A Ultra Fast Euclidean Division Algorithm for Prime Memory Systems

Benoît Dupont de Dinechin
C.E.A., Centre d'Études de Limeil-Valenton
94195 Villeneuve St Georges cedex France

Abstract

In this paper, we describe a general method which is suitable for efficient hardware implementation of euclidean division by numbers of the form $2^n \pm 1$. This method relies on the properties of two's complement binary arithmetic to perform simultaneous computations of the quotient and the remainder of a q-bit number in $\lceil \log_2(\lceil q \div n \rceil) \rceil + 1$ addition steps. Because it is restricted to $2^n \pm 1$ numbers, on which it operates one order of magnitude faster than the previously known constant division algorithms, our method appears to be better suited for implementation in supercomputer and image processing memory systems.

Specifically, it offers a viable alternative to the simple low-order interleaving in multi-bank memory design, as several non-linear skewing schemes no longer involve the speed nor the costs penalties that used to be associated with them. Moreover, since numbers of the form $2^n \pm 1$ include $3, 5, 7, 17, 31, 127, 257 \ldots$ which are all primes, implementations of the related prime memory systems turn out to be affordable as well.

1 Introduction

A typical parallel memory system is composed with three parts : a set of M memory banks, a routing network, and a memory controller. Each bank is an independent storage device, able to process one request every T_c cycle at the most. The routing network embeds the various paths, switches and buffers necessary to connect the banks to the memory ports, while the purpose of the memory controller is to ensure proper address and data flow through the network.

The main objective in parallel memory system design is to achieve a high bank use, in other words to allow the average memory bandwidth to approach $\frac{M}{T_c}$. Although low bank use usually results from the conjunction of several factors, the most important in a well designed memory system is related to bank conflicts. A bank conflict happens when two or more requests are addressed to the same bank within a time interval inferior to T_c cycles. When dependencies [1] hold between requests issued to a bank, the processors must be halted until the conflict is cleared.

*This research was supported at the MASI Laboratory by the Direction des Recherches, Etudes et Techniques (DRET) of the Délégation Générale pour l'Armement (DGA) under grant 86/1358.

From the designer's standpoint, the bank conflict probability is closely related to address mapping, i.e. the conversion of processor addresses into bank numbers. A popular address mapping scheme is the so-called low-order interleaving [8], where the remainder of the address by M is used to select the target bank. This organization is particularly appropriate in high performance systems since it allows accesses to consecutively stored words to be performed at full memory throughput, a useful feature when it comes to filling and saving caches or transferring I/O data. Moreover, low-order interleaving is easy to implement when M is a power of two, because the low-order bits of the address directly express the required remainder.

Unfortunately, accessing adjacent words at full bandwidth is not enough if arrays must be processed by rows, diagonals or sub-blocks as often happens in scientific computing or image processing. This has led numerous researchers to investigate the so-called skewing schemes, that is, conversions of array coordinates into bank numbers. Since many interesting skewing functions rely on such operations as integer division and remainder, an efficient means of performing them is of the highest importance.

This paper is organized in three parts. The first part reviews several results on skewing and lists some skewing functions that would benefit from improved division and modular reduction operations. In the second part, we expound the intuitive origin of our method, stating its main results and presenting a generic hardware implementation.

The last part is devoted to the design of a by-127 modular reduction / division circuit which operates on numbers between 0 and $2^{28} - 1$. The interest of the "pseudo-prime" address mappings, which can be implemented using this or similar devices, is illustrated on a Cray Y-MP computer. The mathematical proofs are developed in the appendix.

2 Background

2.1 Address Skewing

Following Shapiro [10], a skewing scheme for a $P \times P$ matrix is a function $\mu : \{0, 1, \ldots, P-1\} \times \{0, 1, \ldots, P-1\} \mapsto \{0, 1, \ldots, M-1\}$, $\mu(i, j)$ being the bank number where the matrix element A_j^i is stored. A skewing scheme is said to be periodic if $\forall i, j : \mu(i, j) = \mu(i + M, j + M)$, and is linear if

$\exists u, v : \forall i, j : \mu(i,j) = (ui + vj) \,\%\, M$[1]. Most skewing schemes likely to be encountered in literature are periodic, while linear skewing schemes are commonly used on today's supercomputers under a restricted form (more on that later).

A fundamental result from Wijshoff and van Leeuwen [12] states that if M is square-free (not divisible by the square of a prime number), any periodic skewing scheme can be reduced to a linear skewing scheme up to a permutation of the bank numbers. In addition to the μ function, the address where A^i_j is stored within bank $\mu(i,j)$ is sometimes specified, and we denote it by the expression $\alpha(i,j)$. It is usually less demanding to compute α than μ, since the bank number is the first information required for routing through the network, and because in many cases the complexity of α can be traded if necessary against some wasted space within the banks.

Non-linear skewing schemes have been proposed by van Voorhis and Morrin [11] to solve the problem of conflict-free access to vertical lines of length pq, horizontal lines of length pq, and to p by q blocks (with p, q design parameters likely to be powers of two) in the screen memory of an image processing device. The configurations studied included a $M = pq$ bank memory system with the skewing functions $\mu_2(i,j) = (iq + i \div p + j) \,\%(pq)$ and $\mu_3(i,j) = (iq + (i \div p) \,\% q + j) \,\%(pq)$. The most versatile organization turned out to be a memory system fitted with $M = pq + 1$ banks, in association with the function $\mu_4(i,j) = (iq + j) \,\%(pq+1)$.

Unfortunately, practical implementation of the latter was precluded by the necessity to perform operations modulo $(pq + 1)$, even though van Voorhis and Morrin describe in their paper a generic hardware implementation to address this problem. The need to compute addresses for the $pq + 1$ bank memory system efficiently led Park [9] to design a more economical alternative, which nevertheless requires one operation modulo $pq + 1$ to be performed for any access to a pq vertical line, a pq horizontal line, or a p by q block in memory.

2.2 Address Mapping

Although definitely useful for special purpose applications such as image or array processing, multidimensional skewing schemes are difficult to use in their most general form in high performance scientific computers. The reason is that the multi-dimensional array layout in memory is part of the definition of popular high-level language such as Fortran or C, so it cannot be altered without precluding the correct execution of many valid programs.

For instance in Fortran, matrices are supposed to be stored in column major order ; this feature is used every time a sub-array is passed to a subroutine, and also when arrays are aliased either explicitly using the equivalence statement, or implicitly through multiple definitions of a common block. In this context, any usable skewing scheme must be reducible to the composition of the language-induced array layout with the address mapping function provided by the hardware of the computer at hand. So address mapping functions, instead of general skewing schemes, must be considered while designing a supercomputer memory system.

