New Single Length Multiplication Semantic for The Next Generation 64-bit Processors

Fall 1991
CS252: Advanced Computer Architecture
Guillermo Maturana

by
James Hoe, c252-aa
David Chiang, c252-ac

New Single-Length Multiplication Semantic

for

The Next Generation 64-bit Processors

ABSTRACT

With the increasing acceptance of 64-bit processors, single-length multiplication has now become a viable alternative to the traditional double-length multiplication in 32-bit architectures. This paper discusses various issues regarding single-length multiplication. A new multiply instruction semantic is proposed to take advantage of convenient mathematical properties inherent to single-length multiplication. Difficulties involving condition code generation, especially overflow detection in accordance with the new multiplication instruction semantic, are presented in the paper. Modifications to support the new semantic are suggested for a few exemplary multiplier implementations. Comparisons on the basis of performance and chip real estate are made between the suggested implementations and the more traditional approach.

1. Introduction

Multiplication of two N-bit integers, signed or unsigned, will result in a product representable by 2N bits with no loss in precision. Although full precision is maintained, double-length product complicates the architecture and programming model by requiring support for an additional 2N-bit number system in an otherwise N-bit processor. Traditionally, when bounded by the somewhat limited representation power of a 32-bit integer, 32-bit processors needed to make special provisions to store this double-word product, such as the HI and LO registers in the MIPS R2000/3000 architecture. However, with the emergence of 64-bit processors, the necessity of retaining the full 128-bit double-length product becomes an issue.

1.1 Single-Length Multiplication

A single-length product in a N-bit system is the double-length product modular 2^N. In another word, the single-length product is the lower N bits of the 2N-bit full product. This single-length product has the advantage of being consistent with the processor's N-bit number system. However, since only the lower-half of the full product is retained, there will be instances where the single-length products are invalid because the true results overflow the single-length number system. But, as we will argue later, 64-bit integer will have more than sufficient range of representation for all applications that are not inherently floating-point bound. With that in mind, by exploiting the following two properties of single-length multiplication, we are able to devise a new multiplication semantic that will not only simplify the programming model but also improve multiplier performance.

1.2 Properties of Single-Length Multiplication

Property I:

When given two N-bit integers, the lower N bits of their 2N-bit product, or the single-length product, are identical regardless of whether a signed or unsigned multiplication is performed. For example, suppose in an eight-bit system, one is given A=11110101 and B=00001001. The unsigned product of A=245 and B=9 is 2205, or 00001000_10011101 in binary. The two's-complement product of A=-11 and B=9 is -99, or 11111111_10011101. Notice that the lower eight bits of both full products are "10011101" as stated in the property. Although the property does hold for this example, the single-length product is incorrect for the unsigned multiplication since 2205 is not representable in eight bits. In general, regardless whether the the single-length product is valid, the property will hold.

This property can be proven by inspection. Given 64-bit integers A and B, their numerical values depend on interpretation.

When treated as unsigned integers:

$$\mathrm{A}_{63..0} = A_{63}2^{63} + \sum_{i=0}^{62} A_i 2^i$$

$$\mathrm{B}_{63..0} = B_{63}2^{63} + \sum_{i=0}^{62} B_i 2^i$$

When treated as two's-complement integers:

$$\mathrm{A}_{63..0} = -A_{63}2^{63} + \sum_{i=0}^{62} A_i 2^i$$

$$\mathrm{B}_{63..0} = -B_{63}2^{63} + \sum_{i=0}^{62} B_i 2^i$$

Therefore, their respective products in both systems are:

$$(\mathbf{A} \times \mathbf{B})_{unsigned} = \begin{cases} A_{63}B_{63}2^{126} & (U1) \\ +\left(\sum_{i=0}^{62} A_i 2^i\right)\left(\sum_{i=0}^{62} B_i 2^i\right) & (U2) \\ +A_{63}\left(\sum_{i=0}^{62} B_i 2^{i+63}\right) & (U3) \\ +B_{63}\left(\sum_{i=0}^{62} A_i 2^{i+63}\right) & (U4) \end{cases}$$

$$(\mathbf{A} \times \mathbf{B})_{signed} = \begin{cases} A_{63}B_{63}2^{126} & (S1) \\ +\left(\sum_{i=0}^{62} A_i 2^i\right)\left(\sum_{i=0}^{62} B_i 2^i\right) & (S2) \\ -A_{63}\left(\sum_{i=0}^{62} B_i 2^{i+63}\right) & (S3) \\ -B_{63}\left(\sum_{i=0}^{62} A_i 2^{i+63}\right) & (S4) \end{cases}$$

If one treats the subtraction of S3 and S4 as the addition of the two's-complement of S3 and S4, the signed-product becomes:

$$(\mathbf{A} \times \mathbf{B})_{signed} = \begin{cases} A_{63}B_{63}2^{126} & (S1) \\ +\left(\sum_{i=0}^{62} A_i 2^i\right)\left(\sum_{i=0}^{62} B_i 2^i\right) & (S2) \\ +A_{63}\left[\left(\sum_{i=0}^{62} \overline{B}_i 2^{i+63}\right) + 2^{63}\right] & (S3') \\ +B_{63}\left[\left(\sum_{i=0}^{62} \overline{A}_i 2^{i+63}\right) + 2^{63}\right] & (S4') \end{cases}$$

One should note that the signed and unsigned products are only different in the cross-product terms (U3, U4, S3', S4'). Furthermore, only the lowest bit of the cross-product terms can affect the lower 64 bits of the product. Since S3' is the two's-complement of U3, their lowest bits will be the same. Likewise, S4' and U4 will also have the same lowest bits. Therefore, the lower 64 bits of the product, or the single-length product, is identical for both signed and unsigned multiplications.

Property II:

When given two 2N-bit integers, the lower N bits of their product are equal to the single-length product of the lower N-bit half-words of the original 2N-bit integers. Again using A and B from above, the half-words a and b are a=A[3:0]=0101 and b=B[3:0]=1001. The unsigned product of a=5 and b=9 is 45, or "0010_1101". The two's-complement product of a=5 and b=-7 is -35, or "1101_1101". The single-length product of the the half-words a and b is "1101", which is the lower four bits of the product of A and B. In this case, the single-length product is invalid in both cases due to overflow. Again, the property can be deduced by inspection.

Given 64-bit integers A and B, their numerical values in the unsigned system are:

$$A_{63..0} = 2^{32}\sum_{i=0}^{31} A_{i+32}2^i + \sum_{i=0}^{31} A_i 2^i$$

$$B_{63..0} = 2^{32}\sum_{i=0}^{31} B_{i+32}2^i + \sum_{i=0}^{31} B_i 2^i$$

And their product is:

$$(A \times B)_{unsigned} = \begin{cases} (\sum_{i=0}^{31} A_{i+32}2^i)(\sum_{i=0}^{31} B_{i+32}2^i)2^{64} & (T1) \\ + (\sum_{i=0}^{31} A_{i+32}2^i)(\sum_{i=0}^{31} B_i 2^i)2^{32} & (T2) \\ + (\sum_{i=0}^{31} A_i 2^i)(\sum_{i=0}^{31} B_{i+32}2^i)2^{32} & (T3) \\ + (\sum_{i=0}^{31} A_i 2^i)(\sum_{i=0}^{31} B_i 2^i) & (T4) \end{cases}$$

The single-length product of the lower half-words is:

$$(A_{31..0} \times B_{31..0})_{single-length} = (\sum_{i=0}^{31} A_i 2^i)(\sum_{i=0}^{31} B_i 2^i)$$

Since $T4 = (A_{31..0} \times B_{31..0})$, and only T4 contributes to the lower 32 bits of the full product, the lower 32 bits of the full product are precisely the single-length product of the

multiplication, ($A_{31..0} \times B_{31..0}$).

