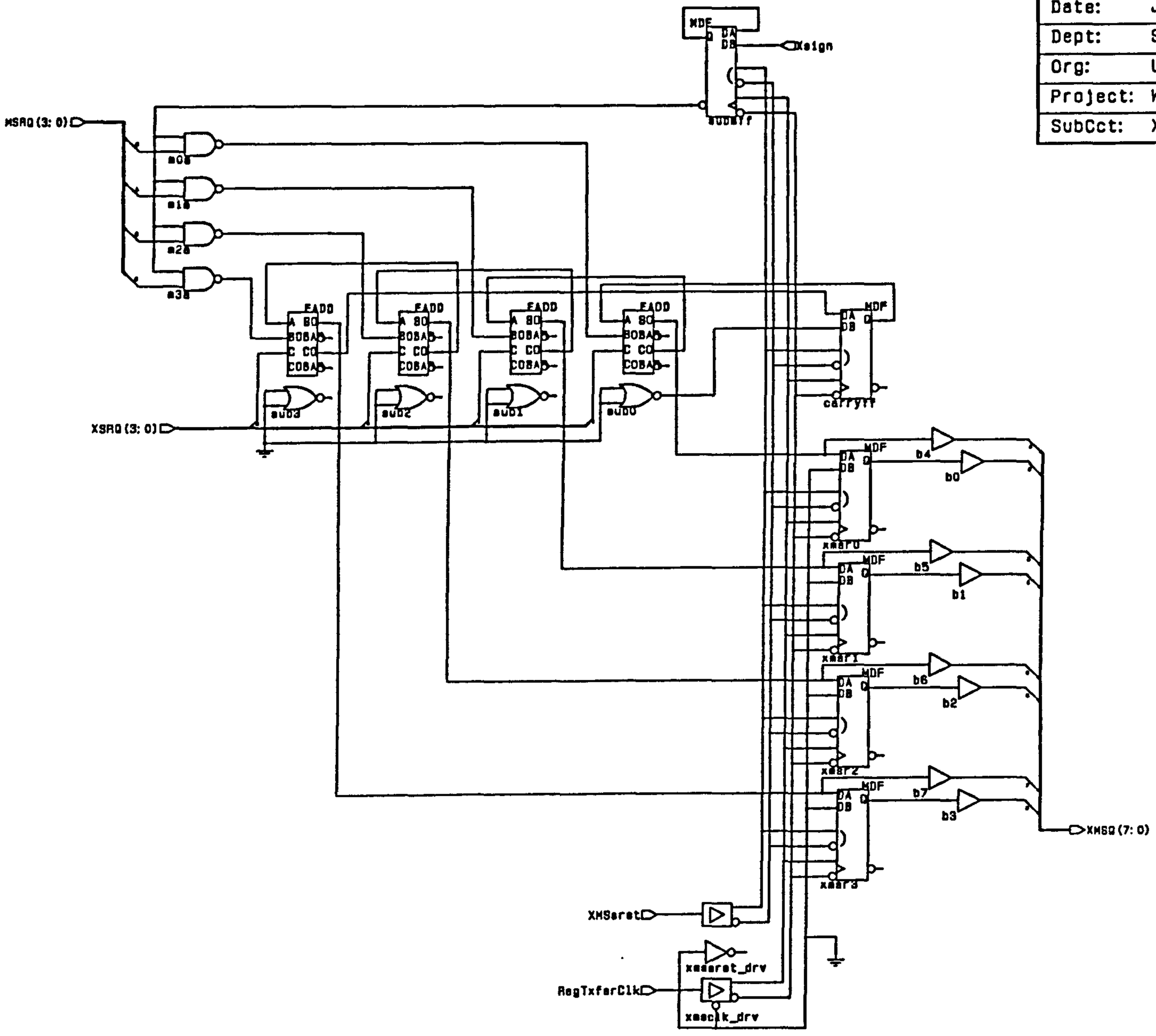
Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHiSpER 1.0
SubCct:	NIN



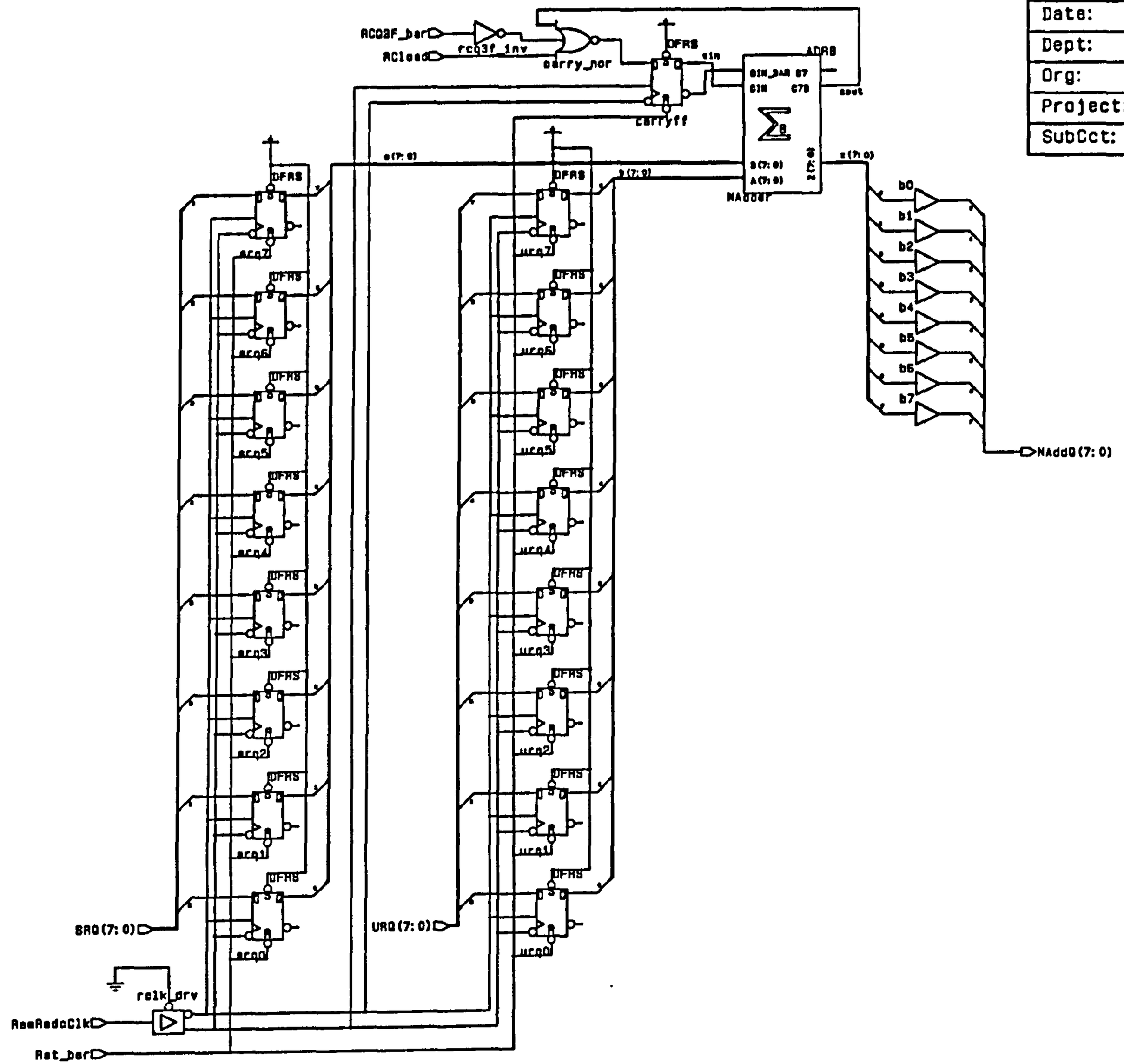
Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHiSpER 1.0
SubCct:	KEY



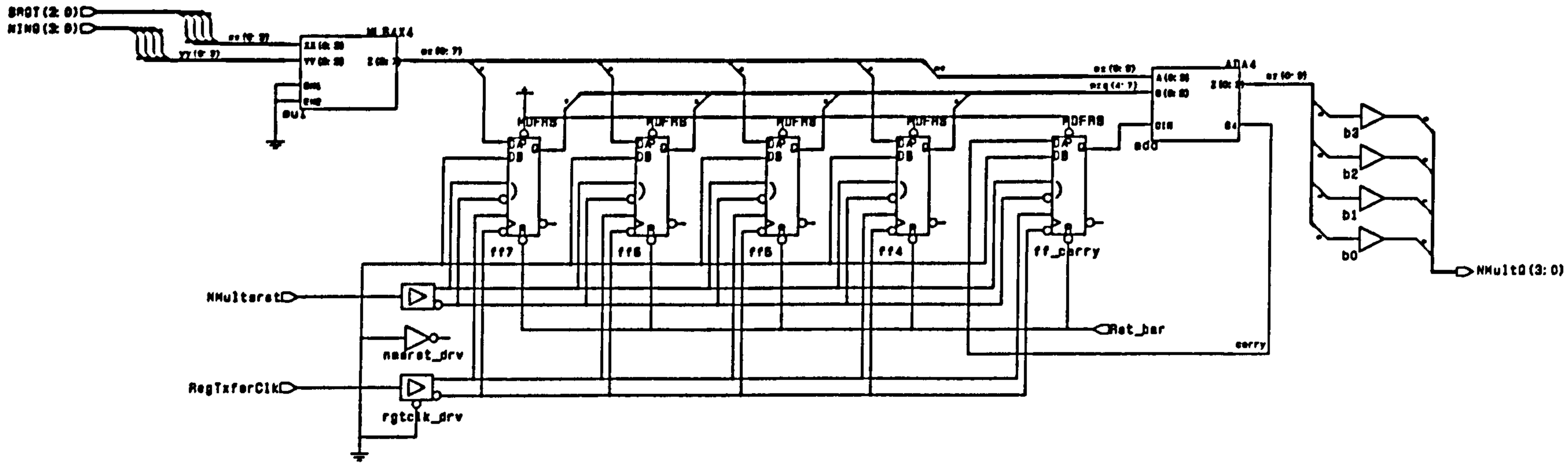
Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHiSpER 1.0
SubCct:	XMS



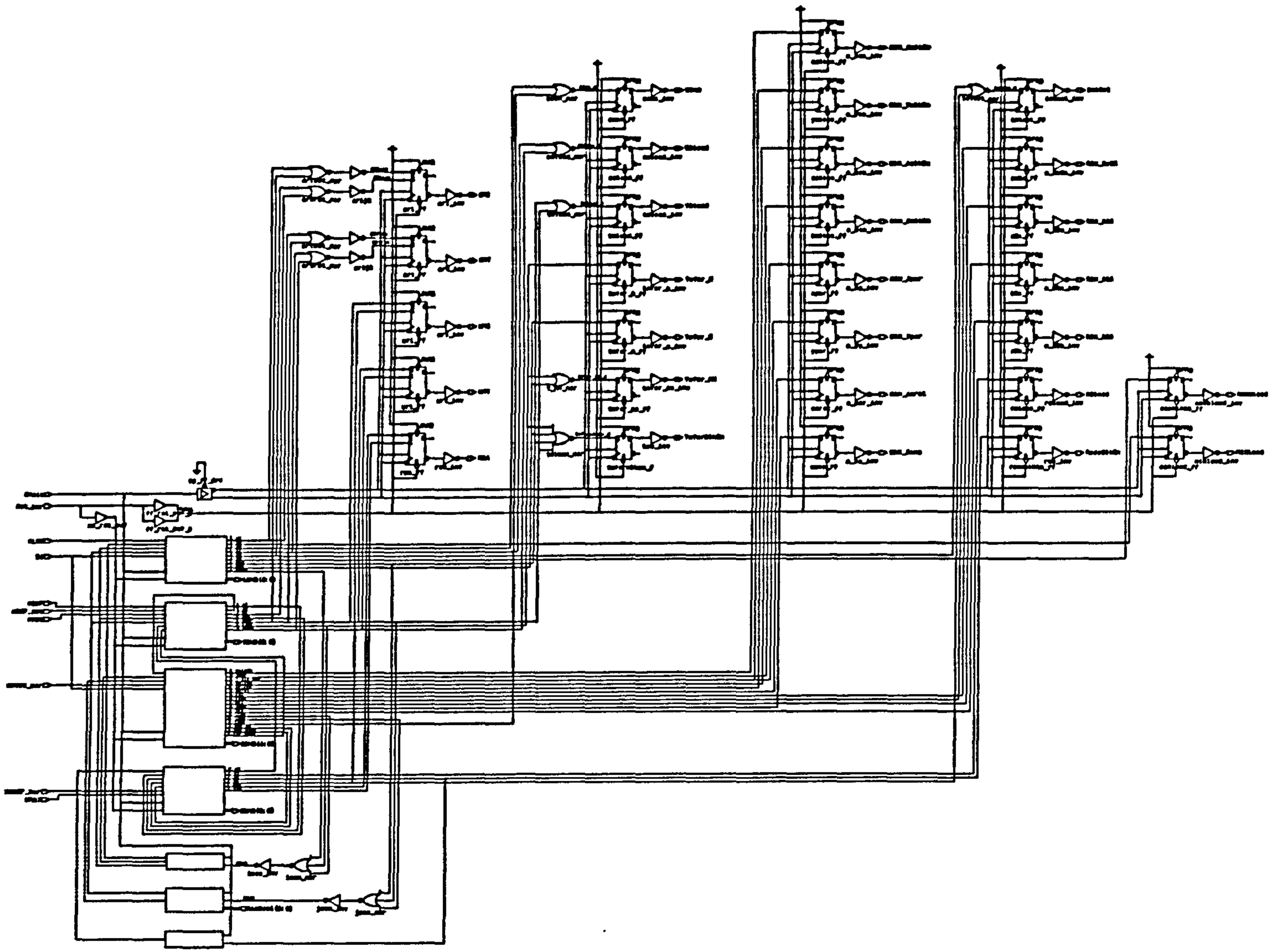
Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHiSpER 1.0
SubCct:	NAdd