Address mapping functions, akin to mono-dimensional skewing schemes, are simpler to compare than multi-dimensional skewing schemes. A popular criterion is the number of banks referenced by an infinite stream of requests equally spaced by an address stride s. The relevance of this measure is easy to understand, considering that equally spaced memory references account for more than 90% of the requests generated by scientific program loops [7].

Connecting strides to common access patterns in a Fortran $(I, *)$ matrix is straightforward to achieve, since the effective address of the (i,j) element takes the form $i + Ij + K$, where K is a constant related to the starting address of the array in memory. Walking through a Fortran $(I, *)$ matrix in column order yields stride one request streams, whereas accesses to the matrix following lines, diagonals and anti-diagonals respectively yield stride I, $I + 1$ and $I - 1$ request streams. These numbers must be doubled when dealing with complex or double precision data.

As stated earlier, high-performance general-purpose computers apply low-order interleaving, which can be described by the simple address mapping function $@ \mapsto @ \,\%\, M$. Combined with the column major order imposed by Fortran, the resulting skewing function for an (I, J) array is $\mu(i,j) = (i + Ij + K) \,\%\, M$. This defines a linear skewing scheme, whose behavior on various access patterns is well-known [4]. In constant stride request streams, the number of banks referenced in a low-order interleaved memory system is $\frac{M}{\gcd(M,s)}$.

One alternative to low-order interleaving has been proposed by Harper and Jump [3] with the address mapping function $@ \mapsto (@ + @ \div M) \,\%\, M$. The advantage of this scheme is that the number of banks referenced by a stride s request stream is $\min(M, \frac{M^2}{\gcd(M^2,s)})$, though it is as easy to implement as low-order interleaving when M is a power of two. Predictably, the application of this system to a number of banks which is not a power of two leads to a cumbersome implementation, involving lookup ROMs and feedback adders.

2.3 The Problem

We set down the problems of skewing and mapping as follows. When one can afford to perform integer division and modular reduction operations by constants differing from a power of two, linear skewing schemes appear to be sufficient in most cases. This is illustrated by the superiour behavior of the $pq + 1$ bank configuration of van Voorhis and Morrin. Similarly, a prime number of banks associated with simple low-order interleaving is the absolute best for supercomputers, as exemplified by the effectiveness of the prime

[1] In this text, $\div$ denotes the integer division operation, % the modular reduction operation, and / the real division operation. $\lceil x \rceil$ is the smallest integer greater than or equal to x, and $\lfloor x \rfloor$ the greatest integer less than or equal to x.

memory system on the Burroughs Scientific Processor [5].

On the other hand, when only divisions and reductions by powers of two are fast enough to meet the system objectives, a difficult choice is to be made between simple low-order interleaving and a non-linear skewing scheme. All current supercomputers rely on the first solution, so application programmers are taught to avoid strides which are multiples of two, four or eight (depending on the machine) by increasing the array dimensions [6].

The problem with non-linear skewing schemes is that they often cause annoying irregular behavior. For instance, the μ_3 skewing function of van Voorhis and Morrin allows parallel access only to p by q block starting at a line number multiple of p, while the μ_2 function extends this restriction to the pq vertical lines too. In a similar fashion, Harper and Jump's scheme appears to be unusable for large M, since for strides multiple of $M-1$, up to M requests are stacked on the same bank before the next one is addressed. Without extensive buffering, the apparent squaring of the bank number provided by this system can't be relied on (see section 4.3). Even with buffering, dependencies between requests would have to be especially loose for the system to be effective.

Concluding that an almost perfect solution would be to discover a fast way of computing quotient and remainders by constants belonging to a class embedding several odd, square-free and prime numbers, we started working on the problem. The Burroughs Scientific processor design with $M = 17$ banks, the fact that 5 and 257 are primes, and the $M = pq + 1$ image processing system of van Voorhis and Morrin with p and q powers of two, oriented our investigations towards the class of numbers expressible as $2^n + 1$. We also considered members of the class $2^n - 1$, since $3, 7, 31, 127$ are also primes, and because $p = 2^k + 1$, $q = 2^k - 1 \Rightarrow M = pq + 1 = 2^{2k}$. As we shall see, fast divisions and modular reductions by $2^n + 1$ and $2^n - 1$ are very similar, so the work in demonstrations can be almost halved.

3 The Method

3.1 Primitive equations

One may find how fast euclidean divisions by $2^n \pm 1$ can be achieved by looking for a relation between $x \div (2^n - 1)$, $x \div 2^n$ and $x \div (2^n + 1)$ for small values of x. When two $2^n \times 2^n$ matrices are filled in line major order with the quantities $\epsilon^-(x) = x \div (2^n - 1) - x \div 2^n$ and $\epsilon^+(x) = x \div 2^n - x \div (2^n + 1)$, two regular patterns appear as illustrated in figure 1 with $n = 2$. For $0 \leq x < 2^{2n} - 1, \epsilon^-(x) = 1$ if $x \div 2^n \geq 2^n - 1 - x \,\%\, 2^n$ else 0, and $\epsilon^+(x) = 1$ if $x \div 2^n > x \,\%\, 2^n$ else 0.

Thus, for the range of x we are temporarily considering, divisions by $2^n - 1$ and $2^n + 1$ differ from divisions by 2^n by 0-1 terms, which values result from simple comparisons of $x \div 2^n - x \,\%\, 2^n$ to $2^n - 1$, and of $x \div 2^n + x \,\%\, 2^n$ to 0 respectively.

