Hardware speedups in long integer multiplication

M. Shand*
P. Bertin†
J. Vuillemin*

Abstract

We present various experiments in Hardware/Software design tradeoffs met in speeding up long integer multiplications. This work spans over a year, with more than 12 different hardware designs tested and measured.

To implement these designs, we rely on our PAM (for *Programmable Active Memory*, see [BRV]) technology which provides us with a 50 millisecond turn-around time silicon foundry for implementing up to 50K gate logic designs fully equipped with fast local RAM and host bus interface.

First, we demonstrate how a simple hardware 512 bits integer multiplier coupled with a low end workstation host yields performance on long arithmetic superior to that of the fastest computers for which we could obtain actual benchmark figures.

Second, we specialize this hardware in order to speed-up one specific application of long integer arithmetic, namely Rivest-Shamir-Adleman public-key cryptography [RSA]. We demonstrate how a single host driving 3 differently configured PAM boards delivers RSA encryption and decryption faster than 225Kbits/sec for 512 bits keys. This beats the best currently working VLSI specially built for RSA by one order of magnitude.

1 Introduction

1.1 Hardware Acceleration

Many computationally intensive problems contain a relatively simple inner loop which performs the bulk of the computation. Speeding up this inner loop through special purpose hardware can result in dramatic performance improvements, and dedicated hardware accelerators are commonly used to boost performance on critical applications.

Perhaps the most ubiquitous is the floating-point coprocessor (FPU). Other examples are vector coprocessors, graphic coprocessors, communication units and coprocessors to accelerate lisp computations.

1.2 Limits to hardware accelerators

With the advent of the high volume commodity market for personal computers, it has become cost effective to provide more specialized accelerator cards. There are however limitations to this approach, both economic and technical.

On the technical side, the main limit to performance achievable by specialized accelerators comes from the available communication bandwidth to the host. Our approach faces that limit and most of our work went into finding appropriate tradeoffs between hardware and software processing in order to keep our application within the available bandwidth.

Economic considerations severely limit the size and number of available specialized hardware accelerators. This normally rules out applications which are used infrequently on a given host, and require significant amounts of hardware to achieve a useful speedup.

For these applications, as well as all those for which no dedicated hardware accelerator exists, one way to improve performance is to use super-computers, a very expensive proposition. Programmable hardware such as PAMs provide an economically attractive alternative to super-computers, as shown here.

Furthermore, when made to compete with specialized hardware, super-computers fare poorly: executing a given algorithm on a general purpose large structure is almost always some orders of magnitude *slower* than what can be achieved with a VLSI, whose very structure maps that of the target algorithm on a small area. This is why, in the cases reported here, we have been able to achieve performance equivalent (or superior) to those of super computers at a much lower hardware cost.

1.3 PAM technology

Using a 5×5 array of LCA (see [X]) chips, we have built a $40 \times 80 \simeq 3K$ bit PAM named Perle-0 on a 25×25 cm^2 printed circuit board. Perle-0 has a VME bus interface, which makes it a general-purpose configurable hardware co-processor tightly coupled to a host CPU (today a 16MHz MC68020).

The configuration data for Perle-0 (about 400K bits) is downloaded by the host itself in 50 milliseconds. The logic controlling

* Digital Equipment Corp., Paris Research Laboratory, 85 Av. Victor Hugo. 92500 Rueil-Malmaison, France.
† Institut National de Recherche en Informatique et Automatique, 78150, Rocquencourt, France.

the download process, as well as the host bus communication protocol are *programmed* into two extra LCAs, statically configured at power-up time from a PROM. By merely changing the content of that PROM, we are able to quickly adapt to different bus protocols, or to add extra features to the bus interface.

The PAM cycle being much faster than that of the host bus, we added 4 Megabits of fast static RAM to Perle-0, directly connected to the PAM (the bandwidth of that memory is up to 1.5 Gigabits per second, while the host bus bandwidth is typically around 50 Megabits per second). Apart from a few mandatory buffers for driving the host bus, Perle-0 is built out of just two kinds of components: LCAs and static RAM.

PAM designs are synchronous logic circuits, each of the registers being updated on each cycle of a global clock signal. The maximal clock speed for such a design is directly determined by its critical combinatorial path, which varies from one design to another. Perle-0 has a clock distribution system whose speed can be *programmed* as part of the design configuration, for speeds up to 70 MHz (the present maximum clock cycle of LCA chips).

Last but not least, we take advantage of an extra feature of the LCA component which makes it possible to dynamically *read back* the content of each Programmable Active Bit (PAB). Together with a "software stepping" facility (stop the main clock and trigger clock cycles one at a time from the host), this provides a powerful debugging tool which can take a snapshot of the internal state of the design after each clock cycle; this feature has drastically reduced the need for a-priori software simulation of our designs.