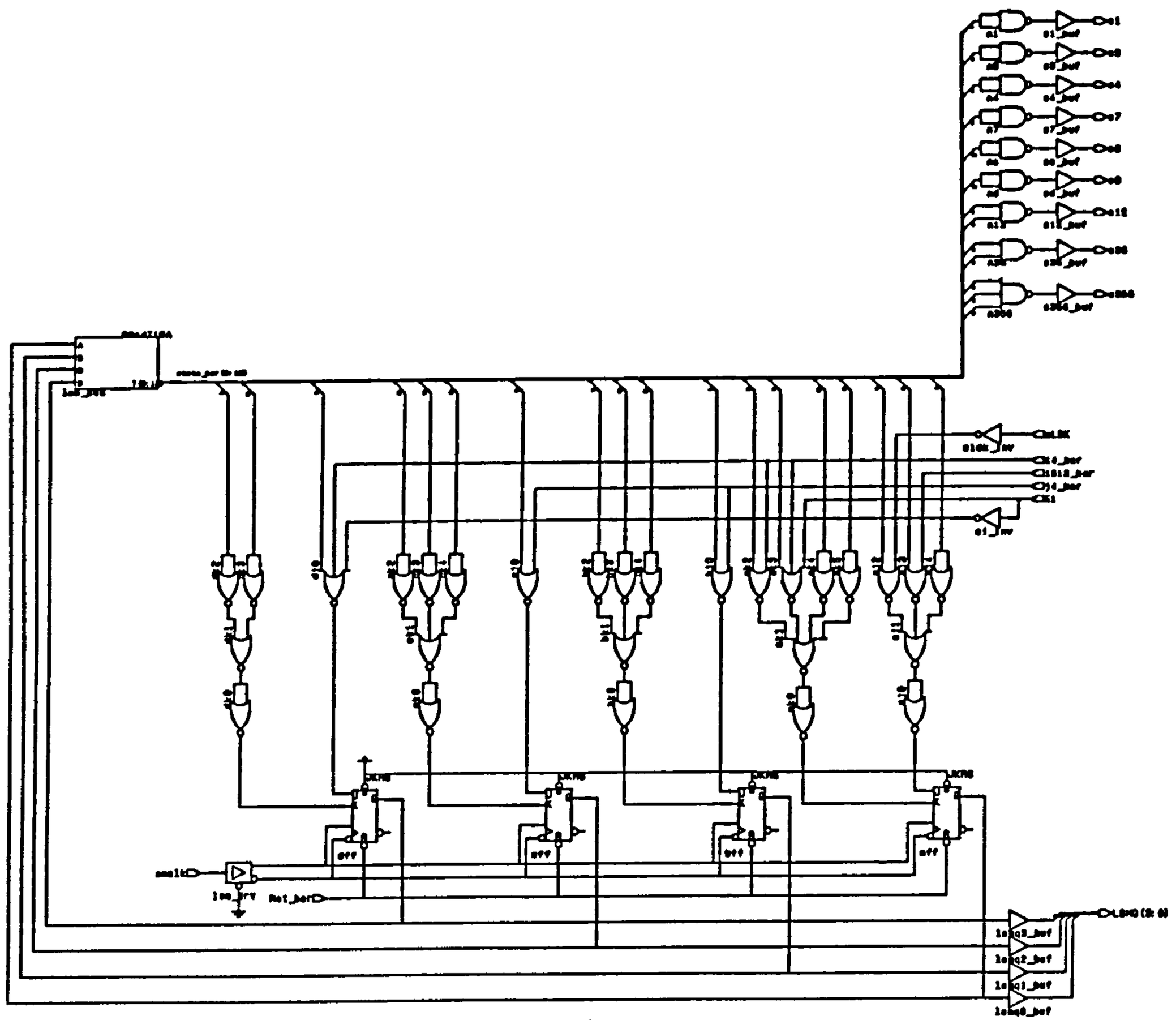
Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHISpER 1.0
SubCct:	NMult



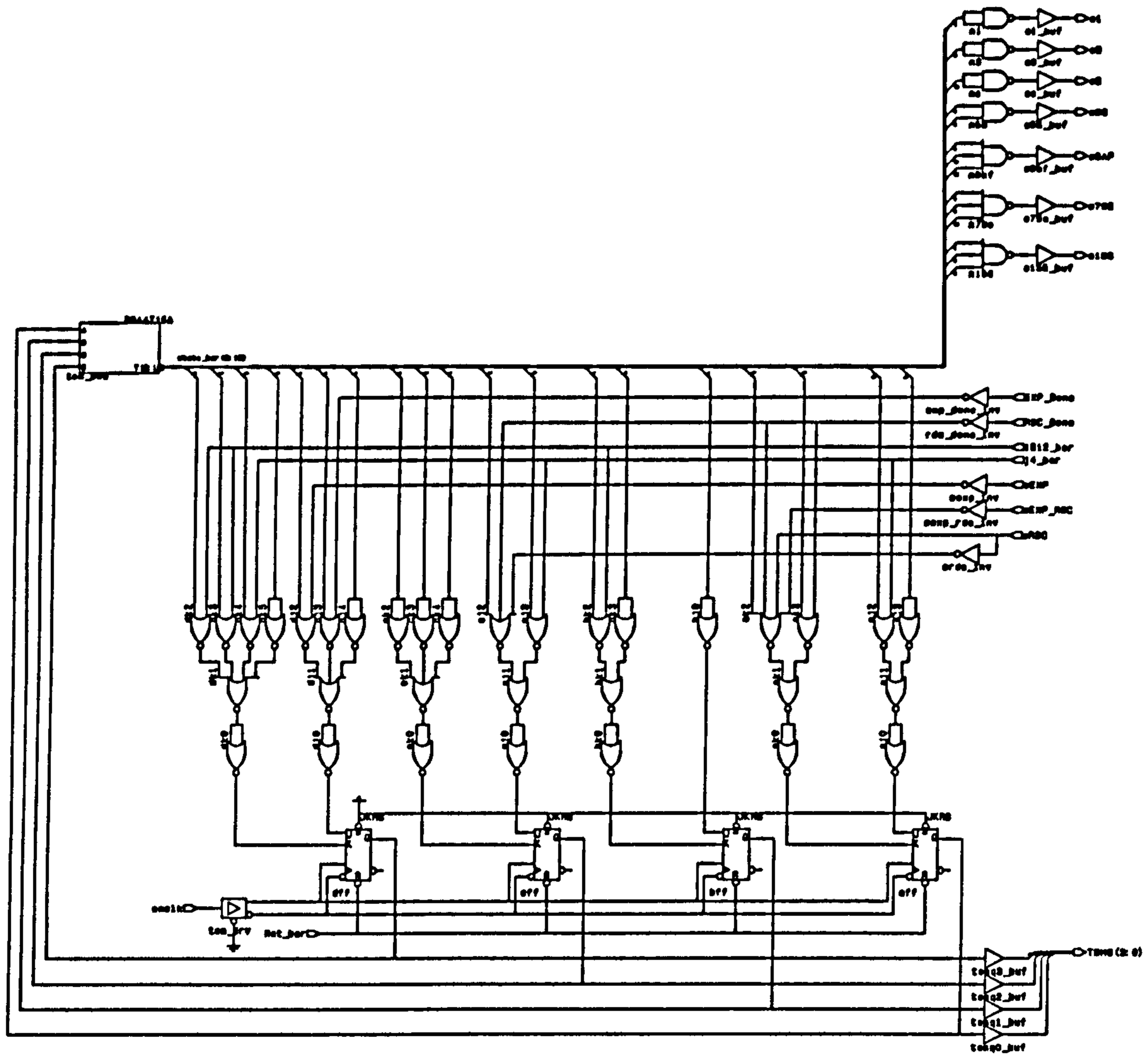
Author:	Paul D. Ohlone
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	UNIXpER 1.0
SubDir:	SRG



Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHISPER 1.0
SubCat:	LSM

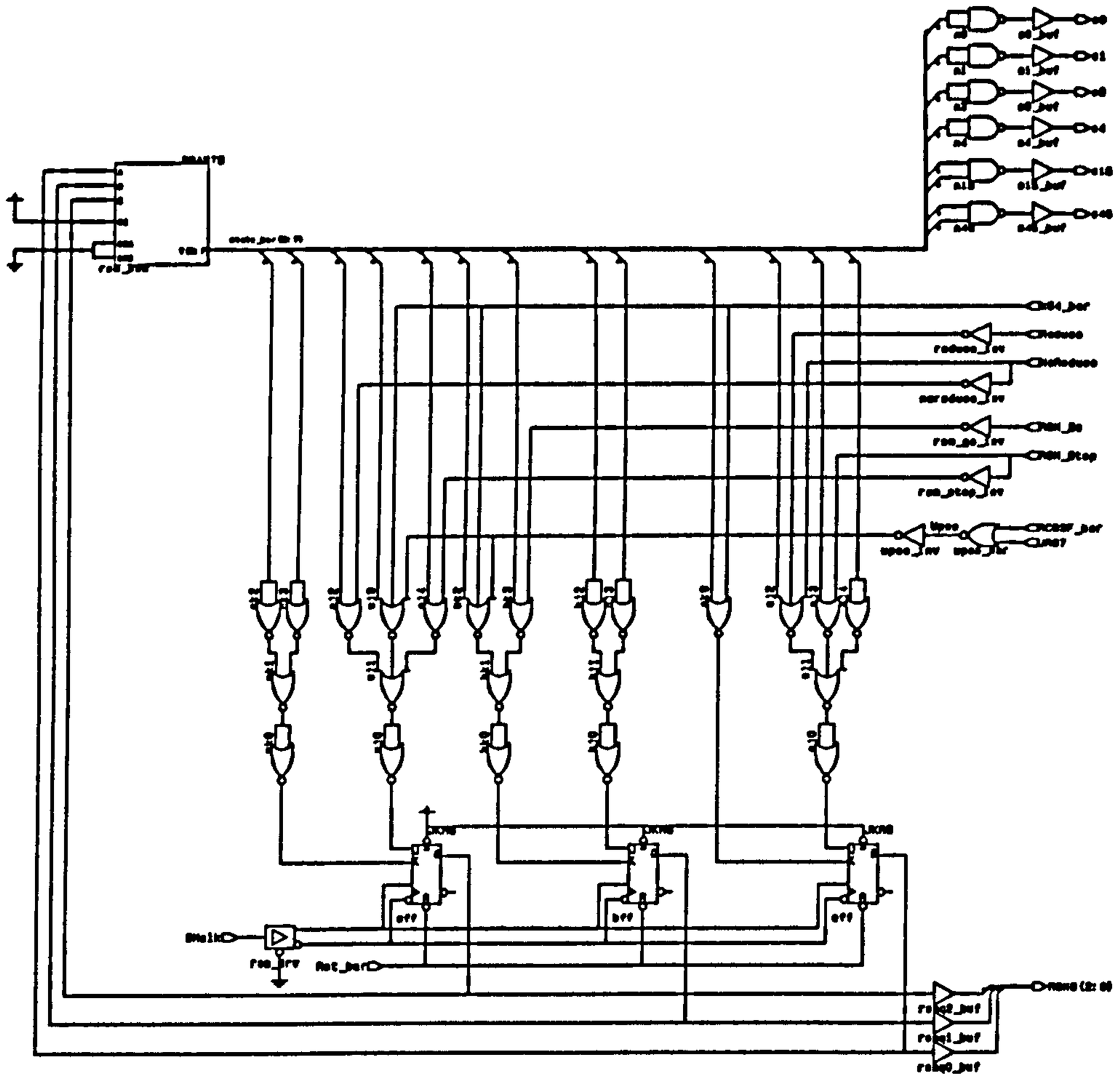


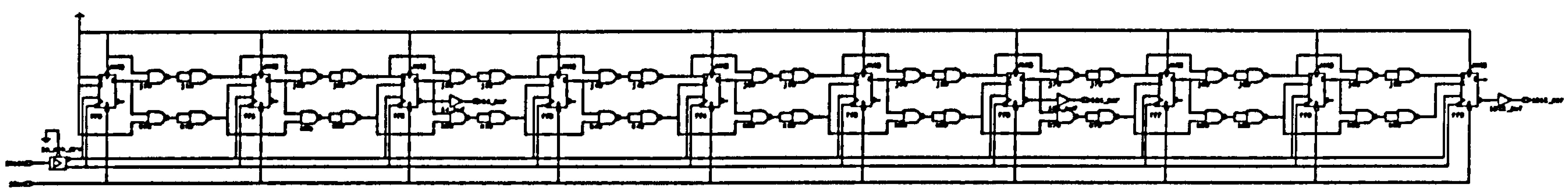
Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHISPER 1.0
SubCct:	TSM



Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	Univereity of Plymouth
Project:	WHISPER 1.0
SubCct:	ESM

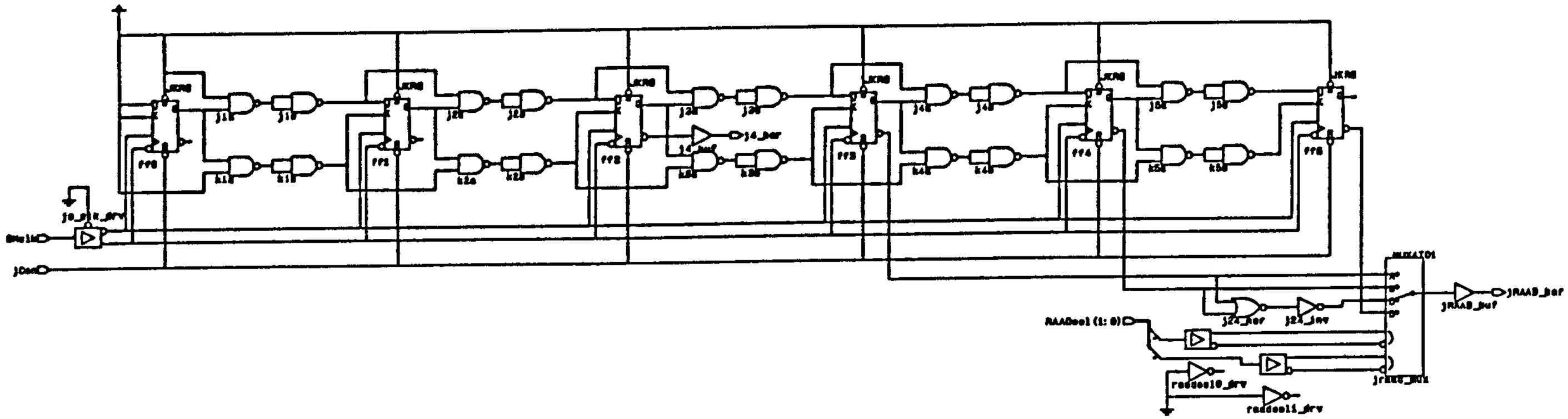
Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHISpER 1.0
SubCat:	ASM



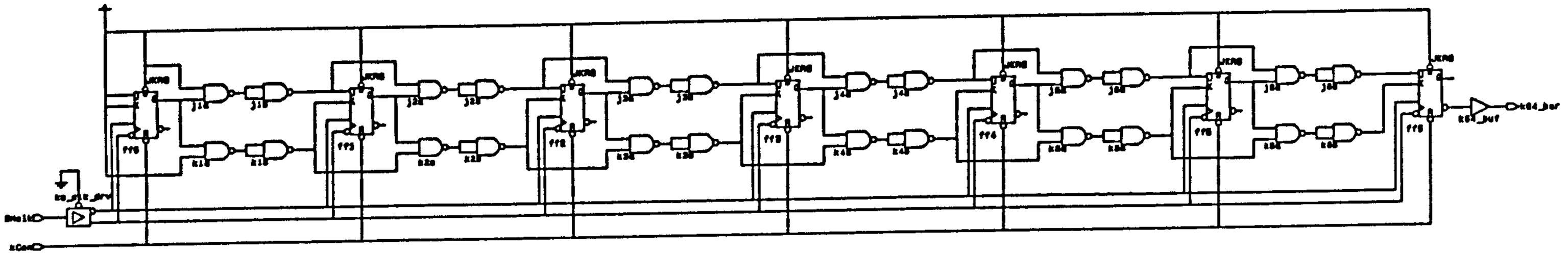


Author:	Paul B. Onions
Date:	June 1994
Dept.	Security Research Group
Org:	University of Plymouth
Project:	OnIDEA 1.0
SubCct:	18

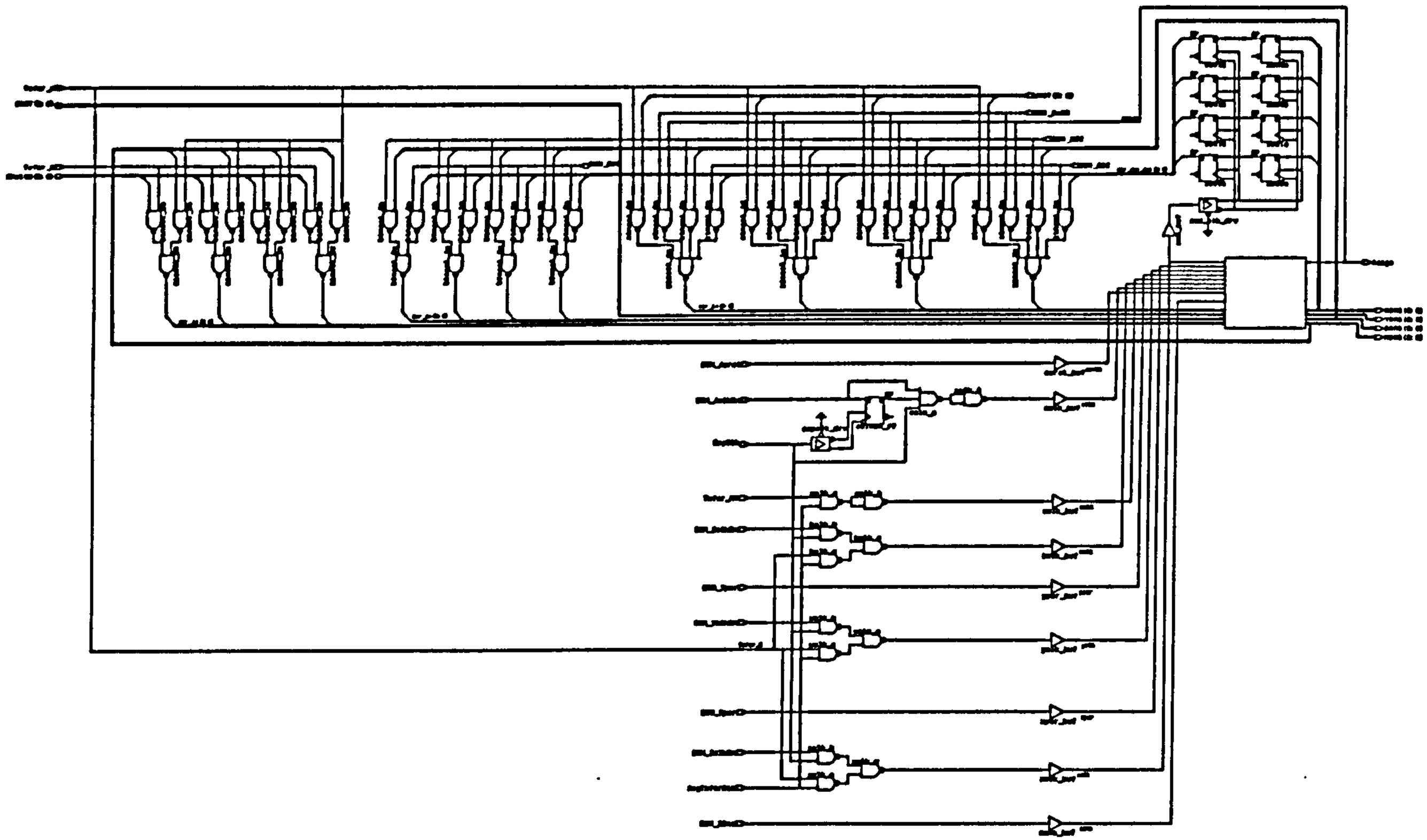
Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WH18pER 1.0
SubCct:	jC

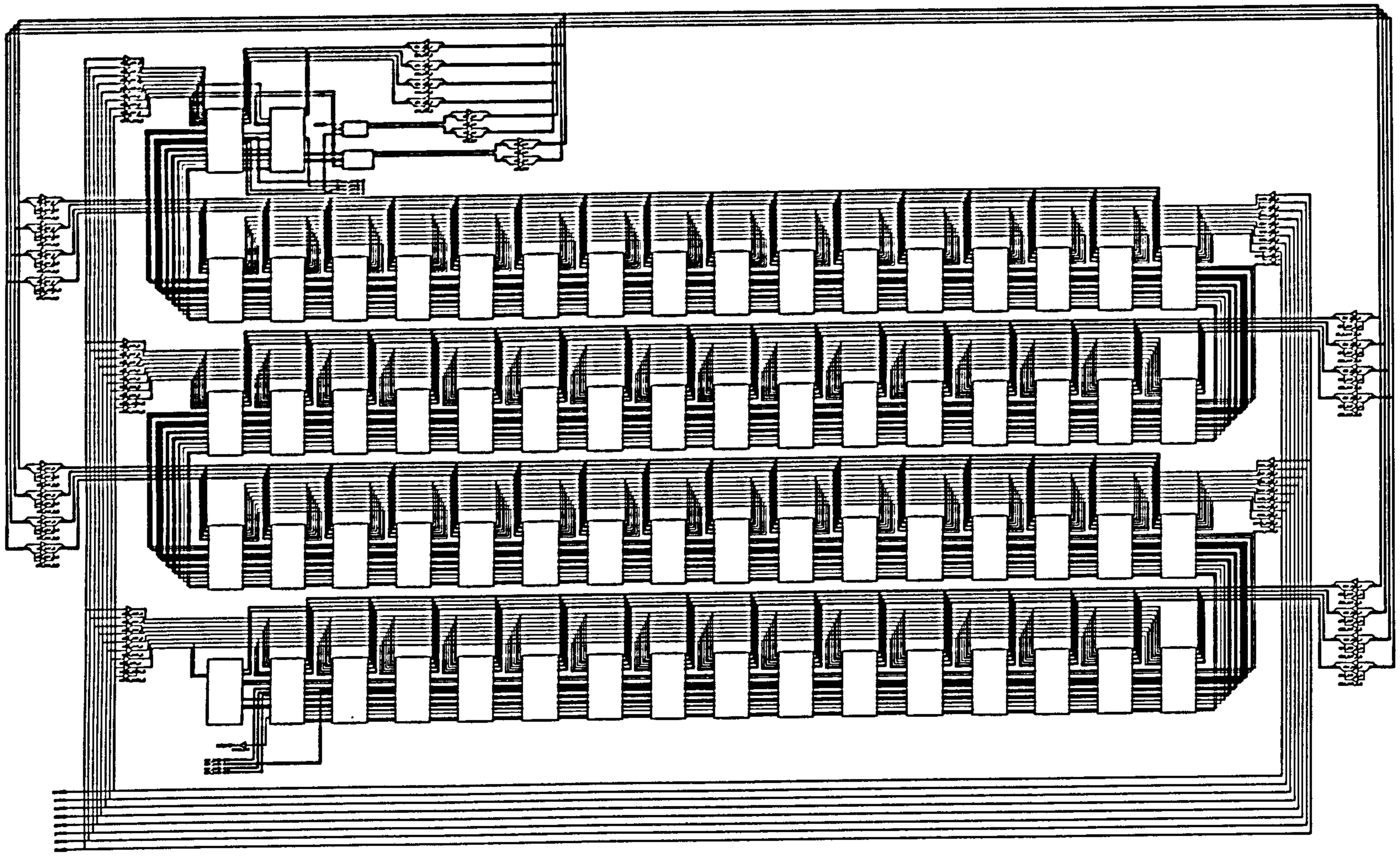


Author:	Paul D. Oniane
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHISPER 1.0
SubCct:	kC