Starting from these observations, the first part of our method is straightforward to devise : all we need

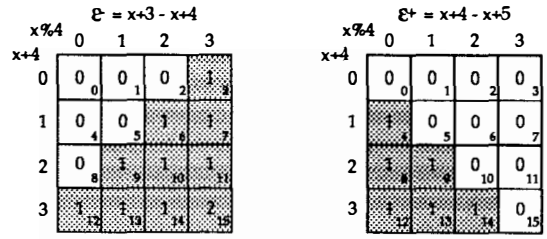

Figure 1: Relations between $x \div 2^n$ and $x \div (2^n \pm 1)$.

is to express $x \div (2^n \pm 1)$ for any x as a function of $x \div 2^n$ and $x \div 2^n \pm x \,\%\, 2^n$. From the definition of the euclidean division :

$$\begin{aligned} x &= p(x \div p) + x \,\%\, p \quad \forall x, p \in \mathbb{N} \\ \Longrightarrow x &= 2^n x \div 2^n + x \,\%\, 2^n \\ &= 2^n x \div 2^n + x \,\%\, 2^n + x \div 2^n - x \div 2^n \\ &= (2^n - 1)x \div 2^n + (x \,\%\, 2^n + x \div 2^n) \end{aligned}$$

Dividing both members by $2^n - 1$, we get :

$$\boxed{x \div (2^n - 1) = x \div 2^n + (x \,\%\, 2^n + x \div 2^n) \div (2^n - 1)}$$

Dually, $x = (2^n + 1)x \div 2^n + x \,\%\, 2^n - x \div 2^n$, so :

$$\boxed{x \div (2^n + 1) = x \div 2^n + (x \,\%\, 2^n - x \div 2^n) \div (2^n + 1)}$$

The expressions for the remainders are even simpler :

$$\begin{aligned} & x \,\%(2^n - 1) \\ &= x - (2^n - 1)(x \div (2^n - 1)) \\ &= x - (2^n - 1)(x \div 2^n + (x \,\%\, 2^n \\ &\qquad + x \div 2^n) \div (2^n - 1)) \\ &= x - (2^n - 1)(x \div 2^n) - (2^n - 1)((x \,\%\, 2^n \\ &\qquad + x \div 2^n) \div (2^n - 1)) \\ &= x - 2^n(x \div 2^n) + x \div 2^n - (2^n - 1)((x \,\%\, 2^n \\ &\qquad + x \div 2^n) \div (2^n - 1)) \\ &= x \,\%\, 2^n + x \div 2^n - (2^n - 1)((x \,\%\, 2^n \\ &\qquad + x \div 2^n) \div (2^n - 1)) \end{aligned}$$

Therefore :

$$\boxed{x \,\%(2^n - 1) = (x \,\%\, 2^n + x \div 2^n) \,\%(2^n - 1)}$$

And dually :

$$\boxed{x \,\%(2^n + 1) = (x \,\%\, 2^n - x \div 2^n) \,\%(2^n + 1)}$$

These equations are clearly recursive, and could be used directly to compute quotients and remainders by $2^n \pm 1$ in some implementations. However, considering that a linear time complexity would not be fast

enough for our purposes, we tried to achieve a logarithmic time complexity through derecursivation. This attempt succeeded, as we got the following results (the proofs are developed in the appendix) : let x be a binary number with q digits, and let $t = \lceil \frac{q}{n} \rceil$; depending on the context, and assuming that : $k \geq q \Rightarrow e_k = 0$, we shall write either :

$$\boxed{x = \sum_{k=0}^{q-1} 2^k e_k}, \text{ or : } \boxed{x = \sum_{k=0}^{nt-1} 2^k e_k}$$

Let P^-, P^+, S^- and S^+ denote the expressions :

$$\boxed{P^- = \sum_{l=0}^{t-1} \sum_{k=0}^{n-1} 2^k e_{nl+k}}$$

$$\boxed{P^+ = \sum_{l=0}^{t-1} (-1)^l \sum_{k=0}^{n-1} 2^k e_{nl+k}}$$

$$\boxed{S^- = \sum_{l=0}^{t-2} \sum_{k=0}^{n(t-l-1)-1} 2^k e_{n(l+1)+k}}$$

$$\boxed{S^+ = \sum_{l=0}^{t-2} (-1)^l \sum_{k=0}^{n(t-l-1)-1} 2^k e_{n(l+1)+k}}$$

Then we get the following results :

$$\boxed{x \,\%(2^n - 1) = P^- \%(2^n - 1)} \quad (1)$$

$$\boxed{x \,\%(2^n + 1) = P^+ \%(2^n + 1)} \quad (2)$$

$$\boxed{x \div (2^n - 1) = S^- + P^- \div (2^n - 1)} \quad (3)$$

$$\boxed{x \div (2^n + 1) = S^+ + P^+ \div (2^n + 1)} \quad (4)$$

Fast modular reductions by $2^n \pm 1$ based on the application of equations (1) and (2) has been already proposed by Yoon, Lee & Bahiri [13]. However, the implementations they describe are based on modulo $2^n \pm 1$ adders, which appear to be twice as slow as regular binary adders. Moreover they do not consider the problem of computing quotients by $2^n \pm 1$, which are also needed in a real memory system.

The implementation we present in the next section computes modular reductions by $2^n \pm 1$ in the same number of additions steps as the "fast" binary to modulo $2^n \pm 1$ translator of Yoon, Lee & Bahiri. While their implementation is based on modulo $2^n \pm 1$ adders, ours requires nothing else than ordinary adders. For this reason, we consider our euclidean division algorithm as "ultra fast".

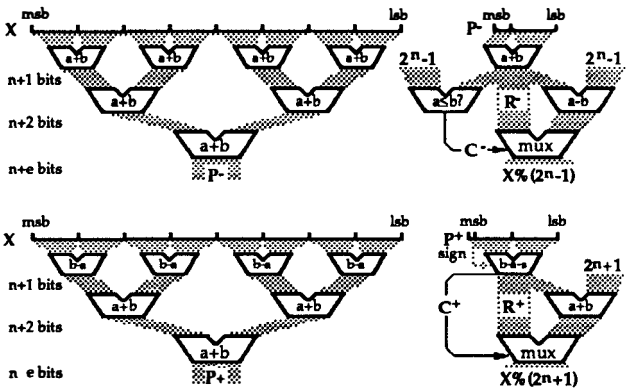

Figure 2: Modular reduction by $2^n \pm 1$.

3.2 A generic hardware implementation

Following equation 1, to compute the remainder by $2^n - 1$, one must slice the number expressed in binary notation in $\lceil \frac{q}{n} \rceil$ parts each n bits long, starting from the low-order bits, and sum them in any convenient order. The result is the partial sum P^- with $n + e$ bits, $e = \lceil \log_2(\lceil \frac{q}{n} \rceil) \rceil$, and we shall assume that P^- fits on $2n$ bits, i.e. $e \leq n$. The n low-order bits of P^- are then added to its e high-order bits, and this gives a pseudo-remainder R^- with $n + 1$ bits. If lower than $2^n - 1$, this number is the required remainder. If not, $2^n - 1$ must be subtracted from it. This effect can be achieved by using a signal C^-, set if $R^- \geq 2^n - 1$ and reset otherwise, to select between R^- and $\overline{R^-} - 2^n + 1$.