1.3 New Multiply Instruction Semantic

By observing the two properties above, we propose the following instruction for the next generation 64-bit architectures:

Format:

MULTi rs,rt,rd

Operation:

```
RF[rd] <= (RF[rs] * RF[rt])[63:0];

V64 <= !((signed) RF[rd] == (signed) (RF[rs] * RF[rt]))
V32 <= !((signed) RF[rd][31:0] == (signed) (RF[rs][31:0] * RF[rt][31:0]))
C64 <= !((unsigned) RF[rd] == (unsigned) (RF[rs] * RF[rt]))
C32 <= !((unsigned) RF[rd][31:0] == (unsigned) (RF[rs][31:0] * RF[rt][31:0]))
N64 <= RF[rd][63];
N32 <= RF[rd][31];
Z64 <= (RF[rd] == 0)
Z32 <= (RF[rd][31:0] == 0)
```

Description:

The contents of register rs and register rt are multiplied, treating both operands as either 64-bit two's-complement values or 64-bit unsigned values. When the operation completes, the lower 64 bits of the resulting product are placed into register rd as the single-length 64-bit result.

Eight condition bits, V64, V32, C64, C32, N64, N32, Z64 and Z32 are affected by this instruction. The V64 bit is set if the product of the two 64-bit operands, treated

as two's complement signed number, cannot be represented in a 64-bit two's-complement signed number system. The V32 bit is set if the two's-complement signed product of the lower 32 bits of the operands cannot be represented in a 32-bit two's-complement signed number system. The C64 bit is set if the product of the two 64-bit operands, treated as unsigned number, cannot be represented in a 64-bit unsigned number system. The C32 bit is set if the unsigned product of the lower 32 bits of the two 64-bit operands cannot be represented in a 32-bit unsigned number system. The N64 bit is set if the 64-bit result is negative in a two's-complement system. The N32 bit is set if the lower 32 bits of the result is negative in a two's-complement system. The Z64 bit is set if all 64 bits of the result are zero's. The Z32 bit is set if the lower 32 bits of the result are all zero's.

The single instruction defined above will replace both signed and unsigned double-length multiplications. The instruction will also indirectly remove several other instructions such as MFLO and MFHI instructions in MIPS R2000's HI-LO scheme for storing the double-word product. Furthermore, the instruction supports operations on both 64-bit and 32-bit data.

The interpretation of the 64-bit quantity produced by this instruction is usage dependent. When multiplying two 64-bit two's-complement signed numbers, if the V64 bit is not set, then the 64-bit quantity is the valid product. Otherwise, if V64 is set, the product has overflowed the 64-bit signed number system. The interpretation for 64-bit unsigned multiplication is similar. When operating on 32-bit data, the lower 32 bits of the resulting quantity are interpreted as signed or unsigned single-length product of the lower 32 bits of the operands. V32 and C32 indicate the validity of the single-length product in the same fashion as their 64-bit counterparts.

1.4 Reasons for Adopting Single-Length Multiplication

Besides simplifying the instruction set architecture, adopting this new single-length multiplication semantic also provides various other advantages such as uniformity of arithmetical operations and conservation of register resources. Unlike double-length multiplication, the architecture no longer needs to make special provisions to store the double-word result. In the HI-LO register scheme used to support double-length multiplication, further operations on the result of a multiplication is often awkward. Even if the product does not overflow into the HI register, operations on the product require special instructions to first move the product back into a general register from the special LO register. For cases where the product does overflow into the HI register, two general registers, or two consecutive general registers in some more restricting cases, are required to retain the complete product for further processing. However, when no provisions are made for operations on double-words, further processing on the double-word result is difficult without somehow transforming the result back into a single-word representation. In single-length multiplication, the awkwardness of double-length multiplication is gracefully avoided, but not without loss of precision.

Besides the advantages already mentioned above, our new semantic also has the advantage of allowing simple, uniform support for 32-bit data. Even with an abundance of 64-bit processors, 32-bit data would still remain prominent whether as remnant of the transition stage or as an alternative for conserving memory. The new semantic also allows for dramatic speed increase in multiplier implemented in CSA (carry save adder) arrays. By generating only the single-length product, no carry propagate chain is necessary to produce the upper-half of the full product.

1.5 The Validity and the Practicality of Single-Length Multiplication

All the advantages of single-length multiplication arise at the cost of precision. Therefore, one needs to question how detrimental this loss of precision is.

In the 32-bit number system, roughly 4 billion integers can be represented. As commonly noted, that is barely enough to quantify Bill Gates' current net-worth in US dollars. It is then obvious that reasonable multiplication can quickly overflow the representable range of 32-bit number system. However, a 64-bit number system has 4 billion times the representation power of a 32-bit system. To give an idea of the vastness of this range, one can specify the distance between Sun and Pluto (5910×10^6 kilometers) in microns using 64-bit integers. Similarly, one can count the age of the universe (1.8×10^{10} years) in seconds thirty-two times. In a more computing-oriented example, it will be 2575 AD by the time a 1000 mips processor finishes counting through the 64-bit number system. There is quite a bit of representation power in a 64-bit number system. We would seriously question the necessity for further expansion beyond 64-bit integer system. Undoubtedly, many scientific calculations can quickly overflow the range of 64-bit number system. However, often times, these calculations also do not require the full precision of 64 bits and are best supported by floating point operations. Therefore, it is our assertion that multiplication in realistic applications should not overflow the 64-bit number system and retaining double-length product provides little "meaningful" precision in 64-bit processors.

1.6 Application to Division

As with many things in nature, symmetry is always desirable. However in this case, a symmetrical application of our proposed semantic to division makes little sense. Firstly, division never overflows the original word length except in one case, when the

minimum negative integer is divided by -1. Therefore, the issue of single-length versus double-length does not surface. Second of all, signed and unsigned division does not exhibit similar convenient properties present in multiplication. Analogous provisions for division of 32-bit data are similarly nonsensical. The only meaningful support would be to set the C32 and V32 condition bits for the quotient. However, this is a trivial task. Since integer division always generates the quotient from the most significant bit to the least significant bit, one can determine the two conditions, C32 and V32, by observing the upper 33 bits of the quotient as they are generated. If the upper 32 bits are not all zero then C32 is set. If the upper 33 bits are not all of the same value (0's or 1's), then V32 is set. Hence, we will not discuss the topic of division in any further detail.