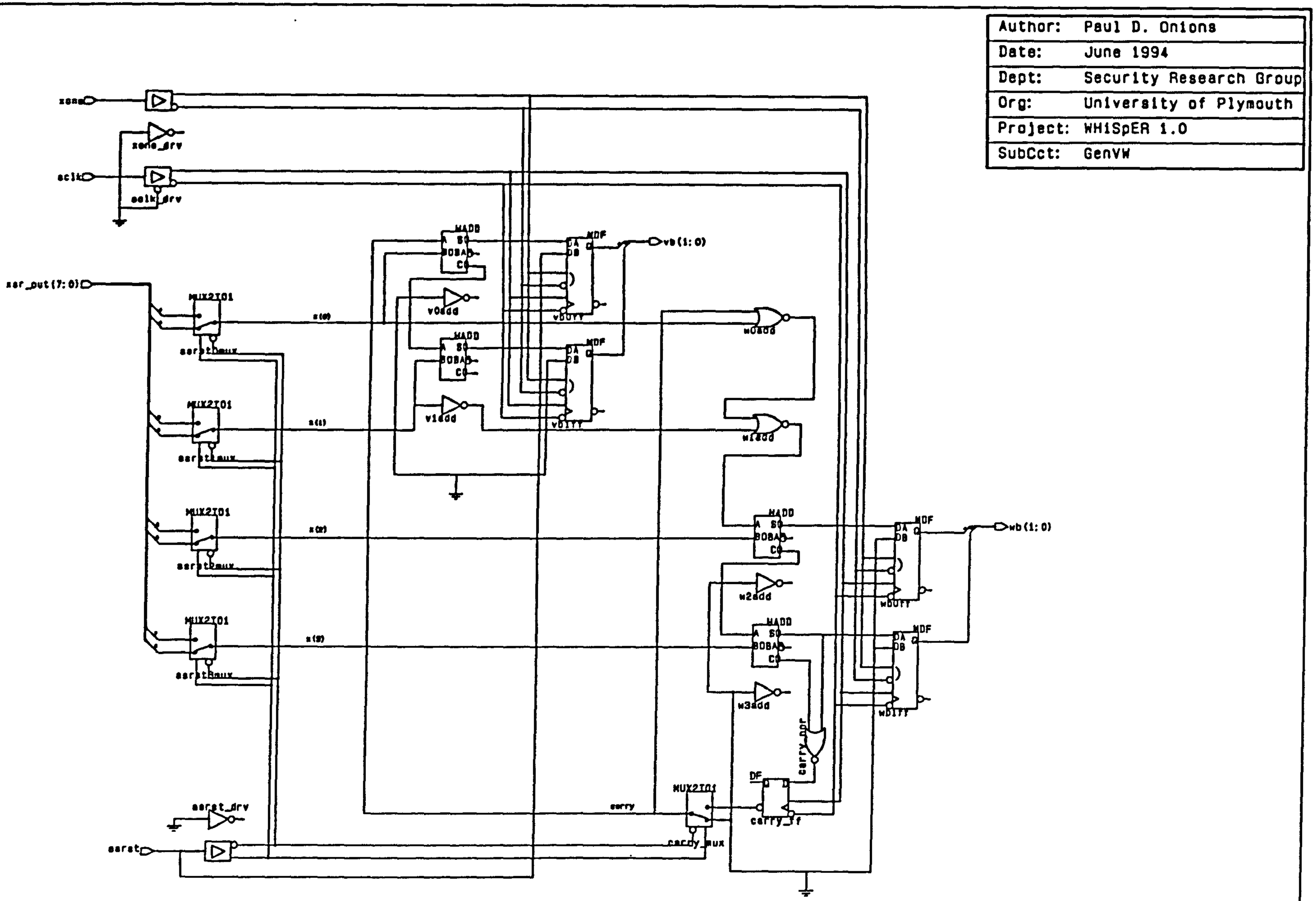


Author:	Paul S. Dalene
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	HMISpES 1.0
SubCct:	AHE

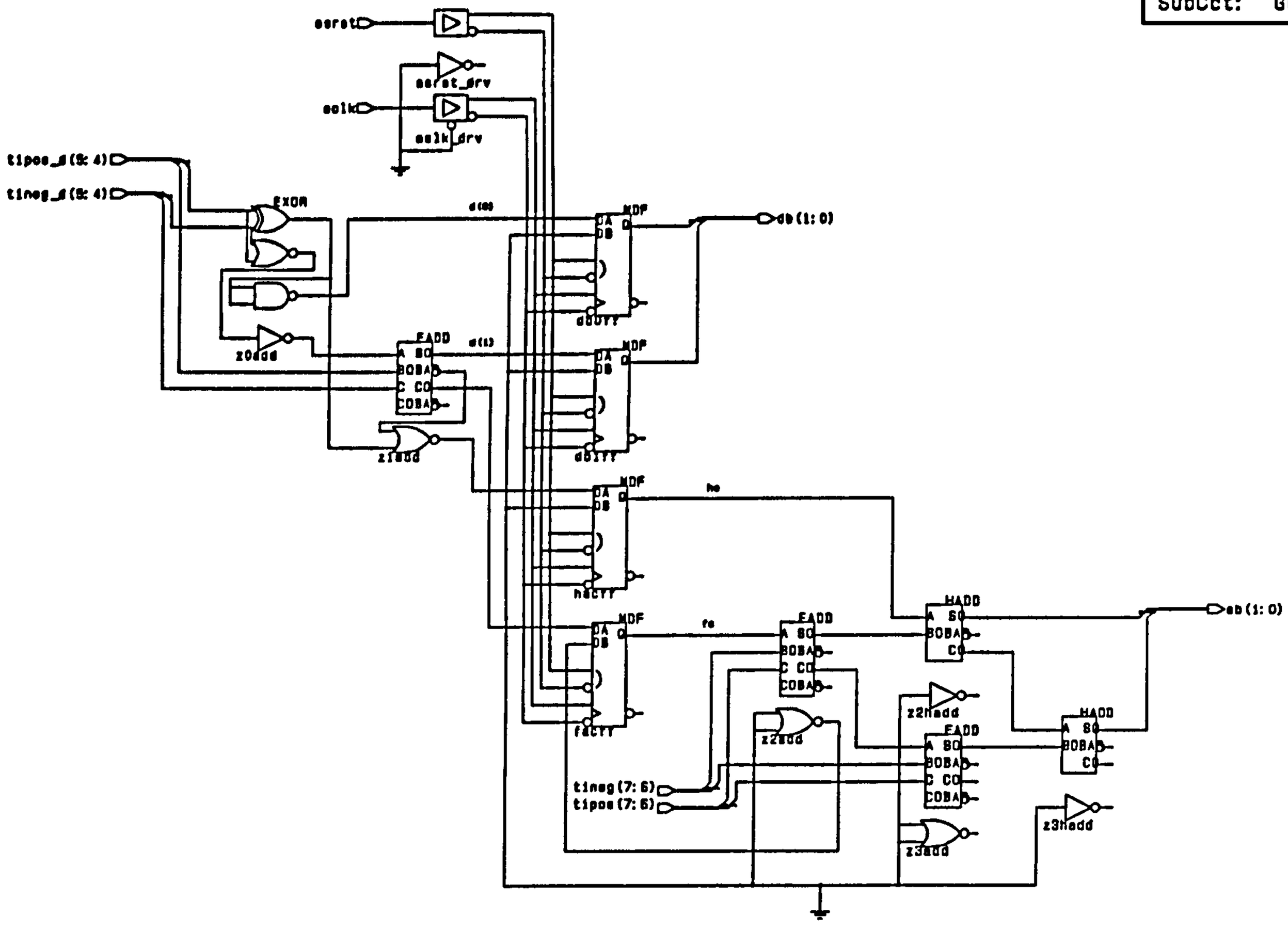




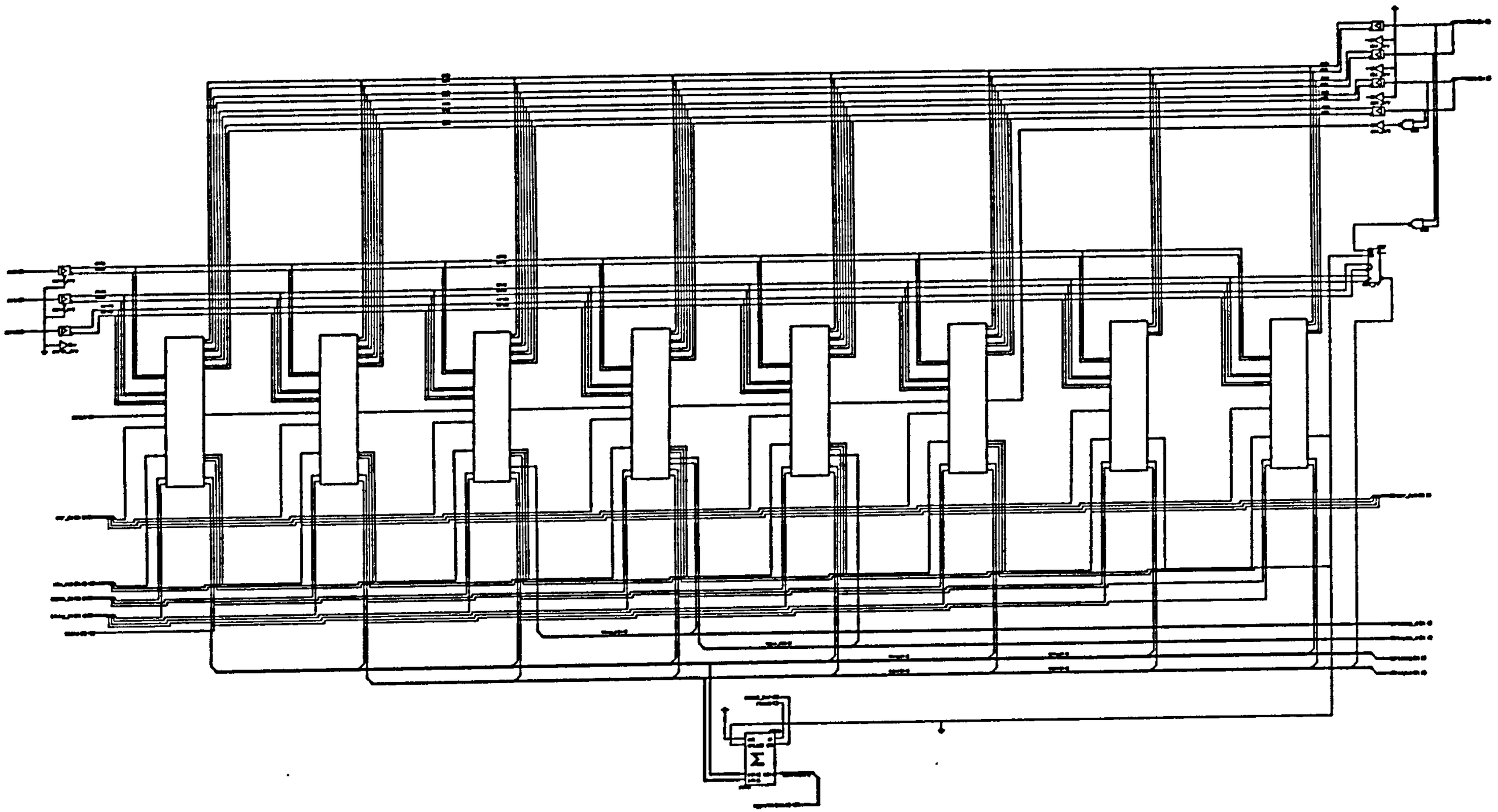
Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHiSpER 1.0
SubCct:	GenVW