As master processor we use a Motorola 68020 based workstation with VME bus. The number of boards a host can support is limited by its available VME slots (in our current configuration 8). The accelerator boards cannot initiate bus accesses; from the host's standpoint they are *active memories*. The communication bandwidth between the host and boards is thus determined by the CPU instruction issue rate and the VME bus bandwidth. Typically, from straightline code, we can hope for at most one 32 bit transfer each 700ns. For more details see [BRV].

1.4 Long Integer Multiplication

Traditionally, floating point computations have received the greatest attention in the implementation of high performance arithmetic. As a consequence, we find in a typical microprocessor with FPU that floating point multiplications are 4 to 32 times faster than the corresponding fixed point operations. Yet, more and more applications demand exact integer arithmetic with a precision which exceeds that of current microprocessors. To answer such demands, one must implement long integer arithmetic in software. The basis for our current work is the (publicly available) BigNum software (see [SVH]), a long integer arithmetic package whose implementation on most standard microcomputers includes highly optimized assembly code for the critical arithmetic loops. Applications for this package have been found in the following areas:

- Computer arithmetic, where new ground is being gained in factoring and primality testing of very large integers.
- High level languages (such as C, Lisp, Modula, C++, Russel, Caml, ...) and specialized mathematical algebra software (such as Mapple, Macsyma, Arithmetica, ...) which need to include automatic arbitrary precision integer arithmetic at run-time.
- Signal processing, where polynomial convolution can be reduced to long integer product.
- Exact computational geometry.
- RSA cryptography, as introduced in [RSA].

We investigate hardware acceleration of long integer multiplication through PAM implemented slave processors attached to the bus of a conventional engineering workstation. Our paper considers two aspects of the utilization of such accelerators.

1. What speedups can be achieved on existing multiply intensive programs by interfacing to a general purpose multiply accelerator with no changes to the software?
2. What are the possible speedups, tradeoffs, and insights to be gained, in redesigning specific algorithms to more fully exploit several concurrently operating slave processors? How fast can we run RSA with 512 bits keys?

1.5 Main Results

To see what could be gained with minimal software change, we modified BigNum's multiply routine to use programmed hardware in lieu of the optimized multiply loop written in the native language of the host machine. By carefully matching the hardware to the bus bandwidth, we were able to produce product bits at 16 Mbit/s, faster than the best reported figures for a Cray II (see [BW]). This hardware speeds up raw multiplication by a factor 25, and Pre-existing BigNum applications can take advantage of it by simply relinking with a modified BigNum library.

To investigate further the tradeoffs that are possible in our hybrid hardware/software system we focused on one application: the RSA cryptosystem (see [RSA]) which can be cast entirely in terms of long multiplications. Starting with a version of RSA from the first part of our work, we proceeded through a series of hardware/software systems spanning two orders of magnitude in performance to our final version using three differently programmed accelerator boards, all operating in parallel with the host. At 226 Kbit/s coding and decoding speed, this system is faster than *any* currently existing 512 bits RSA implementation, in *any* technology, as of February 1990. A recent survey by Brickell [Br] grants the previous speed record for 512 bits keys RSA decryption to a VLSI from AT&T, at 19Kbits/sec.

Our contention is that PAM is a cost-effective alternative to provide significant performance improvements for computationally intensive problems on a general purpose computer system. Their reprogrammability makes it possible to consider hardware acceleration for infrequently executed applications. Their rapid turnaround and reusability encourages exploration of the vitally important hardware/software tradeoffs, and experimentation with designs that are too adventurous to commit to VLSI.

Our hardware organization provides opportunities for synchronous parallelism within each PAM and asynchronous parallelism between different PAMs and with the host itself. In the technical development to follow, we develop a strategy for programming such hybrid systems. Further, we demonstrate that thinking about hardware implementation gives insights that lead to improved software implementations. In pure performance terms, we demonstrate that PAM can compete with ASIC technologies which have much higher initial costs.

2 Hardware Multipliers

One of our first PAM designs was a 512 bit $\times$ 32 bit multiplier, based on [L]. By simply recoding the assembly inner-loop for long multiplication, we were able to interface BigNum with the PAM multiplier, and measure a maximum speedup of 25 for raw multiplication. In order to understand the impact of hardware speedup on performance at the system level, we measured three applications, previously written on top of BigNum.

1. An RSA implementation designed as a benchmark for BigNum on various computers. Our host computes 145 bits of RSA code (512 bits keys) per second on that benchmark. Using the PAM multiplier, we got 955 bits of RSA per second, a 7 fold speed-up. Hand crafting the inner loop of RSA in order to pipeline the successive multiplies pushed us to 1.8Kbit/s, a factor 12 speed-up.

2. A primality certification package, written by François Morain at INRIA. Based on Atkin's algorithm, this method is heavily addicted to long multiplies [Mor]. The PAM multiplier brought a 8 fold speed-up, and was used over 400 hours in helping Morain certify that

$$\frac{2^{3539}+1}{3},$$

   is prime. As of today, it is the largest (1065 decimal digits) known *natural*[1] prime.