A straightforward implementation of this process is illustrated in figure 2. For it to work as expected, a few conditions must be enforced. First P^-, which lies within $[0, t(2n-1)]$ must fit on $2n$ bits, so :

$$t(2^n - 1) \leq 2^{2n} - 1 \Longrightarrow t \leq 2^n + 1$$

Second, R^- must strictly be smaller than $2(2^n - 1)$, because our implementation allows $2^n - 1$ to be subtracted at most once from R^- :

$$R^- < 2(2n-1) \Longrightarrow P^- < 2^{2n} - 1 \Longrightarrow$$

$$t(2^n - 1) < 2^{2n} - 1 \Longrightarrow t < 2^n + 1 \Longrightarrow t \leq 2^n$$

When all the conditions are met, the device depicted in figure 2 will compute remainders by $2^n - 1$ in $e + 2$ addition steps, but as we shall see section 4.2 the two last sums can eventually be merged, leading to a critical path only $e + 1$ steps long.

Similarly, to compute the remainder by $2^n + 1$ one must sum $t = \lceil \frac{q}{n} \rceil$ slices of the original number, but in this case every other term must be complemented first. All intermediate results must also be sign-extended one bit before entering the next stage in order to propagate the correct signs. The partial sum P^+, sign-extended to $2n$ bits, is halved in two n-bit slices which are then subtracted. The sign bit of P^+ is moreover

introduced as an extra borrow[2] during this operation. If the result R^+ is positive, it is the required remainder. Otherwise it must be biased by 2^n+1. Here again a signal C^+ can be defined as one if $R^+ < 0$, and used to select between R^+ and $R^+ + 2^n + 1$.

The conditions for this implementation to work properly are that P^+ must fit on $2n$ bits, i. e. $P^+ \in [-2^{2n-1}, 2^{2n-1}-1]$, and that R^+ must lie within $[-(2^n+1), 2^n+1[$. Since $P^+ \in [-\lfloor \frac{t}{2} \rfloor (2^n-1), \lceil \frac{t}{2} \rceil (2^n-1)]$, we get :

$$\left.\begin{array}{l} -2^{2n-1} \leq -\lfloor \frac{t}{2} \rfloor (2^n-1) \\ \lceil \frac{t}{2} \rceil (2^n-1) \leq 2^{2n-1}-1 \end{array}\right\} \Longrightarrow \lceil \frac{t}{2} \rceil \leq \frac{2^{2n-1}-1}{(2^n-1)}$$

On the other hand, from the way it is computed R^+ always stays within $[-2^n, 2^n-1]$, so there is no additional restriction on t. As in the 2^n-1 case, $e+2$ addition steps are apparently required to compute remainders by 2^n+1, but again some values of t and n allow the two last steps to be merged into one.

To understand why an extra borrow is needed to compute R^+, remember that any q-bit binary number in two's complement notation can be written :

$$x = \sum_{k=0}^{q-2} 2^k e_k - s\,2^{q-1} = \sum_{k=0}^{q-2} 2^k e_k + \sum_{k=q-1}^{q-1+r} 2^k s - s\,2^{q+r}$$

Applying this to $P^+ = \sum_{k=0}^{2n-2} 2^k e_k - s\,2^{2n-1} \Rightarrow P^+ = \sum_{k=0}^{2n} 2^k e^k$, with $e_{2n-1} = s$ and $e_{2n} = -s$. R^+ being produced from P^+ following equation 2, we get :

$$\begin{aligned} R^+ &= \sum_{l=0}^{2} (-1)^l \sum_{k=0}^{n-1} 2^k e_{nl+k} \\ &= \sum_{k=0}^{n-1} 2^k e_k - \sum_{k=0}^{n-1} 2^k e_{n+k} + e_{2n} \\ &= \sum_{k=0}^{n-1} 2^k e_k - \sum_{k=0}^{n-1} 2^k e_{n+k} - s \end{aligned}$$

Once the partial sums P^-, P^+ and the select signals C^-, C^+ are available, fast divisions by $2^n \pm 1$ are easy to achieve. To divide a q-bit number x by 2^n-1, one just needs to add the sum S^- of the $t-1$ terms T_k, $0 \leq k < t-1$, $t = \lceil \frac{q}{n} \rceil$ to the partial sum P^- right-shifted n bits and truncated, C^- being introduced as an incoming carry. Each term T_k is defined as the original number x right-shifted $n(k+1)$ bits and truncated. Similarly, to divide a q-bit number x by 2^n+1, the $t-1$ terms T_k must be summed with every other complemented, the result S^+ is added to P^+ itself right-shifted n bits, truncated, and sign-extended, while C^+ is introduced as a borrow into the sum.

Using the associativity of addition, the $t-1$ T_k terms can be summed in $\lceil \log_2(t-1) \rceil \leq e$ steps. Since the partial sums (P^- or P^+) themselves need e steps to be computed, the addition of S to the shifted and truncated P can be performed at the $(e+1)^{th}$ step. Simultaneously, the select signals (C^- or C^+) can be evaluated in 1 or 2 addition steps, so the quotient can be made available through selection at the $(e+1)^{th}$ or $(e+2)^{th}$ addition step, depending on the peculiar values of n and t (figure 3).

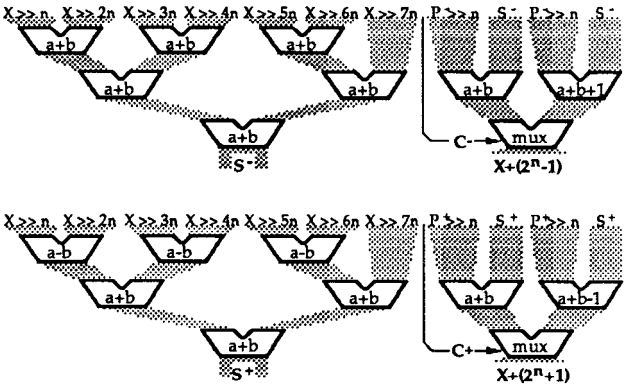

Figure 3: Division by $2^n \pm 1$.

4 Applications

4.1 Prime memory systems

Perhaps the most promising application of our fast division / modular reduction algorithm is the implementation of prime memory systems on forthcoming supercomputers. As noted earlier, low-order interleaving is the single most interesting choice for implementation in high-performance computers, because contiguous accesses account for the largest part of memory references.

Array layout in memory is bound to the definition of the high-level language used (column major order in Fortran, and row major order in C), so users must cope with linear skewing schemes they can partially tailor through adjustments of the array dimensions. The main parameter left to the computer architect is the number of banks M, but his freedom is limited, for the overall design of a memory system must be balanced according to some proven rule of thumb such as providing a theoretical bandwidth one to four times the cumulative throughput of the access channels.