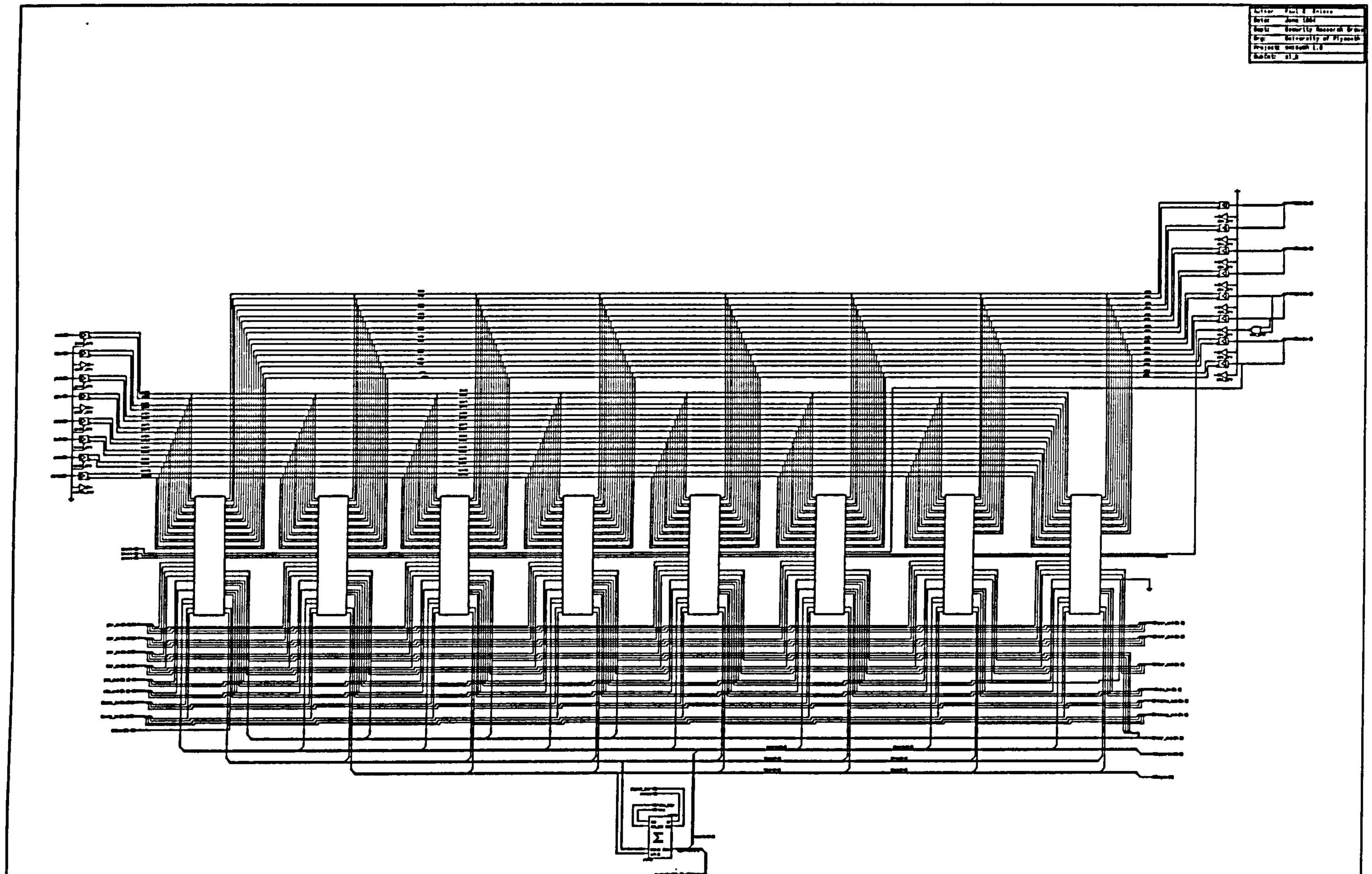


Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHiSpER 1.0
SubCct:	GenDE

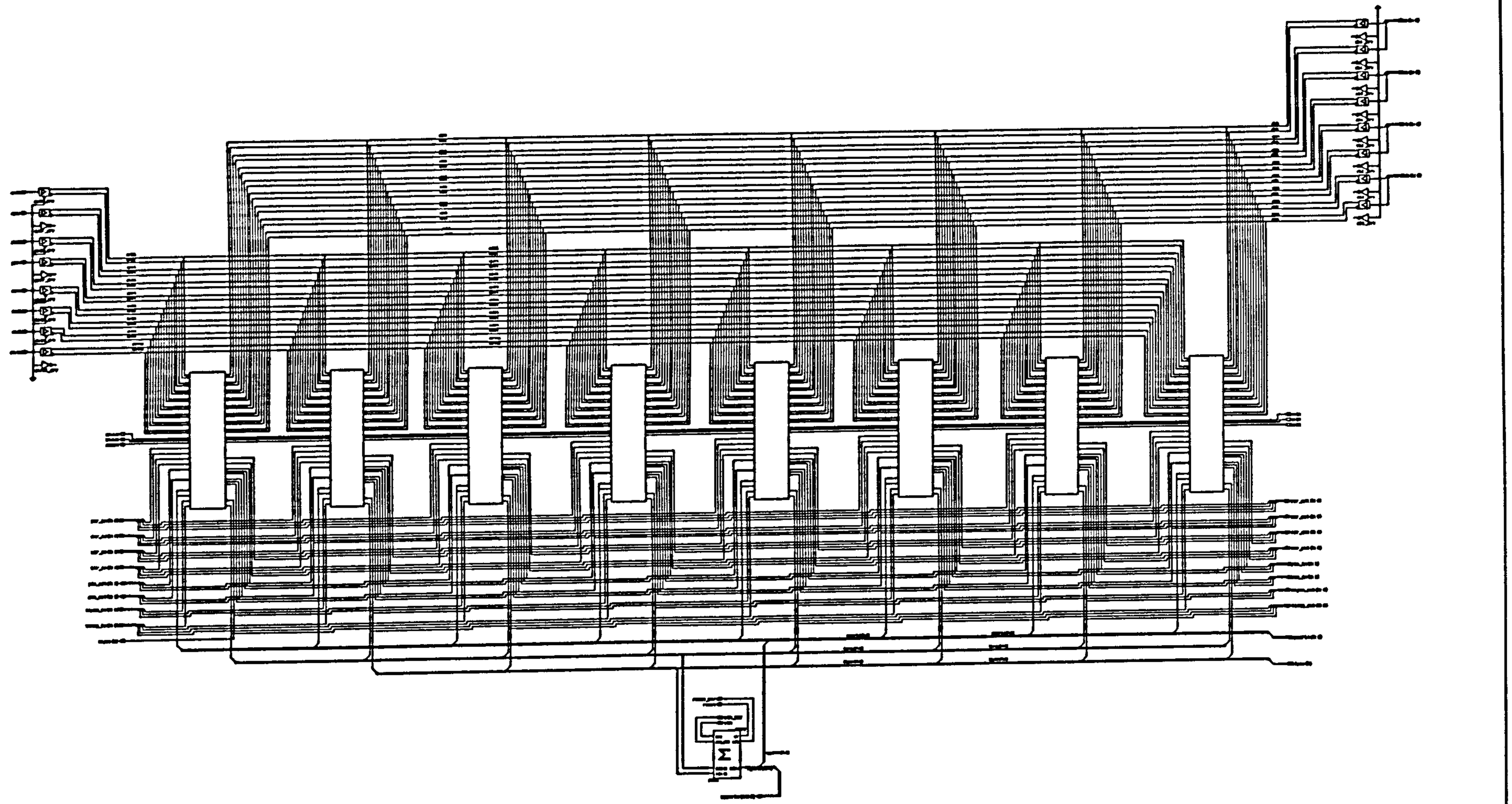


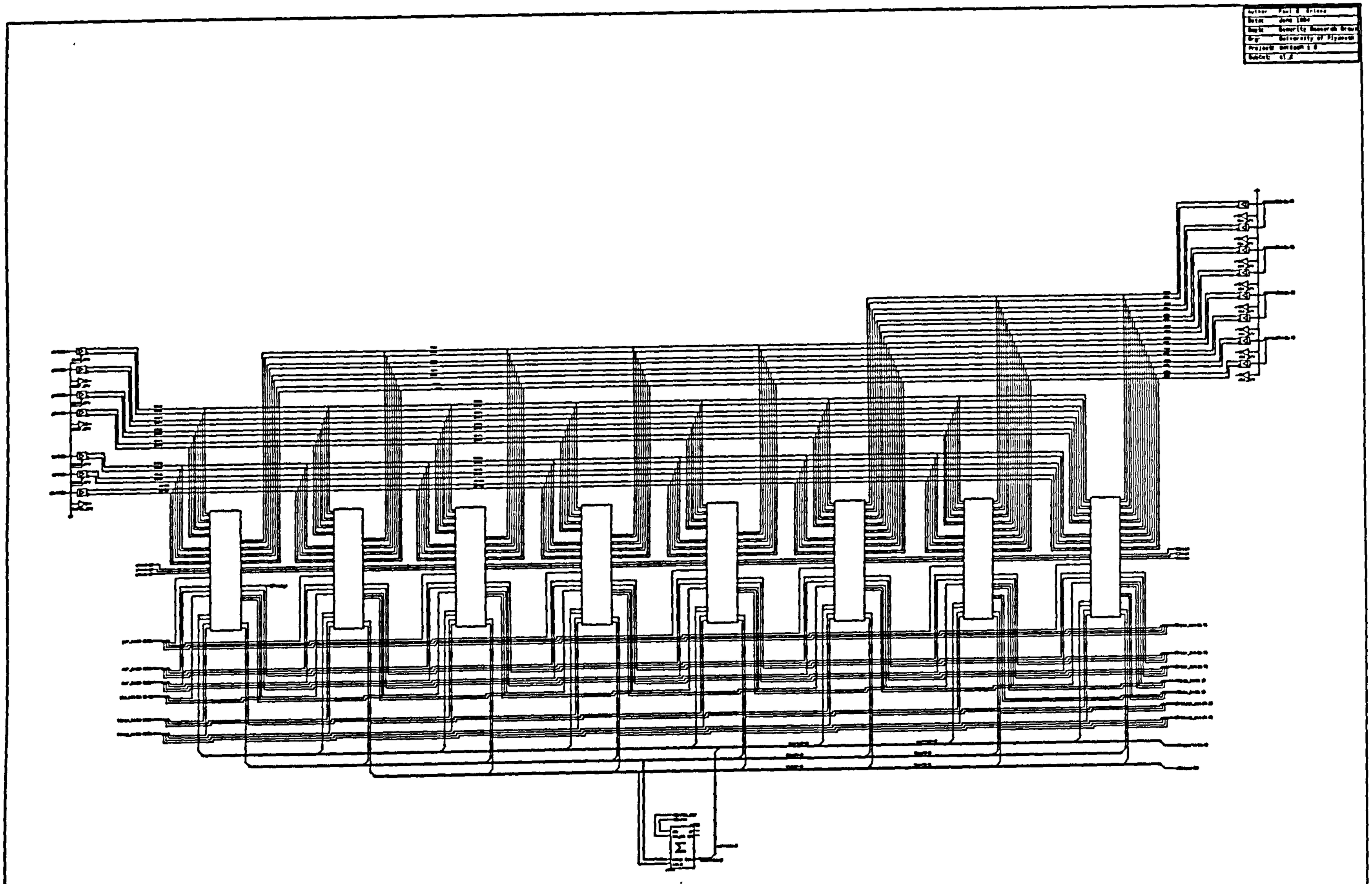
Author	Paul B Brians
Date	June 1994
Dept	Security Research Group
Org	University of Plymouth
Project	Unidpm 1.0
Subdir	alu