3. An arbitrary precision real arithmetic package written and interfaced to BigNum by Hans Boehm (see [Bo]). Measured speed-ups on that application range from 7 for computing 5000 decimals of π, 4 for the 1000-th decimal of $sin(10)$, to 1 in the straightforward binary to decimal conversion, for printing purposes. Using an algorithm of Schönhague (see [K] p. 290) increased to 5 the speed-up on that print routine.

A problem arose from the fixed 512 bits wordlength of our multiplier. To deal with partially full words we could either pad to a multiple of 512 bits, or do in software any incomplete words. We found neither solution satisfactory. Instead, we built a second version of the multiplier which supports variable length operands: any wordlength from 64 to 512 bits in multiples of 64 can be multiplied by an arbitrarily long sequence of 32 bits words. This new multiplier computes 32 bit $\times$ 32 bit products within 2% of the performance of the native CPU multiply instruction; so there is no penalty for short multiplies, and no incentive to test and branch to choose between different implementations for different operand lengths. The driving software becomes simpler and performance more predictable.

3 Successive RSA Implementations

The essence of RSA is modular exponentiation of large integers. Modular reduction can be achieved by a fixed sequence of multiplications, using an algorithm due to Montgomery [Mon]. Thus RSA can be computed by a long sequence of multi-precision multiplications. The security of RSA depends on the difficulty of factoring the modulus used in the modular exponentiation. The size of this modulus determines the size of the numbers we must multiply. With current factoring technology 512 bits is secure, although subject to *massive* attacks such as reported in [LM]. Initially we chose to work with a 512 bit modulus as a natural fit to our existing multiplier hardware.

We discuss successive implementations of our RSA system which exhibit the characteristics listed in Table 1. Performance is indicated by the throughput of the various systems in baud—the higher the baud rate the more blocks decrypted per second.

The names we use to denote the various versions of our programmed hardware are as follows:

bus multiplier (512 bits fixed-length) Our very first multiplier.

bus multiplier The variable length version of the above multiplier, interfaced with BigNum. We call both these *bus* multipliers because for each operation, both operands, and the result, must be fed across the VME bus between the host and the PAM. The main bottleneck in these designs proves to be the VME bus bandwidth.

register multiplier This multiplier improved on the bus multiplier with a redesigned data path and 32 registers in the local RAM. However implementation constraints dictated a return to fixed size operands and a halving of word size to 256 bits.

two-bit/cycle reg. multiplier As above, but with a modified data path which, through Booth recoding, processes two bits of the multiplier per internal cycle. This and the preceding design were implemented in less than three months by two students with no previous hardware design experience (see [LV]).

mod-prod-unit This design performs a Montgomery [Mon] recoded modular multiply on 256 bits operands. It has a specially designed data path that embeds the modulus in the ALU logic. The modular multiply takes an average of 160 internal cycles.

In the table we see over 1000 times improvement between our very first software implementation and our current best accelerated implementation. The speedup comes from several sources; as we developed our successive versions we learned new techniques and improved our basic algorithms. To be fair, we worked hard on the software as well as on the hardware. Our software on more powerful machines (DecStation 5000 / MIPS R3000) approaches the speed of commercial RSA hardware. The PAM gives us a 200 times speedup over our best software on the MC68020, and a 20 times speedup over the highest performance software/machine combination we are aware of.

We started our RSA project with a simple goal: establish a new world record in RSA cryptography speed. RSA is an ideal problem for demonstrating the power of our PAMs. It is very demanding computationally and, at the same time, of sufficient commercial interest that many people have tried to build efficient implementations for it. Our own progress can be divided into four phases.

3.1 Phase 1: Algorithmic Work

We were familiar with the Montgomery technique for modular multiplication and felt it to be superior to division based methods. We began to investigate how to map it into hardware. From our

[1] A prime p is *natural* if the factorization of both $p-1$ and $p+1$ is unknown.