1.7 Statement of Problem

In the rest of the paper, we will describe the design and implementation of 64-bit single-length multipliers that will support our proposed semantic. Generating single-length product obviously does not pose any great difficulty; one simply ignores the upper-half of the full product. A simple carry-propagate shift-add multiplier, as describe in Hennessy and Patterson, always generates the full double-length product. To retrieve the single-length product, one simply retrieves the lower half of the full product. The crux of our problem lies in generating the overflow and carry condition bits.

2. Detecting Overflows and Carries in Single-Length Multiplication

Single-length multiplication will produce invalid product when the true product exceeds the range of single-length number system. Though, we argue this is a rare case, the multiplier still needs to notify the user when an invalid product is generated. The multiplier must be able to detect overflows and carries for single-length multiplications, and it needs to do this efficiently. The easiest way to detect overflow or carries is to

generate the full double-length product and examine the upper-half of the product with combinational logic, but this brute-force way is counter-productive. One advantage of single-length multiplication is the space saving and performance gain resulting from not generating the upper-half of the product. If a single-length multiplier needs to generate the full product to detect overflow, it does not have any advantage over conventional multipliers in terms of chip real estate and performance. What is preferred is an elegant way of detecting overflows without interfering with the efficient operation of a single-length multiplier. The next section will explain the mechanism by which overflow occurs in single-length multiplication.

2.1 How does Overflow Occur in Single-Length Multiplication

There are two general conditions that lead to overflow in single-length multiplication. They are best demonstrated in a long-hand multiplication. In a long-hand binary multiplication of a positive multiplicand by a positive multiplier, one scans across the multiplier from least to most significant bit. If a '1'-bit at the i-th (counting from 0) position of the multiplier is encountered, the multiplicand is multiplied by the value of that '1'-bit (2^i) to generate a partial product, and then the partial product is added to the partial sum. First source of overflow occurs when the partial product (multiplicand x 2^i) already exceeds the representability of a single-length word. Adding this already overflowed partial product to the the cumulative partial sum will definitely lead to an overflow. The other case of overflow comes from accumulation in the partial sum. Even if each individual partial product never exceeds the single-length number system, it is possible for their cumulative partial sum to overflow.

2.2 A Primitive Implementation

We can easily modify a shift-add multiplier to detect the two conditions for overflow

described above. In our modified design (see figure 1), the multiplicand and the multiplier are shifted instead of shifting the sum and the multiplier as describe in Hennessy & Patterson. To multiply two 64-bit signed-positive integers a and b, one begins by loading a and b into the A register (multiplicand) and the B register (multiplier) respectively and clearing the P register (partial sum). The overflow bit and one special flag bit whose meaning will be explained later are also cleared. Then the following steps are repeated 64 times:

1) If the least significant bit of B is '1', then the content of register A is added to the P register; otherwise, a bit sequence representing 0 is added to register P. If there is an overflow into the sign bit of the P register from this addition, an overflow bit is set. Also if the flag bit is set and the least significant bit of B is '1' then overflow bit is also set.

2) Register B is shifted right by one, while register A is left shifted by one and zero-filled. If a '1' bit is shifted into the sign-bit of A then the flag bit is set to signify that all future non-zero partial product will overflow.

At the end of the 64th step, P holds the single-length product of a and b. If the overflow bit is set, then the true product of a and b has overflowed the 64-bit system. This scheme will work for all cases where the multiplier is positive. If the multiplicand is negative, instead of looking for a '1' bit shifting into the sign-bit of A, the flag bit is set if a '0' bit is shifted into the sign-bit of A. For unsigned multiplication, the flag bit is set when a '1' is shifted out of register A, and the unsigned carry bit is set if there is a carry out of P. The actual multiplication process performed is the same in all three cases. This scheme can also be applied to setting the 32-bit carry and 32-bit overflow as required by the semantic.

However, a problem is encountered when the multiplier is negative since the the value

of a '1' bit in a negative number has a very different meaning than in a positive number. Therefore, overflow detection for the case of negative multiplier is very difficult to implement. A simpler alternative would be to guarantee that the multiplier is always positive with some extra logic. In this scheme, if only the multiplier is positive, then the multiplier and multiplicand are switched. If both the multiplier and multiplicand are negative, their two's-complements are multiplied instead, much like a sign-magnitude multiplication.

One might asked the question about why not always perform the multiplication in sign-magnitude number system and make accommodation for the sign at the end? The problem with the sign-magnitude is that the new semantic requires both unsigned carry and signed overflow to be generated for each multiplication. If sign-magnitude multiplication is performed, one would not be able to detect unsigned carry for the original operands if one of them is negative. There is another problem that exists for both the proposed solution of guaranteeing positive multiplier and the sign-magnitude solution. The new semantic also requires the multiplier to produce condition bits for both 64-bit and 32-bit number systems. If an operand has conflicting sign when interpreted in the two number systems, correct condition bits can only be generated for one of the two systems. However, one could overcome this problem by modifying the semantic to require the programmer or the compiler to properly sign extended the operands when the multiplication is intended for 32-bit data.

2.3 Necessary Improvements

Although somewhat brute-forced, the proposed multiplier is functionally correct and physically feasible. Nevertheless, this implementation is not acceptable because a 64-bit carry-propagate multiplier has no place in the high-performance architecture for which this semantic is intended. The next logical step would be to modify the implementation

for a carry-save based multiplier.

This next step turned out to be a difficult one. Recall that to detect overflow, one needs to detect overflow into the sign bit of the partial sum. With carry-propagate adder, we simply needed to compare the carry-in and carry-out of the sign-bit for each addition. If the two carry bits are different then an overflow has occurred. However, there is no known way of detecting signed overflow in a carry-save-adder whose actual sum is stored as both sum and carry during intermediate stages of multiplication. If no efficient way of detecting overflow in CSA could be devised, single-length multipliers would be limited to CPA based multipliers. This would seriously limit the performance of single-length multipliers and eliminate any practicality and advantage that single-length multipliers have. This is a problem that must be overcome.

Realizing our lack of formal mathematical training, we sought the help of Professor Hendrik Lenstra to resolve this bottleneck. We were fruitless after hours of discussion with Professor Lenstra on the problem of detecting overflow in a CSA. However, when the problem is brought up again in the context of multiplication, Professor Lenstra gave us an important hint that is crucial to the success of this project. There are two conditions that lead to overflows in single-length multiplication, but these two conditions are not mutually exclusive. In most cases when the partial sum overflows, the partial product would also have overflowed. Though one may not be able to detect when the partial sum overflows, if one can successfully detect all overflows resulting from overflowing partial products, one would also have detected most of the cases when the partial sums have overflowed. Thus, the remaining undetected cases of overflows result solely from the accumulation of non-overflowing partial products. In this case, the overflow never propagates pass the sign bit. Therefore, if all the overflows resulting from overflowed partial product is correctly detected, the remaining overflows can be detected by checking for the correct sign-bit at the end of the multiplication. With the

help of Professor Lenstra, we developed the following three parts theorem.