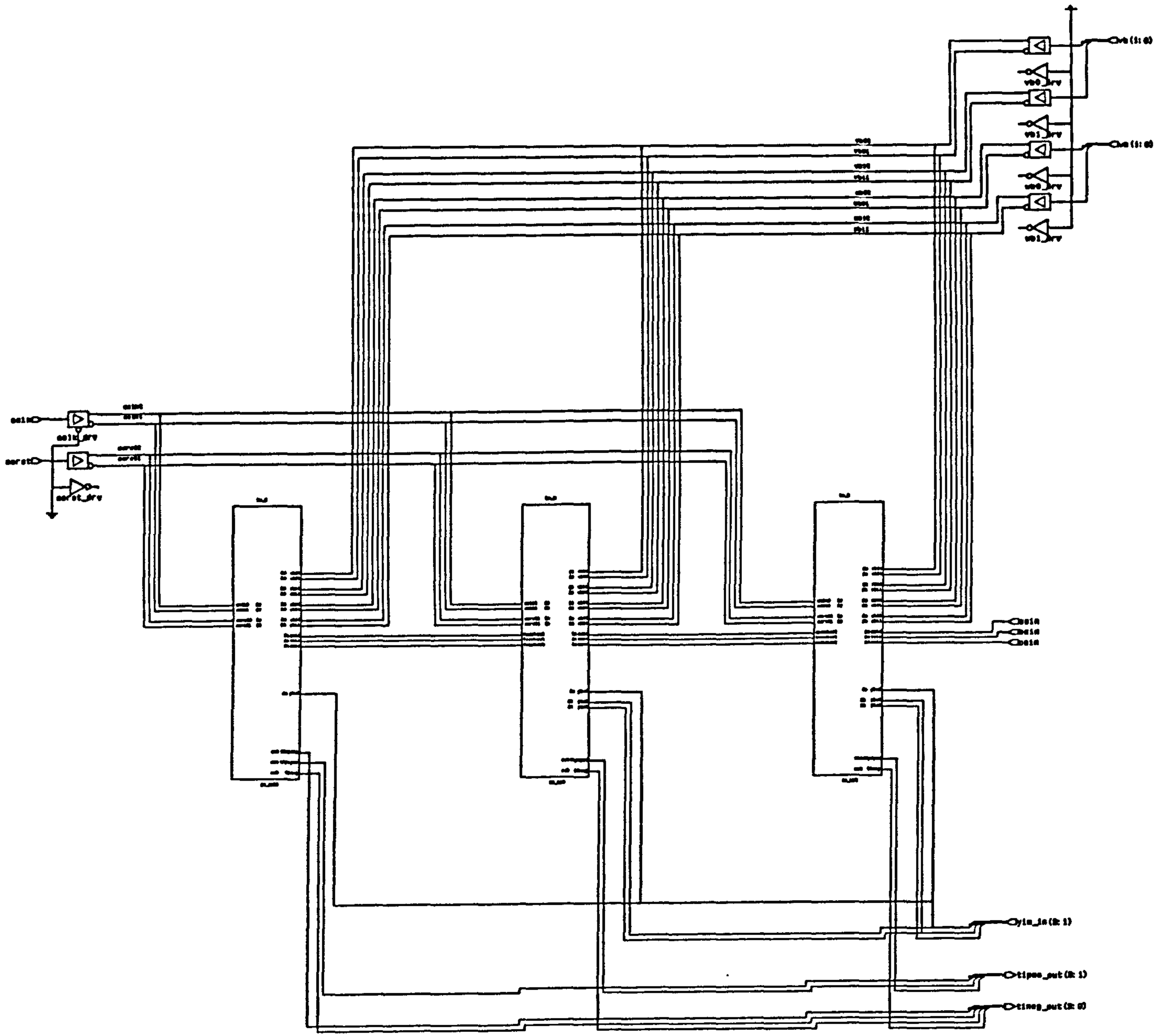

Σ

Author:	Paul S Dates
Date:	June 1994
Team:	Security Research Group
Org:	University of Plymouth
Project:	wmgpm 1.0
Sheet:	alu

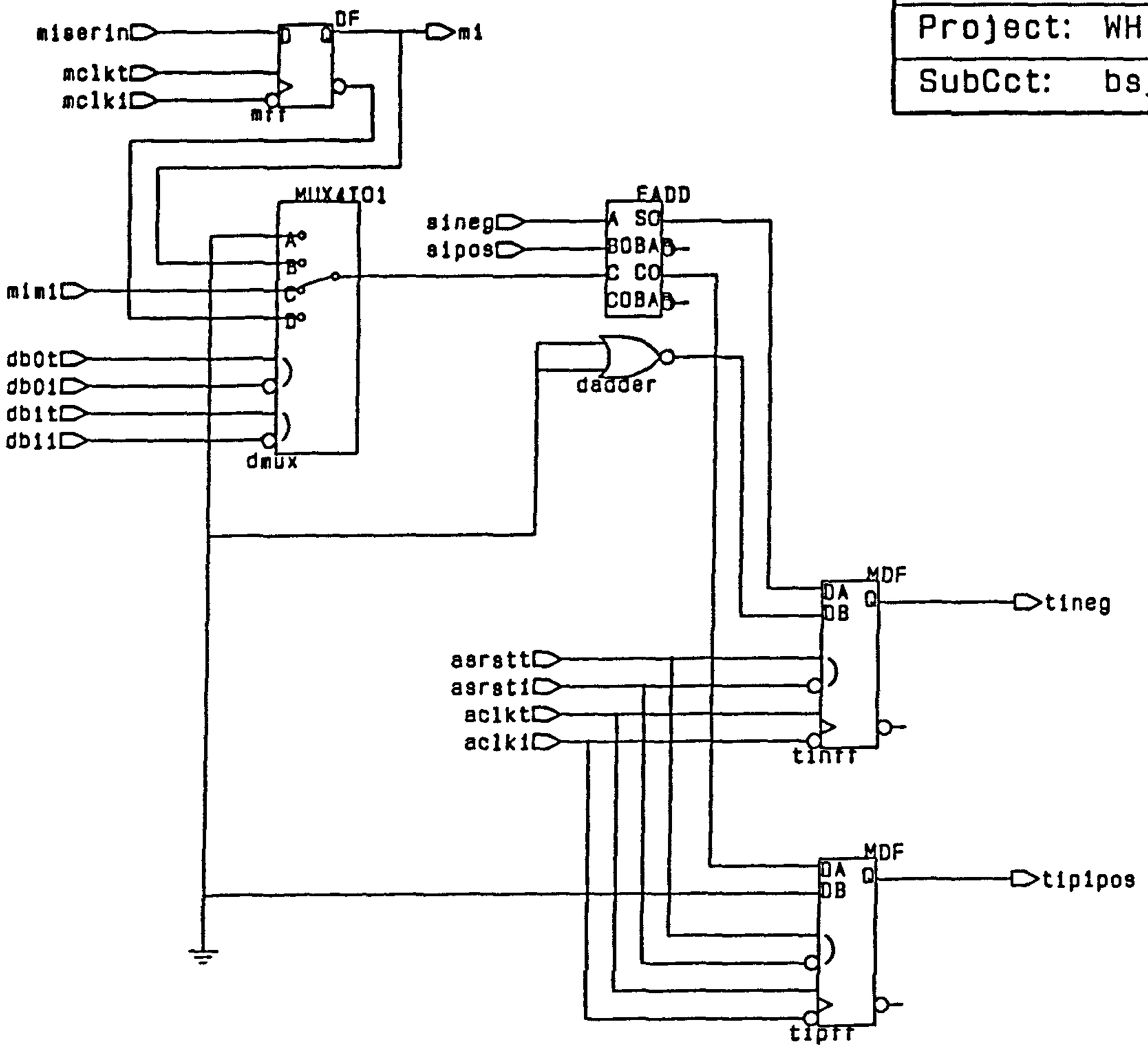



Σ

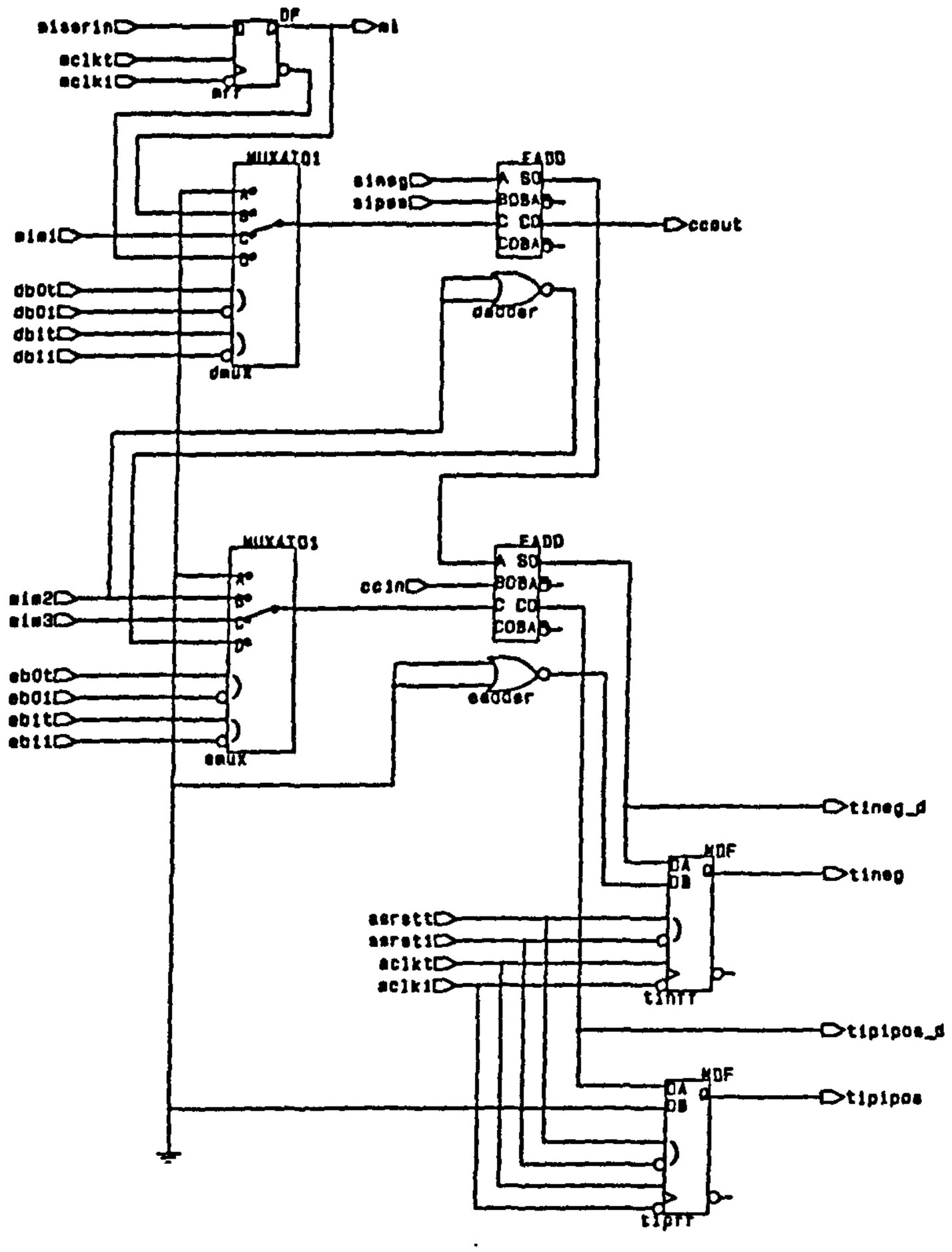
Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHiSpER 1.0
SubCct:	sl_s



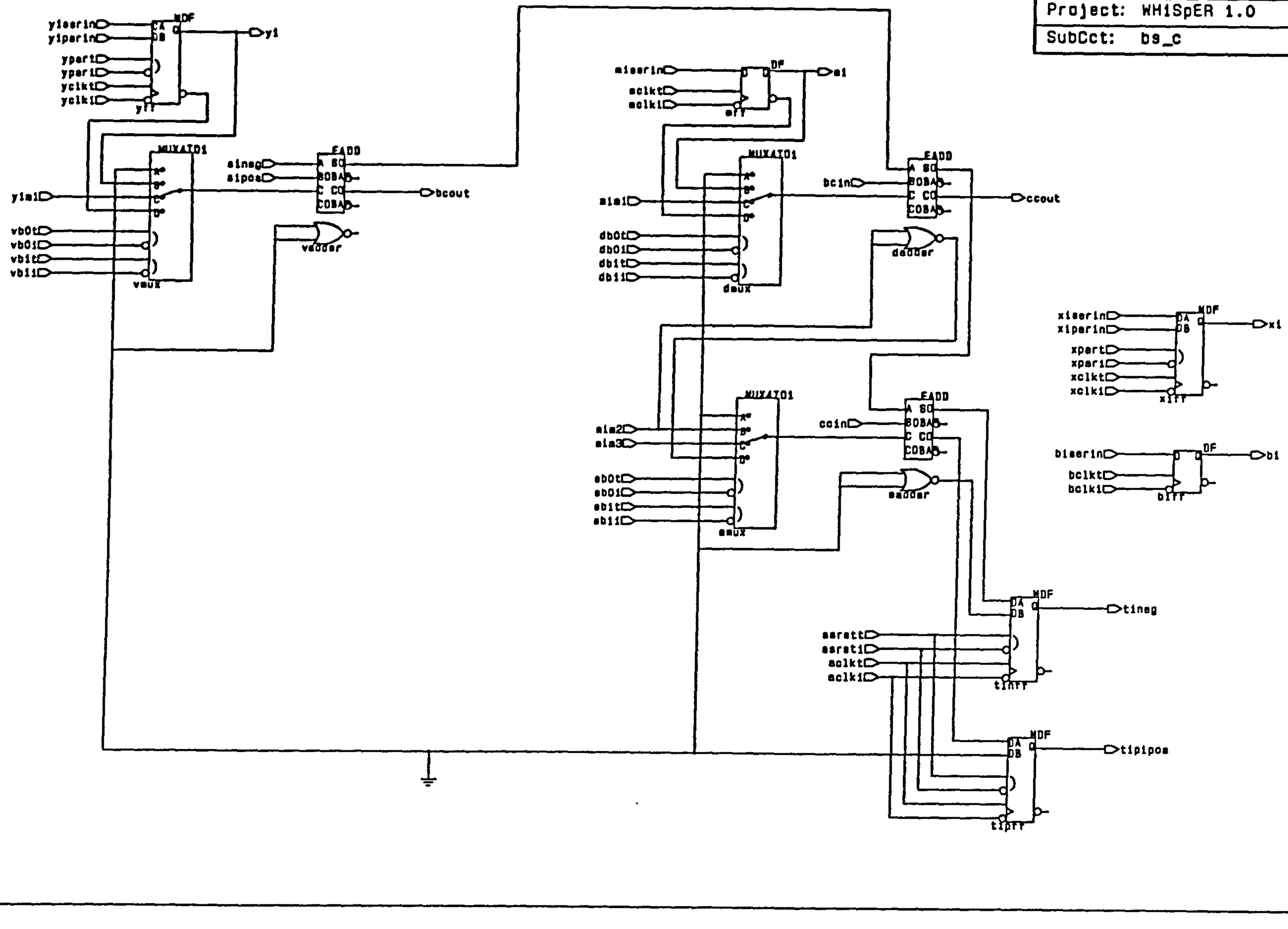
Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHiSpER 1.0
SubCct:	bs_a