PROJECT PHASE	SPEED (baud)	HARDWARE CHARACTERISTICS	SOFTWARE TECHNIQUES
Algorithmic Work	*Morain's original code—full exponential—modulus size of 512*		
	145	none	software bignum (MC68020)
	1 000	bus multiplier	hardware bignum
	1 800	bus multiplier (512 bit fixed-length)	straightline code (overlapped)
	New algorithm—Chinese remainder theorem—modulus size reduced to 508		
	1 000	none	software bignum (MC68020)
	10 300	none	software bignum (MIPS R3000)
Bus Multiplier	2 200	bus multiplier @ 75ns	hardware bignum
	3 600	bus multiplier @ 75ns	optimized assembly code
	3 900	bus multiplier @ 66ns	optimized assembly code
	6 000	3 bus multipliers @ 66ns	straightline C compiled by gcc
Register Multiplier	10 000	3 register multipliers @ 75ns	naive driving code
	15 600	3 register multipliers @ 40ns	naive driving code
	22 000	3 two-bit/cycle reg. multipliers @ 40ns	naive driving code
	32 000	3 two-bit/cycle reg. multipliers @ 40ns	straightline C compiled by gcc
Modular Product Units	60 000	2 mod-prod-units @ 75ns 1 register multiplier @ 50ns	partially optimized driving code (mod-prod-units interleaved)
	78 000	2 mod-prod-units @ 75ns 1 register multiplier @ 50ns	fully interleaved straight line driving code
	92 000	2 mod-prod-units @ 55ns 1 register multiplier @ 50ns	*as above*
	145 000	2 mod-prod-units @ 38ns 1 two-bit/cycle reg. multiplier @ 50ns	*as above*
	200 000	2 mod-prod-units @ 38ns (enhanced carry completion detect) 1 two-bit/cycle reg. multiplier @ 50ns	*as above*
	226 000	*as above*	optimized I/O inlining and interleaving of subroutines

Table 1: The RSA Story

early designs we had a clear idea of the basic data path slice in our modular product unit, and one thing was apparent, it was too big. On Perle-0 we can fit a 512 bit straight multiplier; the basic cell implementing a modular multiplier is at least twice bigger. Fortunately the characteristics of RSA itself helped us there.

The computational needs for RSA encryption and decryption are asymmetric. Encryption involves exponentiation by a relatively small exponent ($2^{16}+1$ is widely used). Decryption involves exponentiation by a number which is algorithmically derived from the encryption exponent and the factors of the modulus; in general, it is as long as the modulus (512 bits in our case). Because the exponent is larger, decryption is much more time-consuming than encryption (and for us much more interesting).

Decryption is computed with knowledge of both the short (public) exponent and the longer derived (private) exponent. Under reasonable assumptions on the public exponent, this knowledge is equivalent to the factorization of the modulus. We can therefore use the Chinese Remainder Theorem (CRT, see [K] p. 270) to transform the modular exponentiation into two parallel exponentiations modulo each of the factors[2]. Typically the factors will be of roughly equal size (256 bits) allowing us to fit one such modular product unit onto our current Perle-0 PAM. Moreover CRT, by halving the exponent length and the word length leads to a fourfold performance improvement[3].

In light of these observations, we chose to concentrate on RSA decryption and to incorporate the Chinese remainder technique into our algorithms. Chinese remaindering is known in the RSA literature, and has been used in RSA software, but does not appear to have been widely used in RSA hardware. Indeed, this reluctance to use CRT is due to the complexities of implementing the algorithm in a pure hardware system. In our PAM context, conversion into and out of the Chinese remainder representation is easily handled by the host software driving an extra PAM loaded with standard BigNum hardware.

We now turn our attention to the Montgomery modular multiplication operation: the algorithm which ultimately maps to hardware. Here, in contrast to the preceding discussion on software, simplicity and regularity are of utmost importance especially when they can be achieved without sacrificing performance. We modify Montgomery's algorithm in order to eliminate a comparison against the modulus and a conditional subtraction. Its purpose is to keep the partial result strictly less than the modulus: closer analysis shows this subtract to be unnecessary. If the tests are omitted, the partial results produced in repeated application of Montgomery's algorithm never grow larger than a few times the modulus. As a consequence, we can compute with these *redundant* internal representations and reduce only once to an irredundant form during the final output stage. This redundancy requires extra space to

[2] Suppose we wish to compute $x \leftarrow a^d \bmod n$ where $n = pq$ (with p,q prime). The Chinese Remainder Theorem lets us compute this by

$$a_p, a_q \leftarrow CRT_{in}(a)$$

$$x_p \leftarrow a_p^{d_p} \bmod p \qquad x_q \leftarrow a_q^{d_q} \bmod q$$

$$x \leftarrow CRT_{out}(x_p, x_q)$$

where $a_p = a \bmod p$ and $d_p = d \bmod (p-1)$, and similarly for q.

[3] The speedup is fourfold both in hardware and software, though for different reasons. In software multiplications are four times faster; both operands are halved and we need to compute half as many multiplications since the exponent length is halved; we have two exponentiations to perform ($4 \times 2/2$). In hardware the multiplication is only twice faster, with only half the area; the exponent length is still halved and we compute the two exponentiations in parallel (2×2).

represent intermediate values. Rather than exceed 256 bits for the intermediate values, we reduced the modulus factors to 254 bits. In the main this was dictated by the word size of our register multiplier, but it was also chosen so as not to unfairly penalize pure software implementations which are naturally biased towards working with multiples of 32 bits. As a net result, our preferred modulus became 508 bits.