2.4 Theorem for Overflow Detection in a Single-length Multiplication

Part a:

Suppose a, b, p ε N,
Let k and l be the number of the leading zero's in a and b respectively,
We then define a, b, and p as the following:

$ab \equiv p \mod 2^{64}$

$0 < a < 2^{63}$ a = (0000...001****) $(1 \le k < 64)$

$0 < b < 2^{63}$ b = (0000...001****) $(1 \le l < 64)$

$-2^{63} \le p < 2^{63}$

Since a has k leading zero's, $2^{63-k} \le a < 2^{64-k}$
Since b has l leading zero's, $2^{63-l} \le b < 2^{64-l}$

Therefore, from combining the previous two equations one concludes:

$$2^{126-k-l} < |ab| < 2^{128-k-l}$$

Theorem:

1) If $k + l \le 63$, then:
$ab \ne p$

2) If $k + l \ge 64$, then:
$ab = p$ iff $p > 0$

Proof of the theorem:

1) $|ab| \ge 2^{126-(K+l)}$
$\ge 2^{63}$

ab $\ge 2^{63} > p$

Hence, $ab \ne p$

2) Suppose p = ab, then p > 0 {done}

Suppose p > 0,
Since $|ab| < 2^{128-(k+l)}$
$|ab| < 2^{64}$,
Therefore,
$0 < ab < 2^{64}$,
Also, $-2^{63} \le p < 2^{63}$,
Therefore,
$0 < p < 2^{64}$,
Hence,

$$|ab-p| = \begin{cases} < 2^{64} \\ \equiv 0 \mod 2^{64} \end{cases}$$

$|ab-p| = 0$
ab = p {done}

Part b:

Suppose a, b, p ε N,
Let k be the number of the leading zero's in a
and l be the number of the leading one's in b respectively,
We then define a, b, and p as the following:

$ab \equiv p \ mod \ 2^{64}$

$0 < a < 2^{63}$ a = (0000...001****) (1 ≤ k <64)
<- 64 ->
<- k ->

$-2^{63} \le b < -1$ b = (1111...110****) (1 ≤ l <64)
<- 64 ->
<- l ->

$-2^{63} \le p < 2^{63}$

Since a has k leading zero's, $2^{63-k} \le a < 2^{64-k}$
Since b has l leading one's, $-2^{64-l} \le b < -2^{63-l}$

Therefore, from combining the previous two equations one concludes:

$$2^{126-k-l} < |ab| < 2^{128-k-l}$$

Theorem:

1) If $k + l \le 63$, then:
$ab \ne p$

2) If $k + l \ge 64$, then:
$ab = p$ iff $p < 0$

Proof of the theorem:

1) $|ab| > 2^{126-(k+l)}$
$> 2^{63}$

ab $< -2^{63} \le p$

Hence, $ab \ne p$

2) Suppose p = ab, then p < 0 {done}

Suppose p < 0,
Since $|ab| < 2^{128-(k+l)}$
$|ab| < 2^{64}$,
Therefore,
$-2^{64} < ab < 0$,
Also, $-2^{63} \le p < 2^{63}$,
Therefore,
$-2^{64} < p < 0$,
Hence,

$$|ab-p| = \begin{cases} < 2^{64} \\ \equiv 0 \ mod \ 2^{64} \end{cases}$$

$|ab-p| = 0$
ab = p {done}

Part c:

Suppose a, b, p ε N,
Let k and l be the number of the leading one's in a and b respectively,
We then define a, b, and p as the following:

$ab \equiv p \pmod{2^{64}}$

$-2^{63} \le a < -1$ a = (1111...110****) $(1 \le k < 64)$
<- k ->

$-2^{63} \le b < -1$ b = (1111...110****) $(1 \le l < 64)$
<- l ->

$-2^{63} \le p < 2^{63}$

Since a has k leading one's, $-2^{64-k} \le a < -2^{63-k}$
Since b has l leading one's, $-2^{64-l} \le b < -2^{63-l}$

Therefore, from combining the previous two equations one concludes:

$$2^{126-k-l} < |ab| \le 2^{128-k-l}$$

Theorem:

1) If $k + l \le 63$, then:
$ab \ne p$

2) If $k + l \ge 64$ and $ab \ne 2^{64}$, then:
$ab = p$ iff $p > 0$

Proof of the theorem:

1) $|ab| > 2^{126-(k+l)}$
$> 2^{63}$

$ab > 2^{63} > p$

Hence, $ab \ne p$

2) Suppose p = ab, then p > 0 {done}

Suppose p > 0,
Since $|ab| \le 2^{128-(k+l)}$
$|ab| < 2^{64}$ (excluding $ab = 2^{64}$),
Therefore,
$0 < ab < 2^{64}$,
Also, $-2^{63} \le p < 2^{63}$,
Therefore,
$0 < p < 2^{64}$,
Hence,

$$|ab - p| = \begin{cases} < 2^{64} \\ \equiv 0 \pmod{2^{64}} \end{cases}$$

$|ab - p| = 0$
ab = p {done}

To restate the theorem plainly, overflows resulting from overflowed partial products can be detected by keeping track of the leading bit patterns of both the multiplicand and the multiplier. To detect overflow completely, one simply needs to also check the single-length product for the correct sign-bit. If no overflowing partial product has been detected and the sign-bit is correct, the single-length product is valid. However, if one is careful, one will notice that the theorem left some boundary cases by limiting k and l to be less than 64 for mathematical reasons. We will try to resolve these boundary conditions now.

In the case of multiply a positive number by another positive number, the theorem left out the trivial case of multiplying by zero's, but no special provision needs to be made here. When multiplying by zero, k+l will be at least 64, and the resulting product, zero, will have the right sign. The theorem will correctly determine overflow.

In the case of multiplying a positive number by a negative number, -1 and 0 are not accounted for. Again, no special provision needs to be made for multiplying by -1. Multiplying -1 by any positive number will not cause an overflow and the theorem will behave correctly. Multiplying a negative number by zero does lead to a special case. An algorithm based purely on the theorem would detect an overflow because zero is technically a positive number while the multiplication of a positive number and a negative number should produce a negative product. Therefore, in a complete algorithm, we need to over-ride overflow generation whenever one of the operands is zero because multiplication involving zero should never cause an overflow.

In the final case of multiplying negative by negative, -1 is left out. Multiplying -1 by another negative number does not cause a problem except when the other number is the minimum negative number (-2^{63}). Such a case results in a product of $+2^{63}$. 2^{63} overflows the 64 bit number system. However, since the resulting 2^{63} looks just like

-2^{63} in a 64-bit two's-complement interpretation, a standard algorithm would see the wrong sign bit and detect the overflow. The negative by negative case has one other hidden problem. Even when all partial products are valid, the partial sum can reach up to 2^{64} which would give a single-length product of zero. A standard algorithm that only looks for correct sign-bit of the final product would not catch this overflow. Therefore, one need to make provision to generate an overflow when the product is zero but both operands are negative.

Overall, two special provisions need to be added to the basic theorem to form a complete overflow detection scheme for single-length multiplication. This complete algorithm also applies to 32-bit single-length multiplication described by the new semantic; one simply asserts that k+l starting at the 32-bit boundary should be greater than 32 rather than 64. Unsigned carry can also be detected with this algorithm if one interprets the 64-bit unsigned system as a 65-bit signed system with an implied '0' sign-bit.