To understand why and when a prime number of banks is appealing in supercomputer design, one must first agree that the primary purpose of these machines is to allow a noticeably faster execution of most scien-

[2] We understand this word with its mathematical meaning, that is, a quantity that is always subtracted from the result. In data books the borrow is complemented, i.e. it must be set for simple substractions to be performed

tific programs, before being highly efficient on a narrow class of computations.

As a first consequence, as long as address decoding stays one order of magnitude faster when M is a power of two, other solutions are not even worth considering in supercomputer design. The reason is that memory latency, which includes address decoding time, is directly involved in scalar performance. On the other hand, as soon as the speed penalty induced by the choice of an M not a power of two can be limited (as in the Burroughs Scientific processor case, where the cycle time is long, or with the availability of our fast division / modular reduction procedure), a prime number of memory banks turns out to be of the highest interest.

First, a prime number is square-free, so a wide range of skewing effects are achievable if necessary through the control of the array dimensions. A prime number of memory banks also reduces on average the frequency of recurring memory conflicts for constant-stride request streams, since problems arise only when the strides are multiples of M. But the most apparent advantage is that programmers and automatic parallelizers are relieved of the burden of avoiding strides multiples or powers of two, which occur frequently in scientific computing. This feature is especially handy when code must be transferred to a supercomputer from a scalar machine which not fast enough.

4.2 A by-127 euclidean division circuit

Returning to the numbers expressible as $2^n \pm 1$, the $2^n - 1$ class appears to offer better candidates for selection than the $2^n + 1$ class as the number of banks in the memory system of a supercomputer. The main reason is that popular interconnection schemes between the processors and the memory such as Benes and Omega networks require the number of inputs and outputs to be a power of two. If $M = 2^n + 1$, the network must be scaled to 2^{n+1}, with almost half of its routing capacity unused. On the other hand, access to $2^n - 1$ banks can be performed through a network of size 2^n only, which is twice as cost-effective.

The second advantage of fitting out a memory system with $2^n - 1$ banks is that programs tuned to run on existing supercomputers often use overdimensioned matrices to avoid strides powers of two [6], so pathological behaviors are far less likely to occur than in the $M = 2^n + 1$ case. Last, the first prime numbers belonging to the class $2^n - 1$, namely 31 and 127, fit better into the range of possible choices for the number of banks of a supercomputer memory system than 17 or 257. Facing the two remaining candidates, we chose to reduce / divide by 127 since, given an address range, higher n yield lower e and therefore allow faster implementations.

The device depicted in figure 4 operates on numbers between 0 and $2^{28} - 1$, that is, $q = 28$, and given that $n = 7$, we get $t = 4$ and $e = 2$. Since skewing usually applies at the word level, and for words are 64 bits long on supercomputers, this allows real byte addresses up to $2^{31} - 1$ to be processed. Compared to the generic implementations depicted in figures 2 and 3, the device of figure 4 is distinguished by the fact that eight-bit full adders are used everywhere, and that subterms are shared as much as possible between the quotient and the remainder chains.

Moreover the values of n and t allow the critical path to be compressed to $e + 1$ addition steps, taking advantage of the following observations :

- The subtraction of $2^n - 1$ and the comparison to $2^n - 1$ are the same operation.
- Instead of subtracting $2^n - 1$, one can add $2^n + 1$, since we are using $n + 1$-bit adders.
- The sum of $2^n + 1$ and of the e most significant bits of P^- can be merged, since bits $e-1, \ldots, 0$ and n are disjoint.

We compute R^- and $2^n + 1 + R^-$ simultaneously, C^- being generated as a by-product of the second addition. This outgoing carry is used to select the proper remainder according to figure 2, and also to increment the quotient directly.

4.3 Pseudo-Prime Memory Systems

Retrofitting an existing supercomputer with a fast by-$(2^n - 1)$ reducer / divider with $(2^n - 1)$ being prime, or including it in a new design is an especially simple matter. Let us first suppose that the original memory system is fitted out with $M = 2^n$ banks. Since it is low-order interleaved, the n low-order word address bits enter the enabling decoders, while the remaining address bits go unchanged to the banks. To convert this design into a $(2^n - 1)$-bank memory system, one just has to insert a device of the kind described section 4.2 in the address path, with R_0–R_{n-1} going to the decoders and Q_p–Q_0 directly entering the banks. In this modified memory system, nothing is changed except that the $2^{n}{}^{th}$ bank is no longer accessed, and that conflicts arise only for strides multiples of $2^n - 1$.

If $M = 2^m$ is less than 2^n, the modification is the same except that the $n-m$ high-order bits of R now go directly to the banks. The resulting address mapping function takes the form : $@ \longrightarrow (@ \,\%(2^n - 1) + 2^n @ \div (2^n - 1)) \,\%\, 2^m = (@ \,\%(2^n - 1)) \,\%\, 2^m$. We name the resulting memory systems $(2^n, 2^m)$*pseudo-prime*. As long as strides are not multiples of $2^n - 1$, requests are distributed evenly on the average, except for the last bank which is accessed $1 - 2^{m-n}$ times as often as the others. The problem with such configurations is that up to 2^{n-m} requests may be stacked on a bank before the next one is addressed. Request stacking may incur substantial performance degradations on memory systems with a small amount of buffering, so the usefulness of pseudo-prime memory systems is likely to be limited to cases with $m \simeq n$.

To evaluate the effectiveness of prime and pseudo-prime address mappings, we ran a set of experiments in dedicated time on a Cray Y-MP 8/8128 computer to measure the average memory bandwith achieved by the loop 1 shown below, with the ix array suitably initialized. The memory of the Cray Y-MP 8/8128 is divided in 128 banks. However, as far as one processor is concerned, the memory system behaves as if it had only 32 distinct banks, due to the provision of a three