Importantly, these changes gave us a stable computational problem which remained essentially the same throughout all the software and hardware systems we built. As a result debugging, and comparison with previous versions was made easier.

3.2 Phase 2: Bus Multipliers

Using our improved algorithms and the bus multiplier, we can optimize driving code to make better and better utilization of the VME bus. Finally we drive three boards in parallel to fully saturate the bus with straightline code where every instruction addresses a PAM board. The boards are chained together to implement the sequence of three long multiplies required in Montgomery's algorithm. Significantly, two out of three multiplies compute the product of the result of a previous step by a value that depends only on the modulus. Thus, chaining in this manner allows us to load two of the PAMs with multipliers in which the multiplicand remains constant for the duration of an exponentiation, thereby reducing overall bus traffic. These techniques got us to 6K baud. Just a little faster than our software on a 14 MIPs workstation; just a little slower than the same software on a newer 24 MIPs machine.

Interestingly we discovered that a modern compiler produced a board driving sequence superior to the best we could craft by hand. We happily switched to describing this sequence in C rather than assembler.

3.3 Phase 3: Register Multipliers

To seriously address the bus bottleneck we must change to a register-based multiplier. Such an approach is only feasible after modifying the modular multiplication technique in to avoid the comparison described in section 3.1. This is crucial in order to use the register multiplier: with the intermediate results being stored in registers internal to the PAM, they cannot be tested by the host and operated on without extra bus cycles and a breaking of the pipeline.

Using the local storage provided by the register multipliers, we extend our previous techniques and preload local registers with various constants derived from the modulus and its factors. We now have the flexibility to store all such constants during PAM initialization and use them for the decryption of successive code blocks. In the bus multiplier the result has to appear on the VME bus and so it is natural for us to shunt it from one board to the next in the computation of an exponential, finding parallelism within each modular multiply. In the register multiplier this is no longer the case. Instead, we keep intermediate results in the same PAM and drive the two exponentiation sequences in parallel. A third PAM (also configured as a register multiplier) concurrently computes the CRT transformation of the forthcoming code block as well as the recombination of the preceding code block.

With the bus bottleneck eliminated we immediately progress to 16K baud. Using the well known Booth recoding technique, we change our internally bit serial multiplier into one which processes 2 bits on each cycle, for an overall factor 2 speedup. We are now at 32K baud and the limit lies in the PAM itself.

3.4 Phase 4: Modular Product Units

Our modular product unit maps the next level of our inner loop into hardware, and makes significantly better use of the PAMs basic resources.

In terms of PAMs we have an organization similar to that used with the register multipliers: two PAMs compute the modular exponentials and a third board does the Chinese remaindering. The modular product unit carries the idea of preloading the modulus constants one step further: we compile them directly into the logic equations of the data path at the time of design synthesis [4]. Thus our two modular product units are *genuinely different* PAM designs; indeed, each distinct RSA modulus factor yields a unique PAM design for its implementation.

With the modular product units we obtain a six-fold improvement to put us at 226K baud. The modular product unit collapses the three multiplies in our earlier implementations of Montgomery into a single operation which performs a three-way addition on each cycle; it allows to perform a multiply step and concurrent modulo reduce—hence a factor three speedup. In addition, intermediate results are stored in a highly redundant carry-save form. At the end of each modular product this gets reduced to conventional binary form (though not entirely irredundant, as discussed in section 3.1). In the worst case this may require a lengthy carry propagation through the entire 256 bit result, requiring as many cycles as the modular product itself, but in the average case just a few cycles suffice to settle the carries. We capitalize on this observation by building a carry completion detector, to gain another speed factor of two, hence our overall six-fold speedup.

4 The Host to PAM interface

To better understand the host to PAM interactions, we present here the "instruction set" for each of our hardware accelerators, and describe how each is typically used. The PAM, seen from the host, is active memory. Each PAM design is programmed to respond to a range of host addresses mapping up to 24 bits of address space. PAM instructions are encoded by the address used to access the board and the operation, either Read or Write. The data associated with an access allows the PAM to communicate a 32 bit data value to or from the host, or can be ignored in the case of an instruction which only triggers a PAM computation.

4.1 Bus Multiplier

The basic operation computed by the fixed-length bus multiplier is

$$C_{out} + P_{0\ldots n+15} \leftarrow S_{0\ldots n+15} + A_{0\ldots 15} \times B_{0\ldots n-1}.$$

Here, C_{out} is the carry out, P, S, A and B are BigNums and their subscripts refers to successive 32 bit digits. To implement this operation we build a PAM with 7 internal registers and 4 different instructions as described in Figure 1.

There are several interesting features of this interface.

[4] this fully-automatic process takes about 30 minutes.