The greatest advantage of this overflow detection scheme is that it is completely operation independent. The detection scheme only requires examining the leading bit patterns of the operands and the sign of the final product. It is conceivable that a separate overflow detection circuit can be designed to fit any of the existing multiplier designs without degrading the performance of the multipliers. With some designs, optimization could be made to blend the detection circuitry into the multiplier circuit to conserve chip area. Provided in the next section is a optimized radix-16 carry-save shift-add multiplier and a sample array multiplier.

3. Sample Implementations and Comparisons

3.1 Radix-16 Carry-Save Shift-Add Multiplier

This sample implementation is based on a carry-save shift-add multiplier, figure 2. The multiplier operates on radix-16 so only 16 cycles are required to perform a 64-bit multiplication. The overflow and carry detections are circuit realization of our theorem and the two special provisions. On figure 3, we see the detection logic for unsigned carries. Most of the logic are there to detect overflowed partial product or $(k+l) \leq 63$. The overflow partial product logic is an extension to the one used in the primitive CPA shift-add multiplier. The added complexity results from the fact that the multiplier is now radix-16. On figure 4, the overflow detection logic is shown. The logic is similar to the unsigned-carry detection except the extra logics for the boundary cases. Also, the inputs to the detection logic pass through xor gates controlled by the sign-bits of the two operands; this allows the same detection logic to be used for all four combinations of signed operands. There is a large logic network for zero detections. Despite of the appearance, the actual propagation delay is 2 gate-delays because when Z32 signal is needed at the end of the eighth iteration, only P[31:29] are changing while P[28:0] remain stable and valid from previous cycles. The same applies to Z64. This zero detection logic is probably the most costly part of the overflow detection logic in terms of chip area. The overflow detection could share the existing zero-detect logic in the ALU if an extra cycle can be spared.

In this implementation, the condition code generation mechanism actually blends itself into the multiplier. However, the addition of the condition code generation logic does not introduce any new critical delay path to the multiplier. Therefore, the inherent performance of a CSA shift-add multiplier is not affected. This particular design does not have a significant space advantage over a well-designed CSA shift-add double-

length multiplier. However, unlike a double-length multiplier, a time-consuming 64-bit carry-propagate addition is eliminated because this CSA multiplier is designed for single-length only.

3.2 CSA Array Single-Length Multiplier

A CSA array single-length multiplier is presented in figure 5. To simplify the schematic, the multiplier only supports 8-bit signed number system. With additional logic, the proposed design can be easily expanded to support the full semantic for 64-bit and 32-bit multiplications. Very much like unrolling loops in programs, the overflow detection logic for an array multiplier is simply the unrolled version of the detection logic presented for the CSA shift-add multiplier. Again, the detection logic does not interfere with the operation of the array multiplier or create any new critical delay path. In fact, The proposed multiplier array could be replaced by any other conceivable array organization, such as the various tree multipliers, without requiring major modifications to the overflow detection logic.

Once more, the single-length CSA array multiplier has a performance edge over double-length multipliers of the same design because generation of the upper product with a CPA adder is not required. However, more importantly, a single-length array multiplier holds another significant advantage over a double-length array multiplier in VLSI implementation. A double-length multiplier would require roughly twice the chip area to implement. Furthermore, because of the skewed physical organization of a double-length multiplier, great difficulty exists in fitting a double-length array multiplier into a 64-bit datapath. Whereas, a single-length array multiplier would blend smoothly into the datapath because it is also only 64 bits wide.

4. Provision for Double-Length and Extended-Precision Multiply

Before concluding our report, we must address the problem of what happens in some rare but conceivable situations when 128-bit double-length product is absolutely necessary. For single-length multiplication operation to be complete, we must be able to generate full double-length product when required. Included below are the sample assembly codes for both signed and unsigned double-length multiplication. With the algorithm for double-length multiplication, any extended-precision multiplication can be achieved with conventional algorithms.

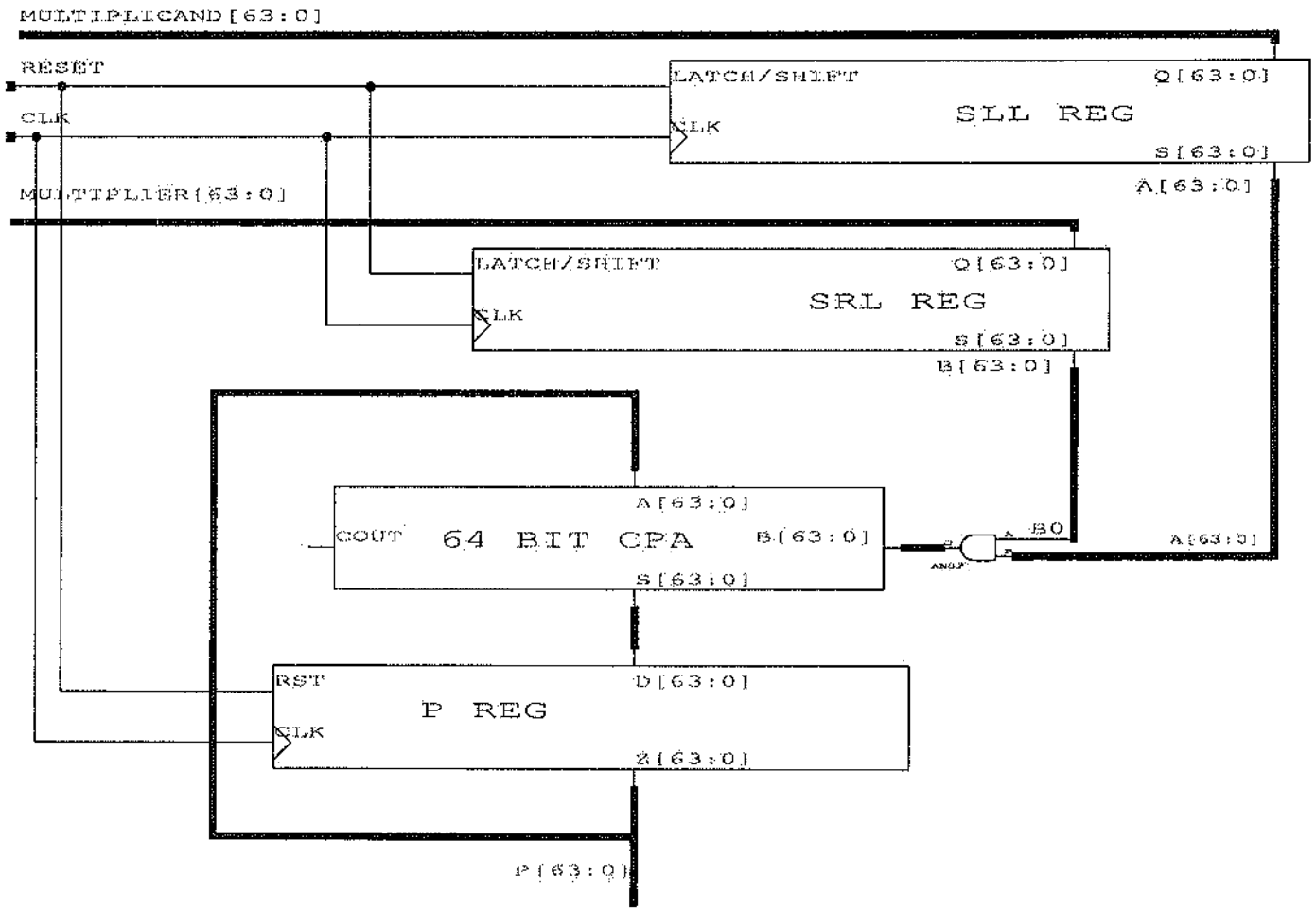
```
#
# signed double-length multiplication
# give temporary register t1, t2, t3
# r1 x r2 ==> r4,r3
#
d_mult_s:
        sll r1,32,t1
        srl t1,32,t1        # lower half r1 ==> t1
        sll r2,32,t2
        srl t2,32,t2        # lower half r2 ==> t2
        multi t1,t2,r3      # lower r1 x lower r2 ==> r3
        sra r2,32,t3        # signed upper half of r2 ==> t3
        multi t1,t3,r4      # lower r1 x upper r2 ==> r4
        sll r4,32,t1
        addcc t1,r3,r3      # add and modify carry
        sra r4,32,r4
        sra r1,32,t1        # signed upper half of r1 ==> t1
        multi t1,t3,t3      # upper r1 x upper r1
        addx t3,r4,r4       # add with carry
        multi t1,t2,t1      # upper r1 x lower r2
        sll t1,32,t2
        addcc r3,t2,r3      # add and modify carry
        sra t1,32,t1
        addx t1,r4,r4       # add with carry
        retl

#
# unsigned double-length multiplication
# give temporary register t1, t2, t3
# r1 x r2 ==> r4,r3
#
d_mult_u:
        sll r1,32,t1
        srl t1,32,r1        # lower half r1 ==> t1
        sll r2,32,t2
```

```
srl t2,32,t2        # lower half r2 ==> t2
multi t1,t2,r3      # lower r1 x lower r2 ==> r3
srl r2,32,t3        # upper half of r2 ==> t3
multi t1,t3,r4      # lower r1 x upper r2 ==> r4
sll r4,32,t1
addcc t1,r3,r3      # add and modify carry
srl r4,32,r4
srl r1,32,t1        # upper half of r1 ==> t1
multi t1,t3,t3      # upper r1 x upper r1
addx t3,r4,r4       # add with carry
multi t1,t2,t1      # upper r1 x lower r2
sll t1,32,t2
addcc r3,t2,r3      # add and modify carry
srl t1,32,t1
addx t1,r4,r4       # add with carry
retl
```