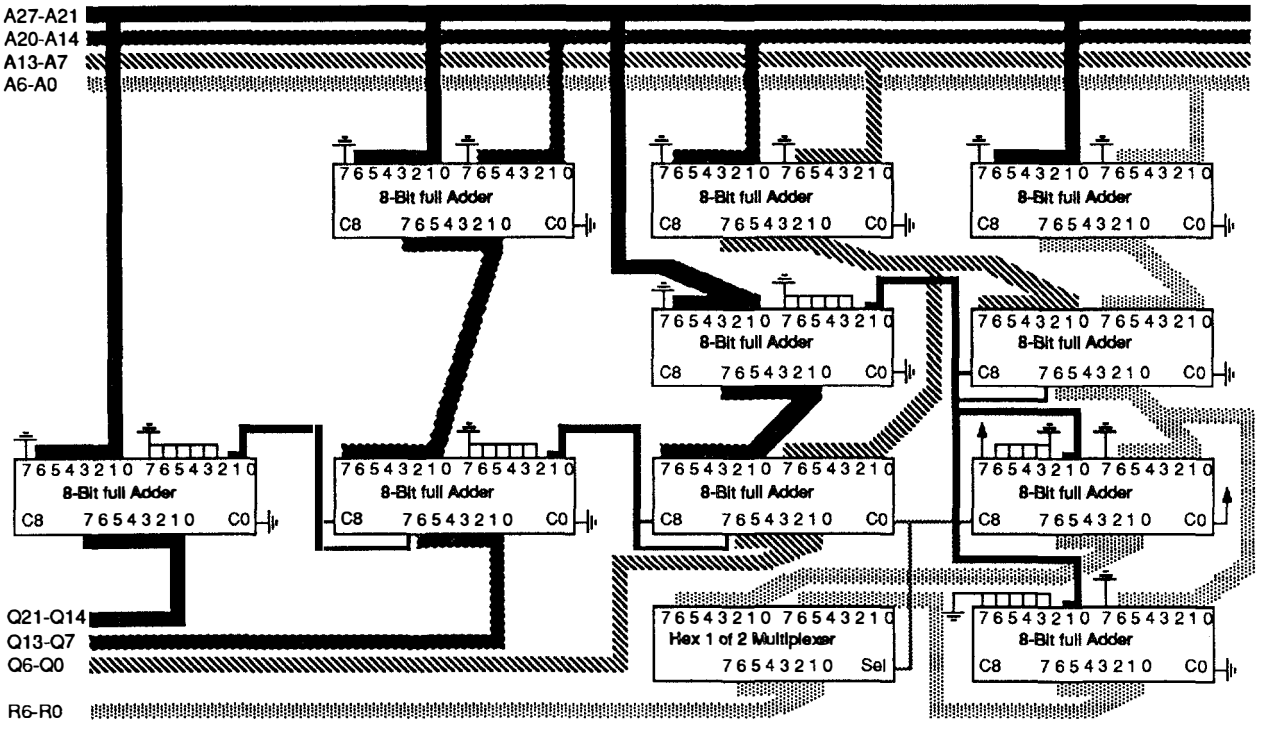

Figure 4: Fast by-127 euclidean division circuit.

layer structure which comprises 32 "subsections" per processor [2].

```
      do 1 i=1,n
         v(ix(i)) = s
1     continue
```

This can be checked in figure 5 (a), where the memory bandwidth in Mwords / second is plotted against the stride is for the simple values $ix(i) = (i \cdot is) \% n$ (here n is the size of the ix array, always set to a large multiple of the period of the address mapping function considered). In figures 5 (b) and 5 (c), the effects of $(32, 32)$ and $(128, 32)$ pseudo-prime address mappings on the memory bandwidth are measured by initializing $ix(i)$ to the values $((i \cdot is) \% 31 + 32((i \cdot is) \div 31)) \% n$, and $((i \cdot is) \% 127 + 128((i \cdot is) \div 127)) \% n$ respectively. Figure 5 (d) shows the effectiveness of the address mapping function described by Harper & Jump, i.e. $ix(i) = (i \cdot is + ((i \cdot is) \div 32)) \% n$.

As expected, request stacking prevent both the $(128, 32)$ pseudo-prime and the Harper & Jump address mappings from achieving smooth behavior. The $(32, 32)$ pseudo-prime address mapping function appears to perform wery well on the Cray, allowing a 2.4% increase of the average memory bandwidth compared to the non-skewed case. Also the behavior of the memory system appears to be much more regular, with the noticeable performance losses being restricted to the strides multiple of 31. Harper & Jump's scheme seems to perform well on these curves, but one should remember that its periodicity is 2^{2n}, that is, 1024 for a 32-bank memory system. The averaged memory bandwith of Harper & Jump's scheme on strides ranging from 1 to 1024 exposes a loss of about 1% compared to the 5 (a) case.

5 Summary and Conclusions

A fast way of computing quotients and remainders by numbers belonging to the class $2^n \pm 1$ has been presented, and the related hardware implementations described. The main interest of this technique lies in its applications to address skewing and address mapping in parallel memory systems. Although the proposed implementations are based on two input adders, the basic principles can be extended in a straigthforward manner to three input adders, allowing even faster modular reduction / division circuits to be assembled.

Acknowledgements

I wish to thank P. Feautrier, who appropriately suggested the use of modular arithmetic to shorten the proofs in the appendix. Thanks also to M. Patron and J. David, who provided assistance while experimenting with the Cray.

References

[1] D. Y. Chang, D. J. Kuck, D. H. Lawrie : "On the Effective Bandwidth of Parallel Memories",

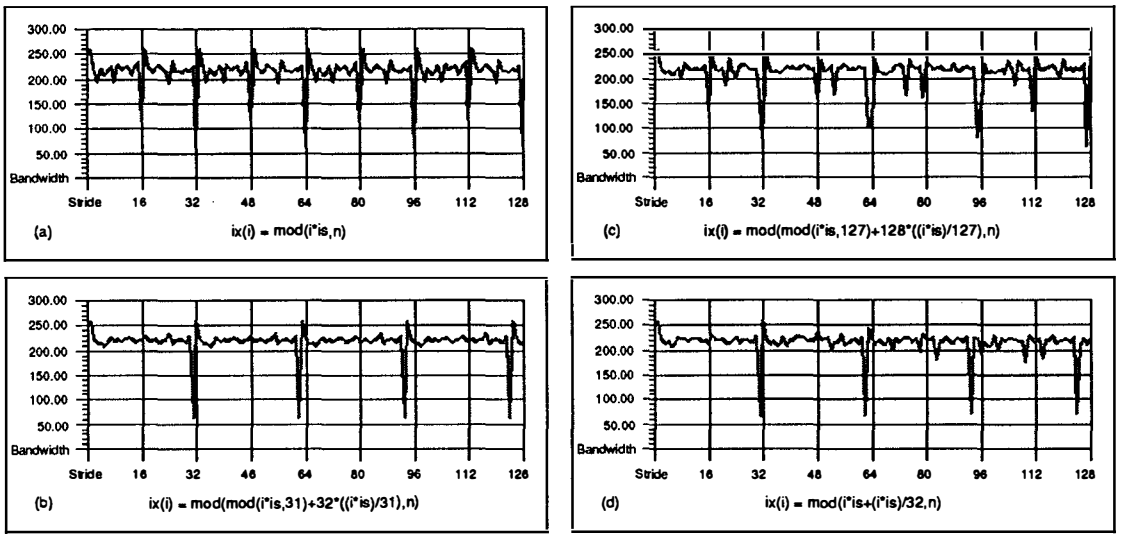

Figure 5: Effects of address mappings functions on a Cray Y-MP computer.