Registers:

B_{pre}	32 bits
B	32 bits
Select$_{A/B}$	1 bit
S_{pre}	32 bits
A	512 bits
P	512 bits
P_{out}	32 bits
P_{final}	32 bits

Instructions:

WriteA $B_{pre} \leftarrow$ *MemData* , *Select*$_{A/B} \leftarrow a$

WriteB $B_{pre} \leftarrow$ *MemData* , *Select*$_{A/B} \leftarrow b$

ReadP *MemData* $\leftarrow P_{final}$

WriteS *synchronize*
$S_{pre} \leftarrow$ *MemData* , $P_{final} \leftarrow P_{out}$
if *Select*$_{A/B}$ = a then
$P \leftarrow P + S_{pre} \times 2^{512}$, $A \leftarrow A + B \times 2^{512}$
$A \leftarrow A/2^{32}$
if *Select*$_{A/B}$ = b then
$P \leftarrow P + S_{pre} \times 2^{512} + A \times B$
$P_{out} \leftarrow P \bmod 2^{32}$, $P \leftarrow P/2^{32}$

Figure 1: Bus Multiplier interface

- Only one of the instructions, WriteS, causes synchronization between the host and the PAM, and initiates computation. Each of the others operate on buffer registers and can proceed in parallel with an earlier uncompleted WriteS.
- The instruction set allows pipelining. An instruction sequence that takes full advantage of the multiplier pipelining appears in figure 2. This table represents an access sequence with time increasing from left to right and top to bottom. Each column represents a particular type of instruction. The column entries indicate the data item associated with the instruction. The left three columns contain the WriteA, WriteB and WriteS instructions that pass arguments to the board. The rightmost column contains the ReadP instructions that retrieve the results. Three pipelined and overlapped multiplies feature in the figure. The primed data values are the final stages of the preceding multiply, and the double primed data values are first stages of the following multiply. The horizontal rules mark the boundaries between these successive multiplications. Observe that the design produces a result every WriteS; there are 17 WriteS/ReadP cycles of latency between the sending first word of input data and receiving first word of the associated results.
- Since each computation is triggered by WriteS and input buffers are read non-destructively, WriteA, WriteB, and ReadP instructions can be sometimes omitted. For instance in non-pipelined use of this design a single WriteB with data 0 followed by 17 alternating WriteS and ReadP commands will flush the current result with a saving of 16 WriteB instructions. In another case, one of the three multiplies in the Montgomery operation is guaranteed to yield a result the low half of which is zero. The corresponding ReadP's can be omitted to reduce bus traffic.

The variable-length bus multiplier is a straightforward extension of the fixed-length bus multiplier. It lets the position where P is tapped into P_{out} be determined by the length of A (i.e. the number of successive WriteA's that occurred) instead of being fixed on the $0th$ bit of P as in $P_{out} \leftarrow P \bmod 2^{32}$. We have the same four instructions, WriteA, WriteB, WriteS and ReadP; however pipelining becomes more complex in the variable-length case and is left out of the current discussion.

WriteA	WriteB	WriteS	ReadP
⋮	⋮	⋮	⋮
—	$B'_{n'-1}$	$S'_{n'+15}$	$P'_{n'-2}$
A_0	—	S_0	$P'_{n'-1}$
A_1	—	S_1	$P'_{n'}$
A_2	—	S_2	$P'_{n'+1}$
⋮	⋮	⋮	⋮
A_{15}	—	S_{15}	$P'_{n'+14}$
—	B_0	S_{16}	$P'_{n'+15}$
—	B_1	S_{17}	P_0
⋮	⋮	⋮	⋮
—	B_{n-1}	S_{n+15}	P_{n-2}
A''_0	—	S''_0	P_{n-1}
A''_1	—	S''_1	P_n
A''_2	—	S''_2	P_{n+1}
⋮	⋮	⋮	⋮
A''_{15}	—	S''_{15}	P_{n+14}
—	B''_0	S''_{16}	P_{n+15}
—	B''_1	S''_{17}	P''_0

Figure 2: Bus Multiplier pipelining

4.2 Register Multiplier

The register multiplier presents the user with 32 registers, each of 256 bits. The basic operation is

$$R[p1] : R[p0] \leftarrow R[a] \times R[b] + R[s1] : R[s0]$$

where $R[n]$ refers to register n and $R[n1] : R[n0]$ denotes concatenation of registers $n1$ and $n0$ (note n0 and n1 need not be consecutive). Host access to the register contents is in terms of 32 bit aligned segments. We denote the *d*th aligned segment of $R[n]$ by $R[n]_d$. Instructions encode a register specifier in 5 bits and an aligned 32 bit segment within a register in 3 bits. Figure 3 describes the register multiplier instruction set, in this figure, primed symbols indicate operands whose values were specified by the preceding instructions.