As Amdahl's law dictates, frequently used cases should be as fast as possible, and performance of rare cases can be and should be sacrificed for the general cases. Though it costs significantly more cycle time to generate double-length by software, it is not impossible. As long as double-length multication is infrequent, this will not degrade the overall performance of the processor.

5. Closing

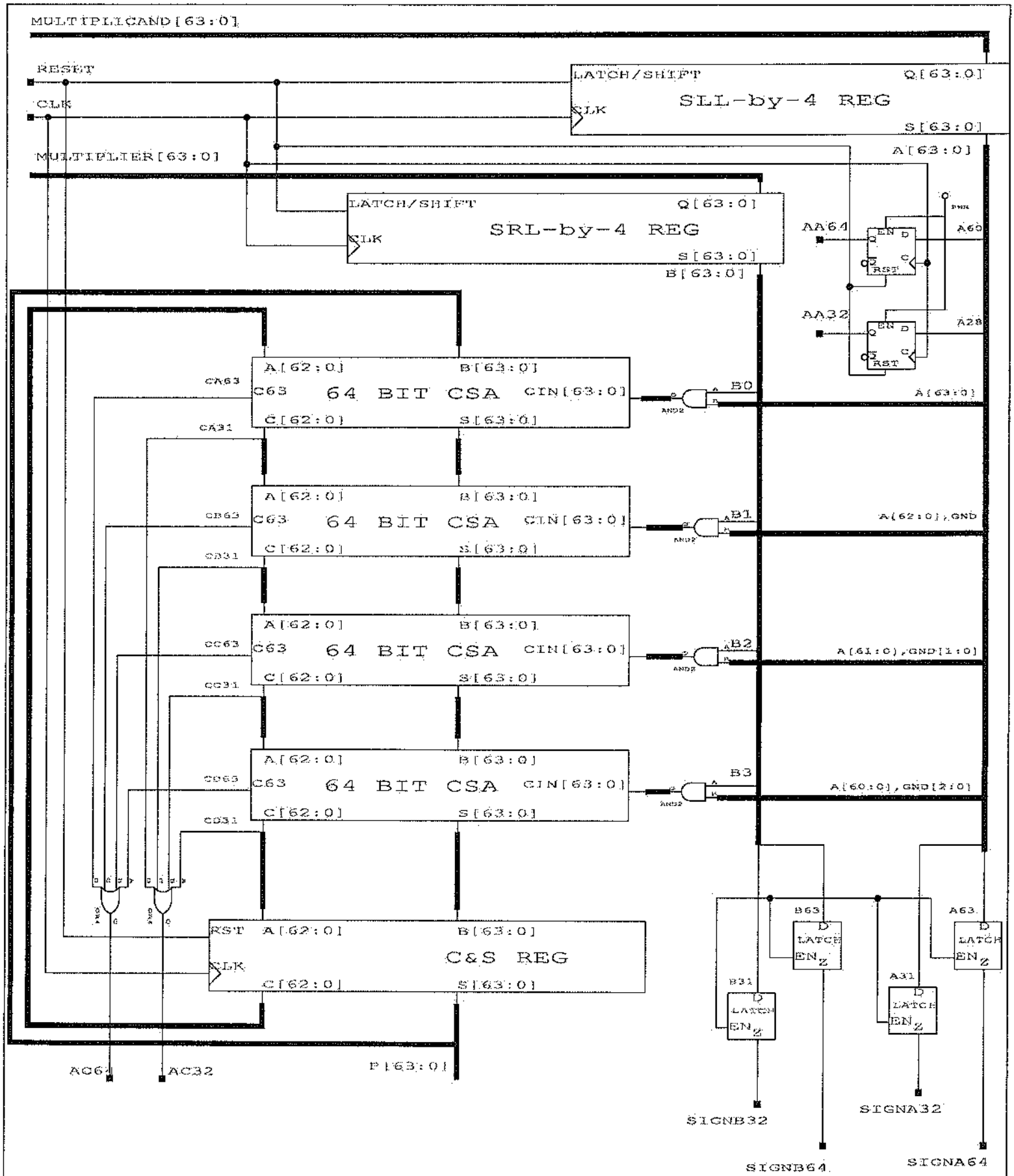
As we have shown, it is possible to inexpensively and efficiently modify many existing high performance multiplier design to comply to our new single-length multiplication semantic. In most cases, we are not only able to conserve significant chip real estate but also achieve valuable performance gain at the same time. If 64-bit integers indeed have enough expressive power for realistic applications, this new multiply semantic can bring about revolutionary changes to both future 64-bit processors' instruction set architectures and arithmetic unit designs.