IEEE Transactions On Computers, Vol. C-26, N 5, May 1977.

[2] U. Deter, G. Hofemann : "Cray X-MP and Y-MP Memory Performance", Parallel Computing, Vol. 17, North-Holland 1991.

[3] D. T. Harper, J. R. Jump : "Vector Access Performance in Parallel Memories Using a Skewed Storage Scheme", IEEE Transactions On Computers, Vol. C-36, N 12, December 1987.

[4] D. H. Lawrie : "Access and Alignment of Data in an Array Computer", IEEE Transactions On Computers, Vol. C-24, N 12, Dec. 1975.

[5] D. H. Lawrie, C. R. Vora : "The Prime Memory System for Array Access", IEEE Transactions On Computers, Vol. C-31, N 5, May 1982.

[6] J. M. Levesque, J. W. Williamson : "A Guidebook to Fortran on Supercomputers", Academic Press, 1989.

[7] T. Matsuura, S. Kamiya, M. Takiuchi : "Design Concept of the FACOM VP Based on Extensive Analyses of Applications", VLSI in Computers : IEEE ICCD'84, Oct. 1984.

[8] W. Oed, O. Lange : "On the Effective Bandwidth of Interleaved Memories in Vector Processor Systems", IEEE Transactions On Computers, Vol. C-34, N 10, Oct. 1985.

[9] J. W. Park : "An Efficient Memory System for Image Processing", IEEE Transactions On Computers, Vol. C-35, N 7, July 1986.

[10] H. D. Shapiro : "Theoretical Limitations on the Efficient Use of Parallel Memories", IEEE Transactions On Computers, Vol. C-27, N 5, May 1978.

[11] D. C. van Voorhis, T. H. Morrin : "Memory Systems for Image Processing", IEEE Transactions On Computers, Vol. C-27, N 2, February 1978.

[12] H. A. G. Wijshoff, J. van Leeuwen : "The Structure of Periodic Storage Schemes for Parallel Memories", IEEE Transactions On Computers, Vol. C-34, N 6, June 1985.

[13] H. Yoon, K. Lee, A. Bahiri : "On the Modulo M Translators for the Prime Memory System", Journal of Parallel and Distributed Computing, Vol. 8, N 1, Jan. 1990.

Appendix

In this section, we shall denote the congruence relation by the $\equiv$ symbol. It is defined by :

$$\forall x, y, m \in \mathbb{Z} : x \underset{m}{\equiv} y \Longleftrightarrow \exists k \in \mathbb{Z} : x - y = km$$

We also have the following properties :

$$\left.\begin{array}{l} x \underset{m}{\equiv} a \\ y \underset{m}{\equiv} b \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} x + y \underset{m}{\equiv} a + b \\ xy \underset{m}{\equiv} ab \\ x^n \underset{m}{\equiv} a^n \ \forall n \in \mathbb{N} \end{array}\right.$$

Let x be a binary number with q digits, and let $t = \lceil \frac{q}{n} \rceil$. In the following we shall assume $x = \sum_{i=0}^{nt-1} 2^i e_i$.

Modular reduction by $2^n - 1$

$$2^n \underset{2^n-1}{\equiv} 1 \Longrightarrow 2^{nl} \underset{2^n-1}{\equiv} 1^l = 1 \Longrightarrow$$

$$2^{nl+k} = 2^{nl} 2^k \underset{2^n-1}{\equiv} 1^l 2^k \underset{2^n-1}{\equiv} 2^k \quad \forall k, l \in \mathbb{N}$$

$$x = \sum_{i=0}^{nt-1} 2^i\, e_i = \sum_{i=0}^{nt-1} 2^{n(i\div n)+i\,\%\,n}\, e_{n(i\div n)+i\,\%\,n} \Longrightarrow$$

$$x = \sum_{l=0}^{t-1}\sum_{k=0}^{n-1} 2^{nl+k}\, e_{nl+k} = \sum_{l=0}^{t-1} 2^{nl} \sum_{k=0}^{n-1} 2^k\, e_{nl+k} \Longrightarrow$$

$$x \underset{2^n-1}{\equiv} \sum_{l=0}^{t-1} 2^{nl} \sum_{k=0}^{n-1} 2^k\, e_{nl+k} \underset{2^n-1}{\equiv} \sum_{l=0}^{t-1}\sum_{k=0}^{n-1} 2^k\, e_{nl+k} = P^-$$

$$\boxed{x\,\%(2^n-1) = \left(\sum_{l=0}^{t-1}\sum_{k=0}^{n-1} 2^k\, e_{nl+k}\right)\%(2^n-1)} \quad \square$$

Modular reduction by $2^n + 1$

$$2^n \underset{2^n+1}{\equiv} -1 \Longrightarrow 2^{nl} \underset{2^n+1}{\equiv} (-1)^l \Longrightarrow$$

$$2^{nl+k} = 2^{nl}\,2^k \underset{2^n+1}{\equiv} (-1)^l\,2^k \quad \forall k, l \in \mathbb{N}$$

$$x = \sum_{i=0}^{nt-1} 2^i\, e_i = \sum_{i=0}^{nt-1} 2^{n(i\div n)+i\,\%\,n}\, e_{n(i\div n)+i\,\%\,n} \Longrightarrow$$

$$x = \sum_{l=0}^{t-1}\sum_{k=0}^{n-1} 2^{nl+k}\, e_{nl+k} = \sum_{l=0}^{t-1} 2^{nl} \sum_{k=0}^{n-1} 2^k\, e_{nl+k} \Longrightarrow$$

$$x \underset{2^n+1}{\equiv} \sum_{l=0}^{t-1} 2^{nl} \sum_{k=0}^{n-1} 2^k\, e_{nl+k} \underset{2^n+1}{\equiv} \sum_{l=0}^{t-1} (-1)^l \sum_{k=0}^{n-1} 2^k\, e_{nl+k} = P^+$$

$$\boxed{x\,\%(2^n+1) = \left(\sum_{l=0}^{t-1} (-1)^l \sum_{k=0}^{n-1} 2^k\, e_{nl+k}\right)\%(2^n+1)} \quad \square$$