Once again, this PAM instruction set is well suited for pipelining. The multiply command is broken into two parts: Charge provides the destination for the high 256 bits of result of the previous multiply and specifies a portion of the operands for the next multiply; Mult completes the operand specification, initiates a multiply and provides the destination for the low 256 bits of the result. For instance, to perform a 256 $\times$ 768 bit multiply, we issue the command sequence

Charge(a, $s0$, $-$)
Mult($b0$, $s1$, $p0$)
Mult($b1$, $s2$, $p1$)
Mult($b2$, $s3$, $p2$)
Charge($-$, $-$, $p3$)

Registers:

R	32×256 bits

Instructions:

WriteR(n, d)	$R[n]_d \leftarrow$ *MemData*
ReadR(n, d)	*MemData* $\leftarrow R[n]_d$
Charge($a, s0, p1$)	$R[p1] \leftarrow (R[a'] \times R[b'] + R[s1'] \times 2^{256} + R[s0'])/2^{256}$
Mult($b, s1, p0$)	$R[p0] \leftarrow (R[a] \times R[b] + R[s0]) \bmod 2^{256}$

Figure 3: Register Multiplier interface

The bus multiplier needs 97 instructions in order to compute $P \leftarrow A \times B + S$ with A, B 512 bits and P, S 1024 bits. With arguments and results in registers the register multiplier performs the same computation with only 6 instructions; this goes up to 102 instructions if all the operands must be transferred to and from the host; in all intermediate cases, say where S is zero, or A is already in a register, the register multiplier requires fewer instructions to operate than the bus multiplier.

4.3 Modular Product Unit

The Modular Product Unit (MPU) is a dedicated PAM design driven from a single application. As such it can afford a much more *intimate* interface which demands that the driving software make up for quirks that lead to a simpler hardware organization.

The MPU aims to compute

$$P \leftarrow (A \times B + Q \times M)/\beta$$

where M is the modulus, β is a power of two greater than M, and Q is *chosen* such that β divides $(A \times B + Q \times M)$ as in [Mon]. At the bit-level, this can be reduced to the following recurrence

$$\begin{array}{rcl} P_{i+1} & \leftarrow & (P_i + b_i \times A + q_i \times M)/2 \\ q_i & = & (P_i + b_i \times A) \pmod 2, \end{array} \qquad (1)$$

where b_i is the ith bit of B.

Our MPU instruction set appears in Figure 4. Let us consider some aspects of this interface which are introduced to improve performance, or to simplify the MPU implementation.

- The straightforward way to compute exponentiation is by repeated squaring and multiplies; thus our MPU supports MultB^1 and Square instructions. Precomputing B^3 speeds up the exponentiation $B^E \pmod M$ by about 20% (see [K]). Larger odd powers of B lead to further speed ups, but these are much smaller and not worth the extra hardware. This is why our MPU has an instruction MultB^3 and associated B^3 register. We pipeline the computation of B^3 in our third PAM (loaded with the register multiplier design and otherwise used for CRT).
- The speed of a direct MPU implementation of equation (1) would be severely limited by the following critical path: compute q_t from the low order bits in the data path and distribute q_t throughout the data path so it can be used in the computation of q_{t+1} during the next cycle. To achieve an aggressive cycle time we use a pipelined version of equation (1). As a consequence, our final result contains 6 more redundant bits than our multiplier based version; we use base $\beta = 2^{260}$. The host must then read back 9 words from the MPU and perform a modular reduction down to 8 words by a software table look-up.

Registers:

B^1	256 bits
B^3	256 bits
S	262 bits
A	262 bits
P	32 bits

Instructions:

WriteB(align)	$B^1 \leftarrow B^1 +$ *MemData* $\times 2^{256}$
	$B^3 \leftarrow B^3 + (B^1 \bmod 2^{32}) \times 2^{256}$
	$B^3 \leftarrow B^3/2^{32}$, $B^1 \leftarrow B^1/2^{32}$
	if align then
	$A \leftarrow 2A$
	$A, S \leftarrow A$
ReadP	*MemData* $\leftarrow P$, $P \leftarrow S \bmod 2^{32}$
	$S \leftarrow S/2^{32}$, $A \leftarrow 1$
Square	$A, S \leftarrow (A \times S + Q \times M)/2^{260}$
MultB^1	$A, S \leftarrow (A \times B^1 + Q \times M)/2^{260}$
MultB^3	$A, S \leftarrow (A \times B^3 + Q \times M)/2^{260}$

Figure 4: Modular Product Unit interface

- To simplify the MPU's controller automaton and reduce the number of cycles before which the controller can acknowledge a host ReadP operation, we make explicit in the interface an output buffer misalignment and demand that the host fix up this misalignment through an argument to the WriteB instruction.

5 Strategies for programming PAMs

Three complementary principles emerge from our experiments with the acceleration of long integer multiplication:

- Use pipelining in the host/PAM interface. This frees the host to do other tasks and opens the way to parallelism with other PAMs.
- Balance the load between the host and the accelerator, and reserve for each what each does best.
- Move control to as high a level as possible to eliminate host/PAM interaction, and simplify PAM programming.