Simple CPA SHIFT-ADD

SINGLE-LENGTH MULTIPLIER

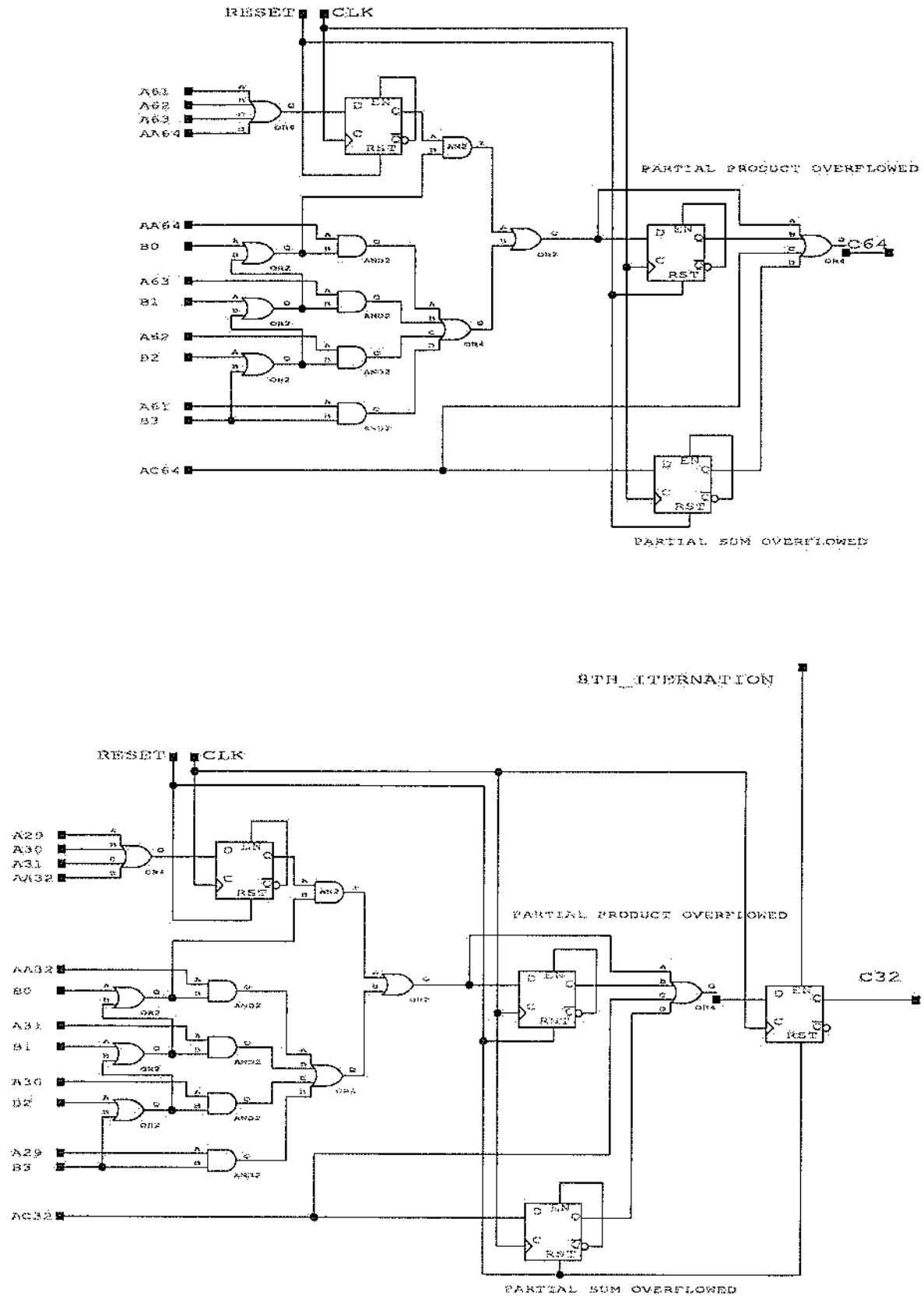
Figure 1



RADIX-16 CSA SHIFT-ADD

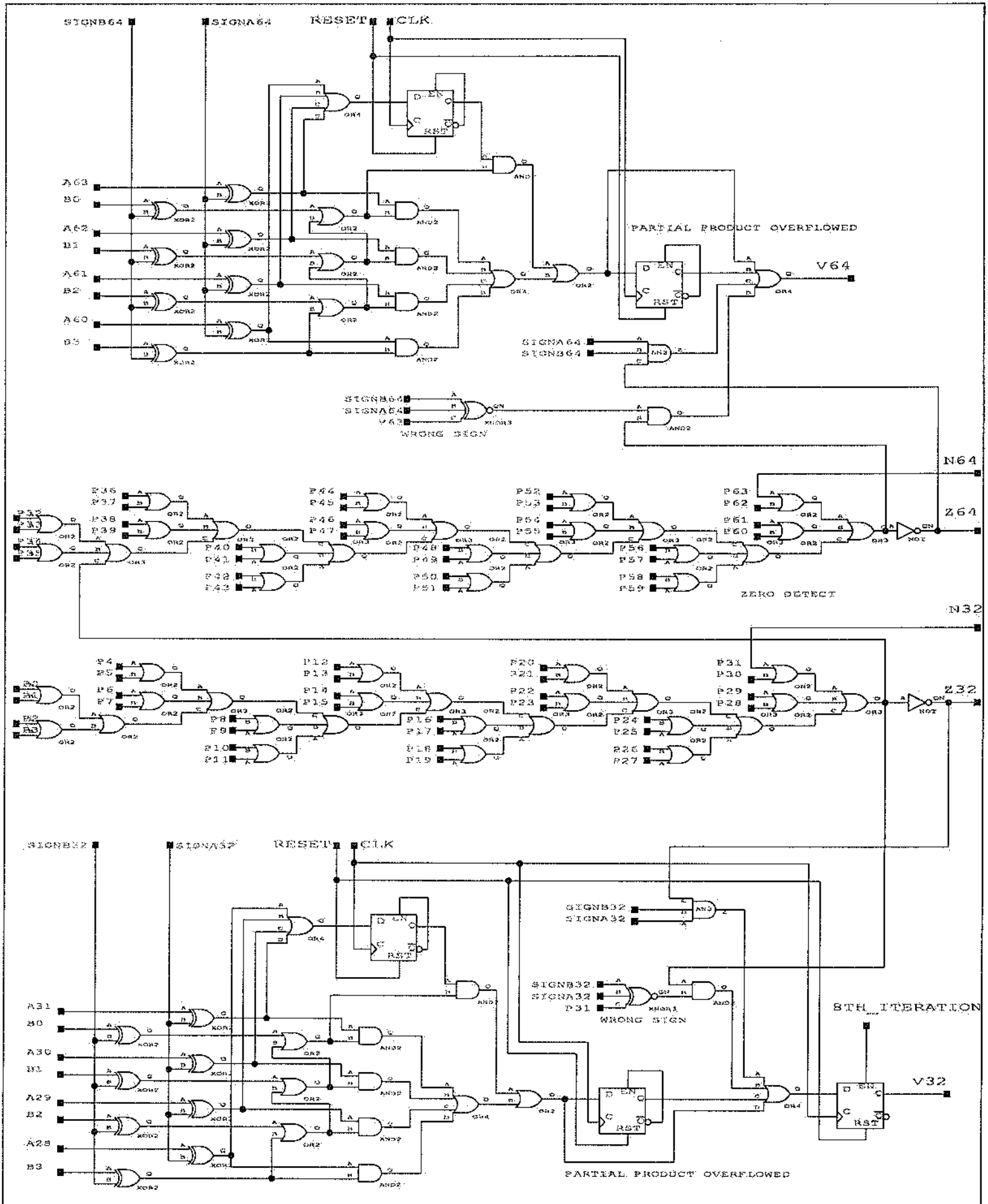
SINGLE-LENGTH MULTIPLIER

Figure 2



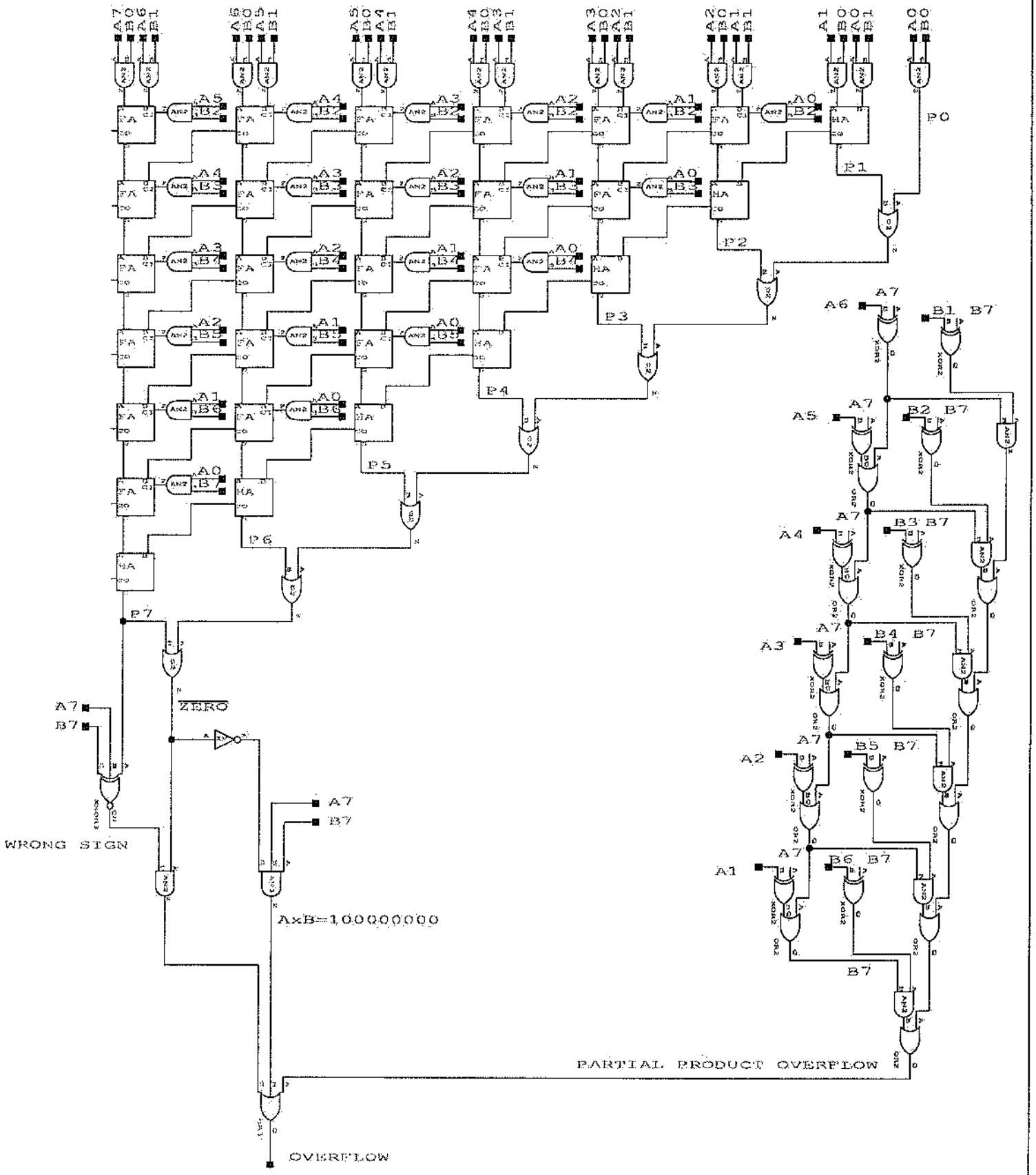
Unsigned Carry C64/C32

Figure 3



Signed Overflow: V64/V32

Figure 4



8-BIT CSA-ARRAY

SINGLE-LENGTH MULTIPLIER

FIGURE 5

BIBLIOGRAPHY

[Agrawal et al. 1983] Agrawal, Dharma P., Girish C. Pathak, Nikunja K. Swain and Bhuwan K. Agrawal. "On Design and Performance of VLSI Based Parallel Multiplier" Proceedings 6th Symposium on Computer Arithmetic., (June 1983) pp 17-22.

[Ciminera and Serra 1983] Ciminera, L. and A. Serra. "Fast Iterative Multiplying Array" Proceedings 6th Symposium on Computer Arithmetic., (June 1983) pp 60-66.

[Conrad et al. 1975] Conrad, Clifford L., Nancy J. Conrad, and Harry B. Higley. "Computer Mathematics" Rochelle Park, NJ: Hayden Book Company, Inc., 1975.

[Flores 1963] Flores, Ivan. "The Logic of Computer Arithmetics" Englewood Cliffs, NJ: Prentice-Hall, Inc., 1963.

[Gillie 1965] Gillie, Angelo C. "Binary Arithmetic and Boolean Algebra" New York, NY: McGraw-Hill Book Company, 1965.