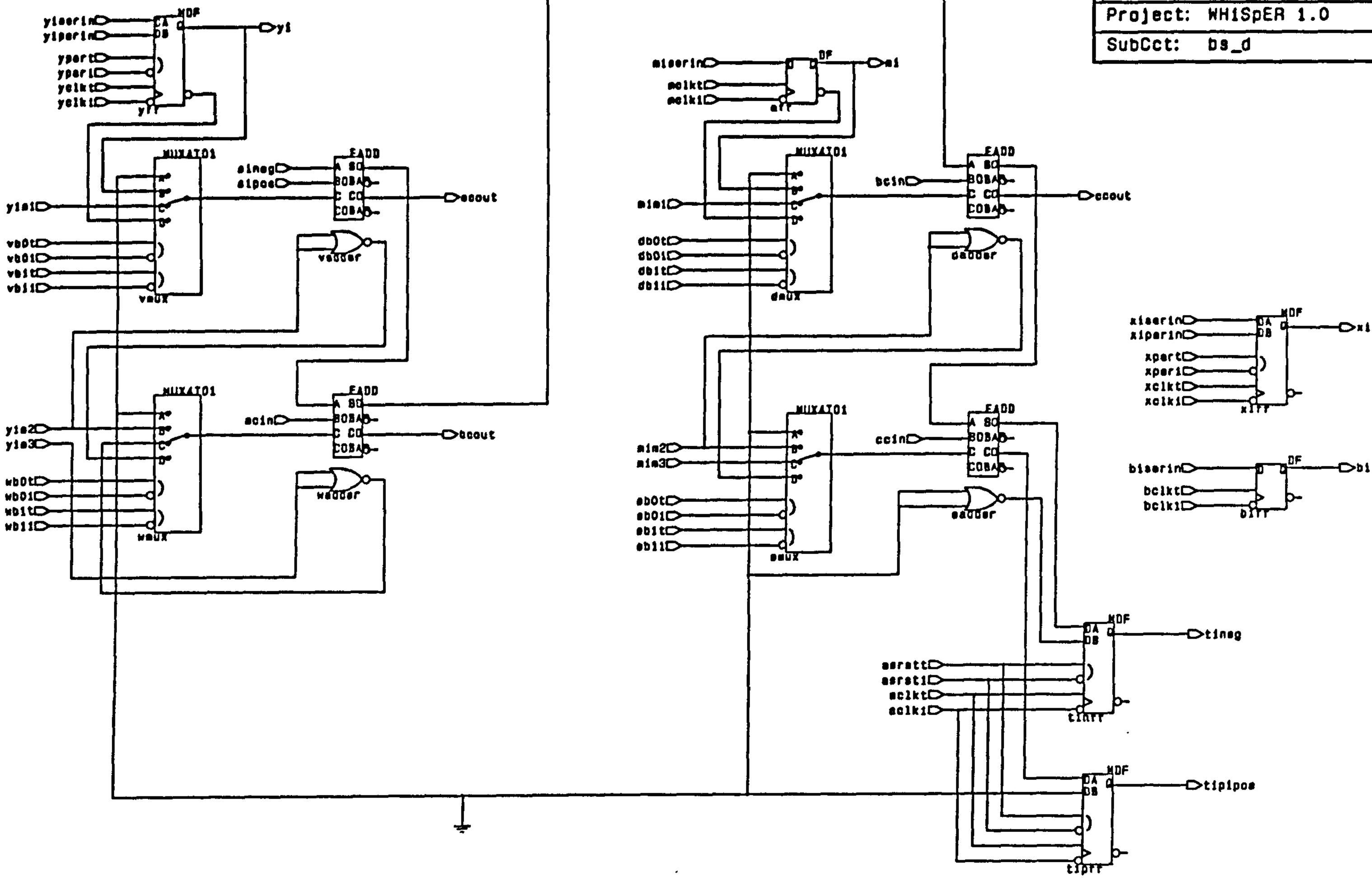
Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHiSpER 1.0
SubCct:	bs_b



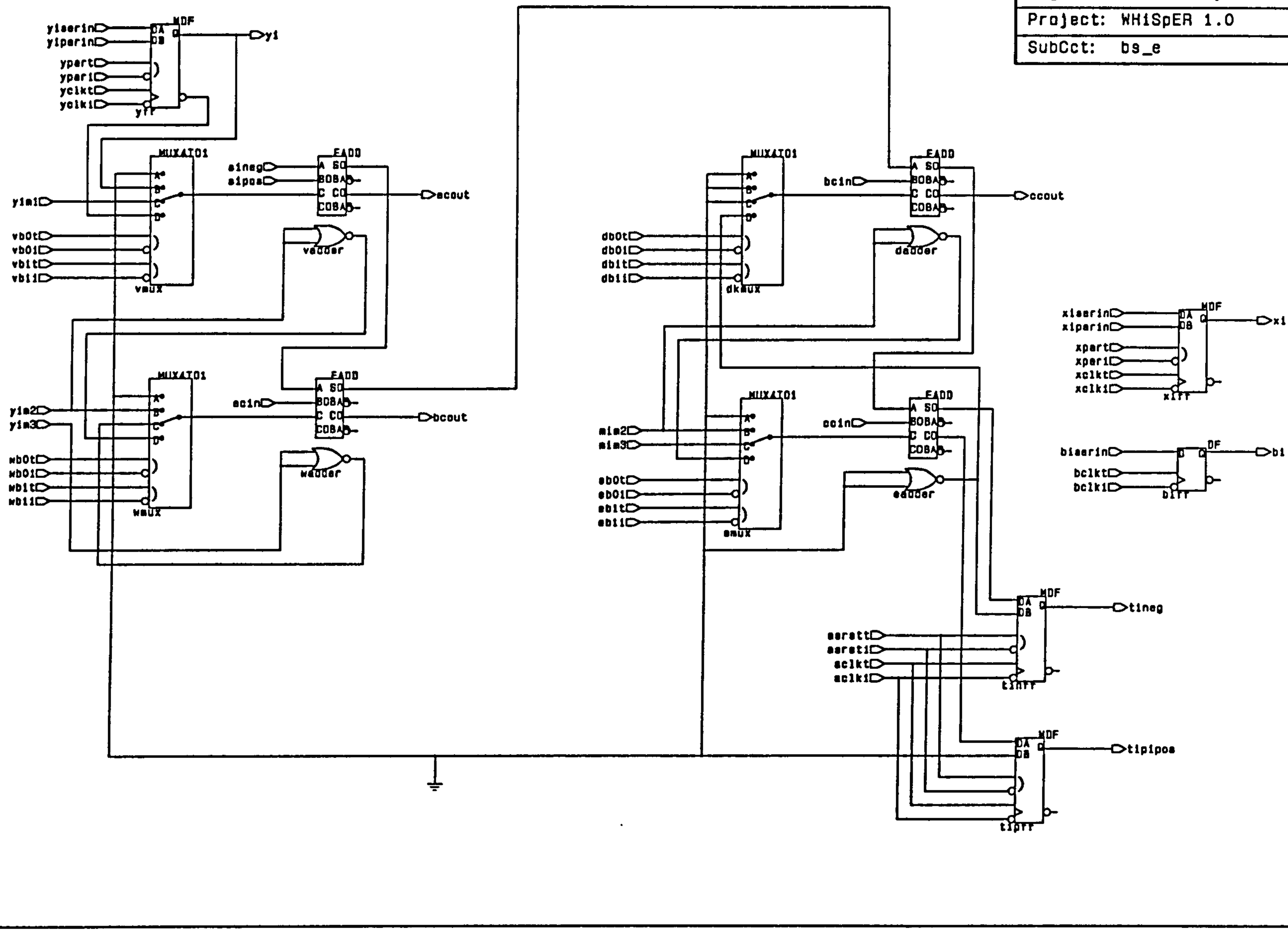
Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHiSpER 1.0
SubCct:	bs_c



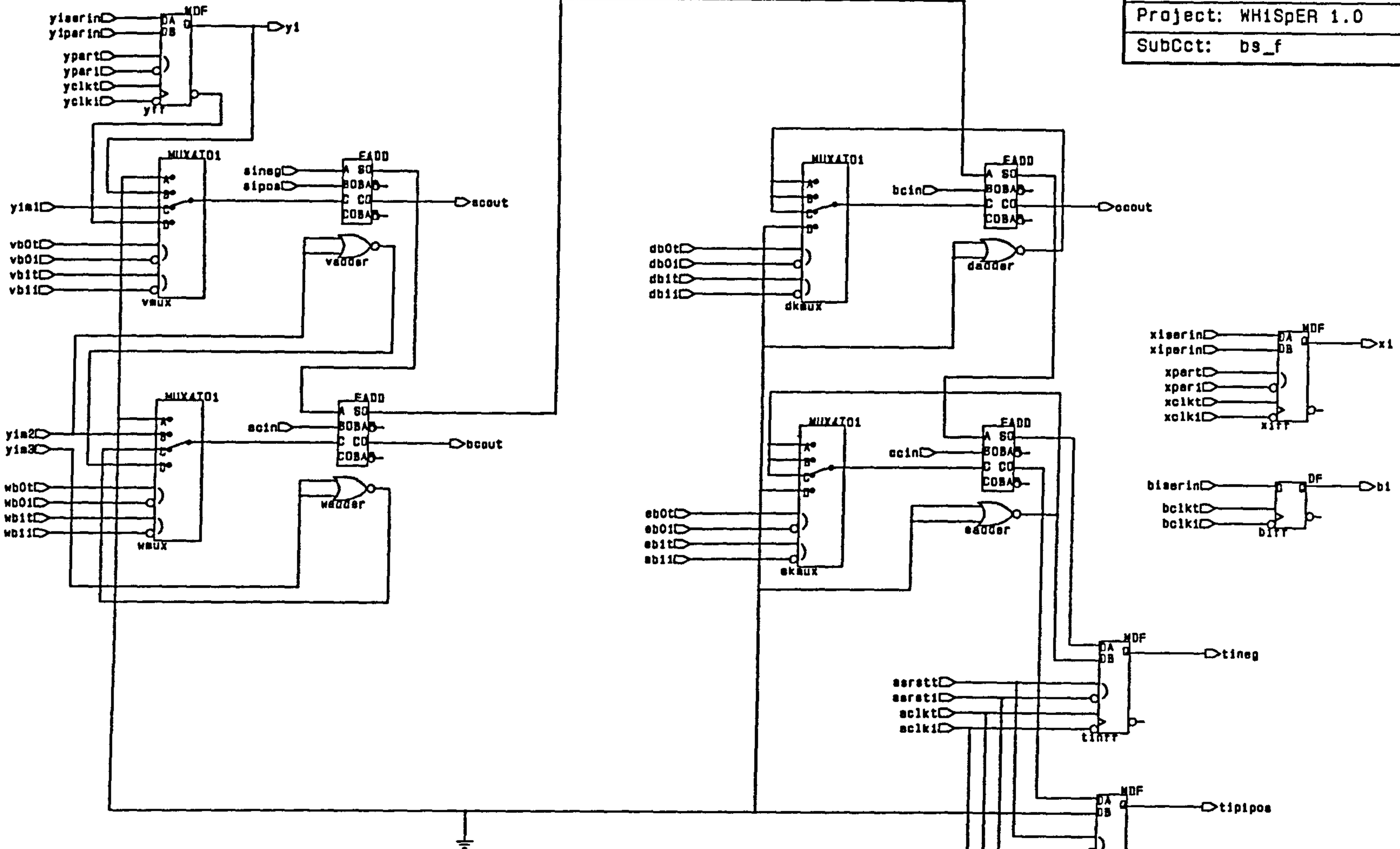
Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHiSpER 1.0
SubCct:	bs_d