Our BigNum experience shows that mapping the inner loop of a long multiply into hardware gives appreciable performance gains over a range of applications.

Much larger gains can be achieved for a specific application by building more specific accelerators and tailoring algorithms to the new hardware. Our highest performance designs use straightline

driving code, to get the most useful work out of the host. Vital to this is pipelining of the host/PAM commands. Idle PAMs must acknowledge the host immediately and then proceed with the requested instruction; this frees the host to issue instructions to other PAMs or execute native instructions. Thus any value returned by an access must be the result of a previously issued instruction. This result is prepared in the computation started by previous access in order to be ready for immediate return in the current access.

Excessively tight coupling between PAM and host has a drawback: if our PAM design can respond to a new command each 10μs, ancillary software computations can only be overlapped with PAM operation by carefully interleaving the two instruction sequences. This is contrary to all standard program structuring through subroutines, and we are seeking automatic solutions to this problem.

On the other hand, the high level control in algorithms accelerated through the PAM can be arbitrarily complex. The Chinese remainder theorem, which splits the modular arithmetic over the prime factors of the modulus (providing a fourfold speedup) is rarely used in ASIC implementations of RSA. It is appropriate for us because it only complicates the software component of our RSA system. Indeed for hardware it proves a great bonus as it allows parallelization of the exponentiation.

6 Critique

Rather than use hardware accelerators, perhaps is it better to use a larger and more powerful computer system? Certainly! Several months after our first multiplier, the next generation workstations became available. They gave a 3 to 5 fold performance improvement over software on the older PAM host. Moving to even larger machines however is not the answer. Supercomputers, while giving excellent vectorized floating-point performance, do not provide BigNum performance appreciably better than that of a modern RISC workstation—indeed for some operations these machines perform worse. With possibly more effort, programmable hardware accelerators can give unsurmountable performance, such as we demonstrate for RSA cryptography.

As a pleasant by product of this research on hardware designs, we have obtained a BigNum software implementation of RSA on the DecStation 5000 (with a 25MHz R3000 micro processor from MIPS) which yields 10.3Kbits/sec encryption rate on 512 bits keys. This places that machine in third position in the list of best performance RSA hardware designs compiled by [Br].

7 References

[Bo] H. J. Boehm, *Constructive Real Interpretation of Numerical Programs*, Proc. ACM conf. on Interpreters, pp. 214-221, 1987.

[Br] E.F. Brickell *A Survey of Hardware Implementations of RSA*, to appear in Proceedings of Crypto '89, Springer Verlag Lecture Notes in Computer Science, 1990.

[BRV] P. Bertin, D. Roncin , J. Vuillemin *Introduction to Programmable Active Memories*, in Systolic Array Processors edited by J. McCanny, J. McWhirter and E. Swartzlander, Prentice Hall, pp. 301-309, 1989. Also available as PRL report 3, Digital Equipment Corp., Paris Research Laboratory, 85, Av. Victor Hugo. 92563 Rueil-Malmaison Cedex, France.

[BW] D. A. Buell, R. L. Ward, *A Multiprecise Integer Arithmetic Package*, The journal of Supercomputing 3, pp. 89-107, Kluwer Academic Publishers, Boston 1989.

[K] D. E. Knuth, The Art of Computer Programming, vol. 2, Seminumerical Algorithms. Addison Wesley, 1981.

[L] R. F. Lyon *Two's complement pipeline multipliers*, IEEE Trans. Comm., COM-24: 418-425, 1976.

[LV] J.C. Las Vergnas, C. Vatinel *Implémentation d'un multiplicateur 256 bits rapide sur une carte reconfigurable. Application à la cryptographie RSA*, Digital Equipment Corp., Paris Research Laboratory, Internal Document. July 1989.

[LM] A.K. Lenstra, M.S. Manasse *Factoring by electronic mail*, in Proceedings of Eurocrypt 1989, Springer Verlag Lecture Notes in Computer Science, 1990.

[Mon] P. L. Montgomery *Modular multiplication without trial division*, Math. Comp. 44, 170, pp. 519-521, 1985.

[Mor] F. Morain *Implementation of the Atkin-Goldwasser-Killiau primality testing algorithm*, Rapport de recherche INRIA #911, October 1988.

[RSA] R. L. Rivest, A. Shamir, L. Adleman *Public key cryptography*, CACM 21, 120-126, 1979.

[SVH] B. Serpette, J. Vuillemin , J.C. Hervé *A Portable Efficient Package for Arbitrary-Precision Arithmetic* , PRL report 2, Digital Equipment Corp., Paris Research Laboratory, 85, Av. Victor Hugo. 92563 Rueil-Malmaison Cedex, France.

[X] Xilinx *The Programmable Gate Array Data Book* , Product Briefs, Xilinx, Inc., 1989.