Division by $2^n - 1$

$$x \underset{2^n-1}{\equiv} P^- \underset{2^n-1}{\equiv} P^- - (2^n-1)(P^- \div (2^n-1))$$

This value belongs to $[0, 2^n - 1[$, therefore :

$$x\,\%(2^n-1) = P^- - (2^n-1)(P^- \div (2^n-1))$$

From the definition of the euclidean division :

$$\begin{aligned} x &= (2^n-1)(x \div (2^n-1)) + x\,\%(2^n-1) \\ \Longrightarrow x \div (2^n-1) &= \frac{x - x\,\%(2^n-1)}{2^n-1} \\ &= \frac{x - (P^- - (2^n-1)(P^- \div (2^n-1)))}{2^n-1} \\ &= \frac{x-P^-}{2^n-1} + P^- \div (2^n-1) \end{aligned}$$

Replacing P^- by its value yields :

$$x \div (2^n-1) = \frac{x - \sum_{l=0}^{t-1}\sum_{k=0}^{n-1} 2^k\, e_{nl+k}}{2^n-1} + \left(\sum_{l=0}^{t-1}\sum_{k=0}^{n-1} 2^k\, e_{nl+k}\right) \div (2^n-1)$$

The first term S^- of this sum can be rewritten :

$$\begin{aligned} S^- &= \frac{\sum_{l=0}^{t-1} 2^{nl} \sum_{k=0}^{n-1} 2^k\, e_{nl+k} - \sum_{l=0}^{t-1}\sum_{k=0}^{n-1} 2^k\, e_{nl+k}}{2^n-1} \\ &= \frac{\sum_{l=0}^{t-1} (2^{nl}-1) \sum_{k=0}^{n-1} 2^k\, e_{nl+k}}{2^n-1} \\ &= \sum_{l=1}^{t-1} \frac{(2^n)^l - 1}{2^n-1} \sum_{k=0}^{n-1} 2^k\, e_{nl+k} \\ &= \sum_{l=1}^{t-1}\sum_{j=0}^{l-1} 2^{nj} \sum_{k=0}^{n-1} 2^k\, e_{nl+k} \end{aligned}$$

The application of the four successive changes of coordinates $u = l - j$, $v = nj + k$, $l = u - 1$ and $v = k$ leads to the expected result :

$$\begin{aligned} S^- &= \sum_{l=1}^{t-1}\sum_{j=0}^{l-1}\sum_{k=0}^{n-1} 2^{nj+k}\, e_{nl+k+nj-nj} \\ &= \sum_{l=1}^{t-1}\sum_{j=0}^{t-1-u}\sum_{k=0}^{n-1} 2^{nj+k}\, e_{nj+k+nu} \\ &= \sum_{u=1}^{t-1} \sum_{v=0}^{n(t-1-u)+n-1} 2^v\, e_{nu+v} \\ &= \sum_{l=0}^{t-2} \sum_{k=0}^{n(t-l-1)-1} 2^k\, e_{n(l+1)+k} \end{aligned}$$

And : $\boxed{x \div (2^n-1) = S^- + P^- \div (2^n-1)}$ $\square$

Division by $2^n + 1$

$$x \underset{2^n+1}{\equiv} P^+ \underset{2^n+1}{\equiv} P^+ - (2^n+1)(P^+ \div (2^n+1))$$

This value belongs to $[0, 2^n + 1[$, therefore :

$$x\,\%(2^n+1) = P^+ - (2^n+1)(P^+ \div (2^n+1))$$

From the definition of the euclidean division :

$$\begin{aligned} x &= (2^n+1)(x \div (2^n+1)) + x\,\%(2^n+1) \\ \Longrightarrow x \div (2^n+1) &= \frac{x - x\,\%(2^n+1)}{2^n+1} \\ &= \frac{x - (P^+ - (2^n+1)(P^+ \div (2^n+1)))}{2^n+1} \\ &= \frac{x-P^+}{2^n+1} + P^+ \div (2^n+1) \end{aligned}$$

Replacing P^+ by its value yields :

$$x \div (2^n+1) = \frac{x - \sum_{l=0}^{t-1}(-1)^l \sum_{k=0}^{n-1} 2^k\, e_{nl+k}}{2^n+1} + \left(\sum_{l=0}^{t-1}(-1)^l \sum_{k=0}^{n-1} 2^k\, e_{nl+k}\right) \div (2^n+1)$$

The first term S^+ of this sum can be rewritten :

$$
\begin{aligned}
S^+ &= \frac{\displaystyle\sum_{l=0}^{t-1} 2^{nl} \sum_{k=0}^{n-1} 2^k\, e_{nl+k} - \sum_{l=0}^{t-1} (-1)^l \sum_{k=0}^{n-1} 2^k\, e_{nl+k}}{2^n+1} \\
&= \frac{\displaystyle\sum_{l=0}^{t-1} (2^{nl} - (-1)^l) \sum_{k=0}^{n-1} 2^k\, e_{nl+k}}{2^n+1} \\
&= -\sum_{l=1}^{t-1} (-1)^l \frac{(-2^n)^l - 1}{-2^n - 1} \sum_{k=0}^{n-1} 2^k\, e_{nl+k} \\
&= -\sum_{l=1}^{t-1} \sum_{j=0}^{l-1} (-1)^l (-2^n)^j \sum_{k=0}^{n-1} 2^k\, e_{nl+k} \\
&= -\sum_{l=1}^{t-1} \sum_{j=0}^{l-1} (-1)^{l+j}\, 2^{nj} \sum_{k=0}^{n-1} 2^k\, e_{nl+k}
\end{aligned}
$$

The application of the four successive changes of coordinates $u = l - j$, $v = nj + k$, $l = u - 1$ and $v = k$ leads to the expected result :

$$
\begin{aligned}
S^+ &= -\sum_{l=1}^{t-1} \sum_{j=0}^{l-1} (-1)^{l-j} \sum_{k=0}^{n-1} 2^{nj+k}\, e_{nl+k+nj-nj} \\
&= -\sum_{l=1}^{t-1} \sum_{j=0}^{t-1-u} (-1)^u \sum_{k=0}^{n-1} 2^{nj+k}\, e_{nj+k+nu} \\
&= -\sum_{u=1}^{t-1} (-1)^u \sum_{v=0}^{n(t-1-u)+n-1} 2^v\, e_{nu+v} \\
&= \sum_{l=0}^{t-2} (-1)^l \sum_{k=0}^{n(t-l-1)-1} 2^k\, e_{n(l+1)+k}
\end{aligned}
$$

And : $\boxed{x \div (2^n+1) = S^+ + P^+ \div (2^n+1)}$ □