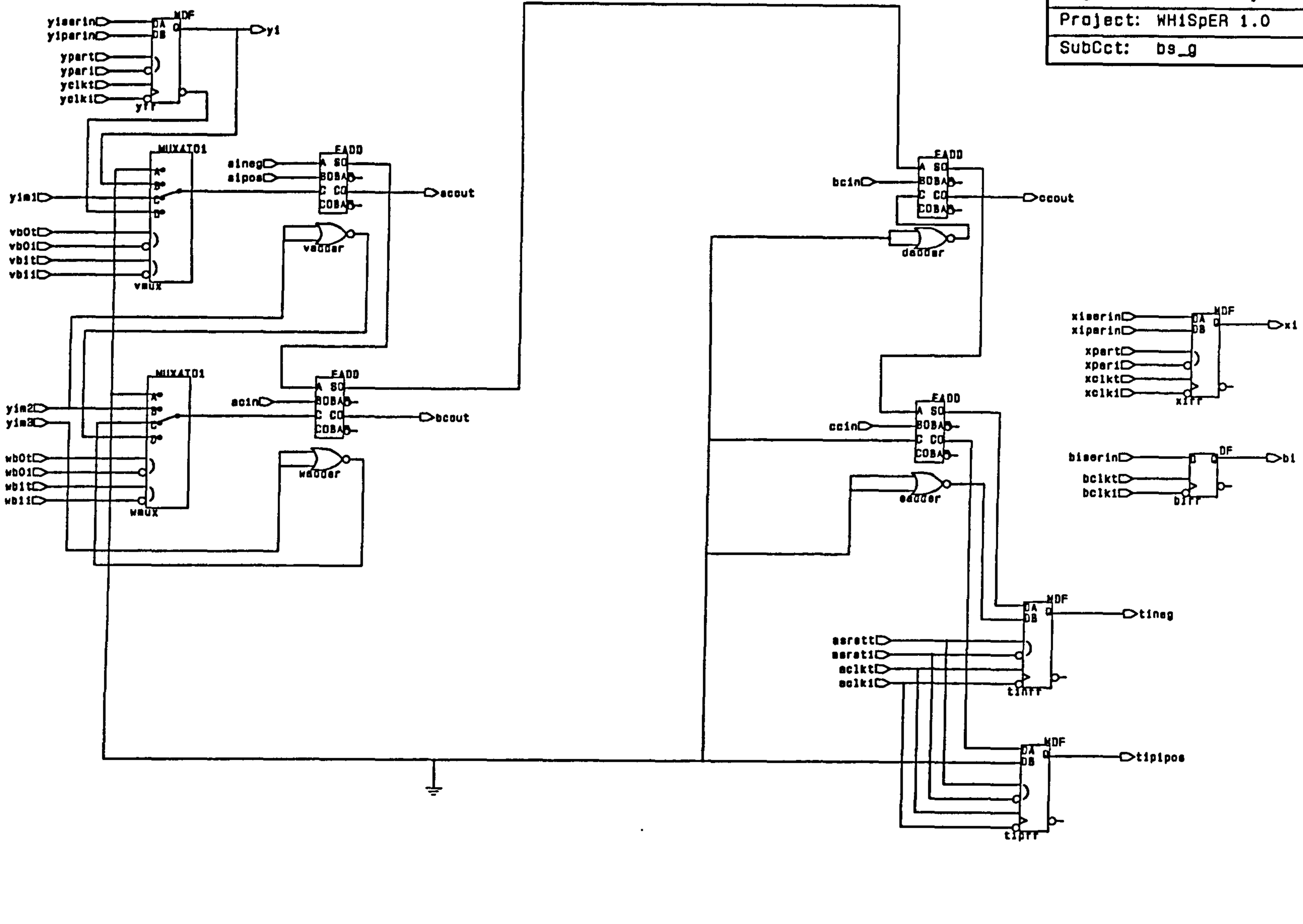
Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHiSpER 1.0
SubCct:	bs_e



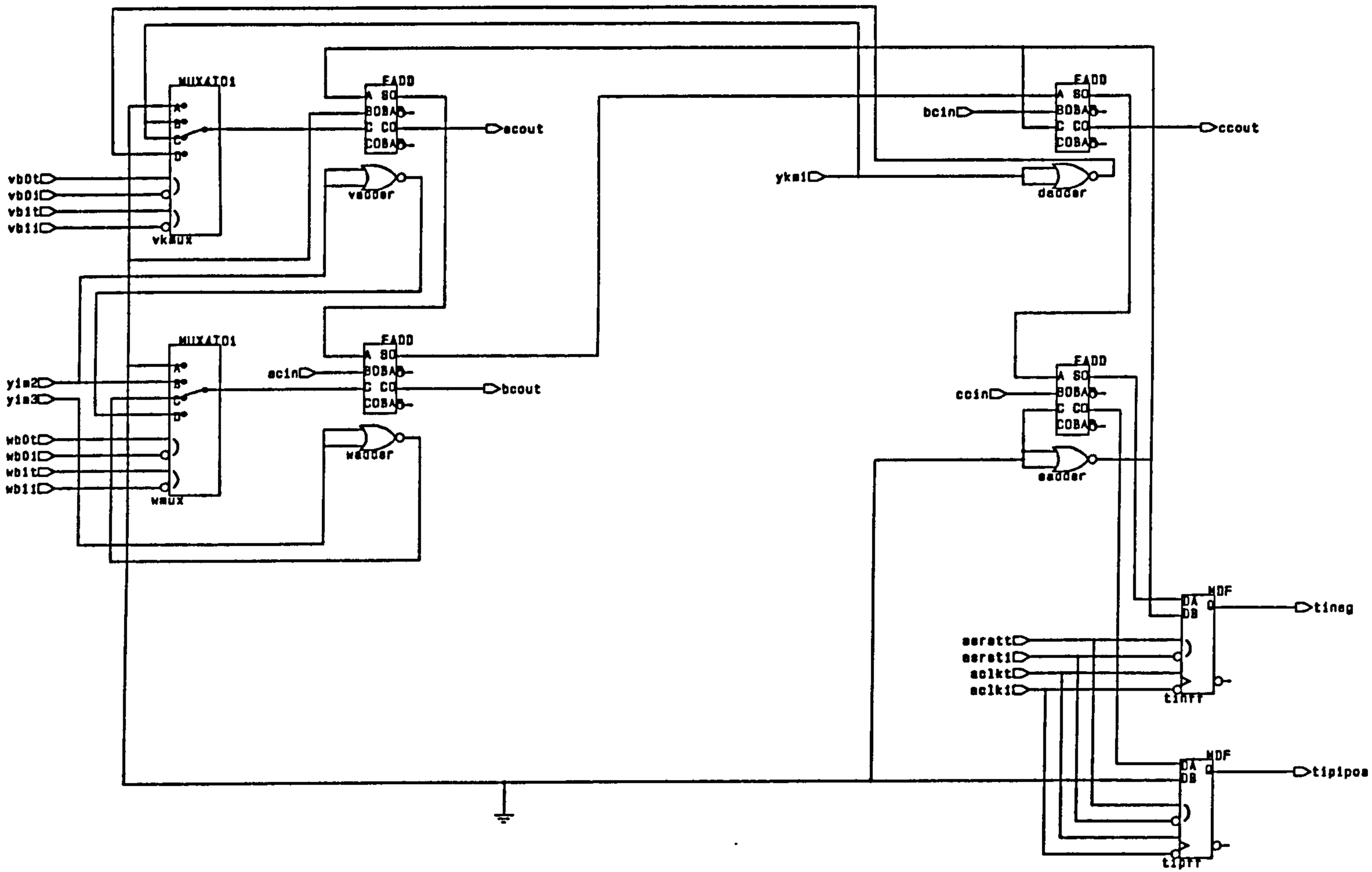
Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHiSpER 1.0
SubCct:	bs_f



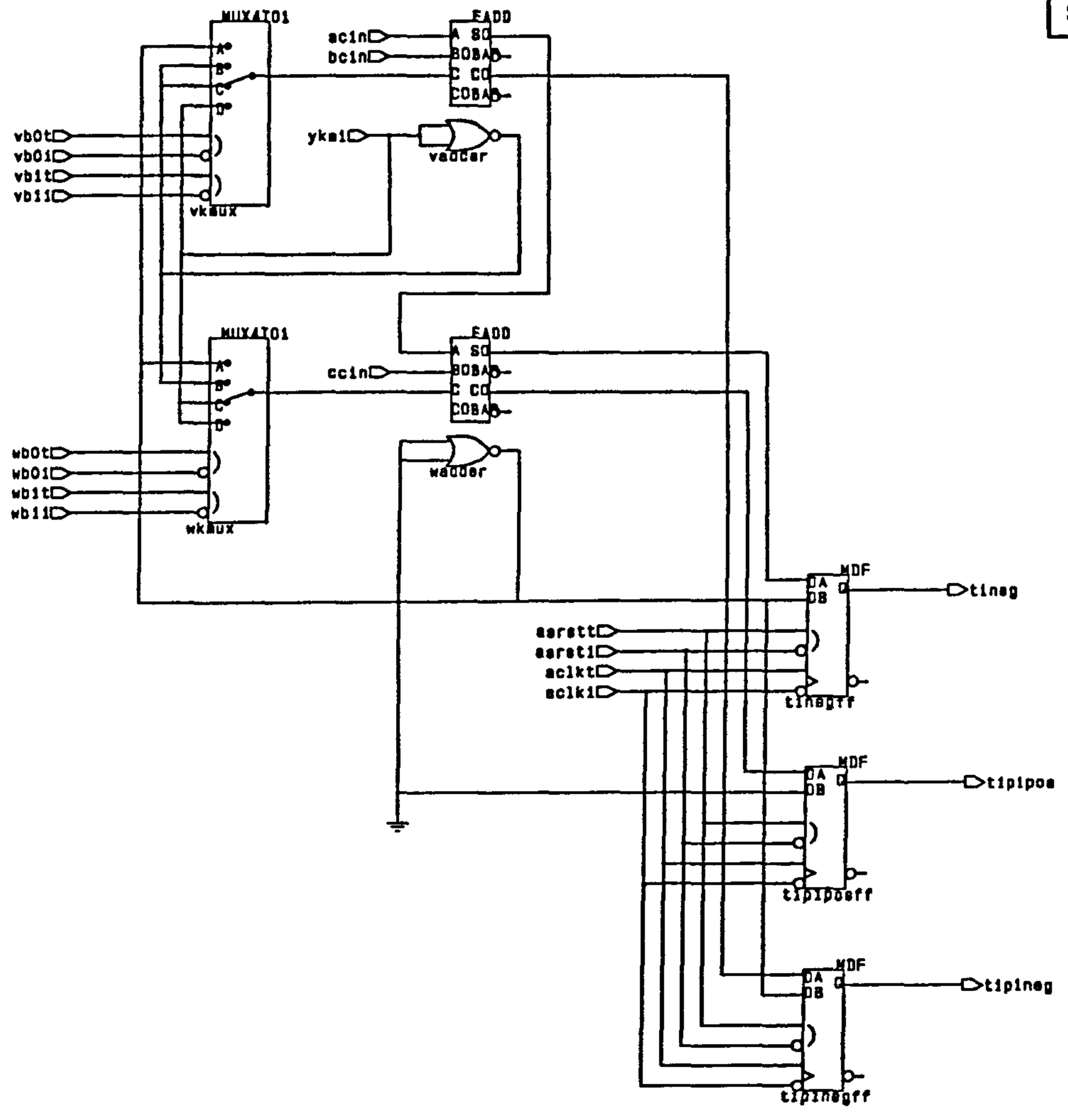
Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHiSpER 1.0
SubCct:	bs_g



Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHiSpER 1.0
SubCct:	bs_h



Author:	Paul D. Onions
Date:	June 1994
Dept:	Security Research Group
Org:	University of Plymouth
Project:	WHiSpER 1.0
SubCct:	bs_1



PAGE/PAGES EXCLUDED UNDER INSTRUCTION FROM UNIVERSITY