[Gosling 1980] Gosling, John B. "Design of Arithmetic Units for Digital Computers" New York, NY: Springer-Verlag New York Inc., 1980.

[Habibi 1970] Habibi A. and P.A. Wintz, "Fast Multiplier" IEEE Transaction on Computers., Vol C-19 (February, 1970), pp 153-157.

[Stein and Munro 1971] Stein, Marvin L., and William D. Munro. "Introduction to Machine Arithmetic" Reading, MA: Addison-Wesley Publishing Co., 1971.

[Takagi 1991] Takagi, Naofumi. "A Radix-4 Modular Multiplication Hardware Algorithm Efficient for Iterative Modular Multiplications" IEEE Proceedings of the 10th Symposium on Computer Arithmetic (May 1991).

[Wallace 1964] Wallace, C.S., "A Suggestion for a Fast MUltiplier" IEEE Transaction on Electronic Computers, Vol 13 (1964) 14-17.

[Dadda 1985] Dadda, Luigi. "Fast Multipliers for Two's-Complement Numbers in Sereal Form" IEEE Proceedings of the 7th Symposium on Computer Arithmetic (June 1985) pp 57-63.

[Patterson 1990] Patterson, David A. and John L. Hennessy. "Computer Architecture A Quantitative Approach" San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1990.