# THE VLSI IMPLEMENTATION OF A SQUARE ROOT ALGORITHM

**J. Bannur[*] and A. Varma**

**Technical Report DISC/84-3**

**Digital Integrated Systems Center**

**Department of Electrical Engineering- Systems**
**University of Southern California**
**Los Angeles, CA 90089.**

**15 October 1984**

[*] The author is presently with National Semiconductor Corporation, Mail Stop D3-668, Santa Clara, CA-95051. This work was performed at the University of Southern California.

# *ABSTRACT*

VLSI implementation of a square root algorithm is studied. Two possible implementations of the basic non-restoring algorithm are presented — the second is more area-efficient and modular than the first. The implementations are simple and easy to control, but, at the same time, are more area-time efficient than many existing designs. A hardware algorithm suited to microprogram implementation is also given. Extension of the algorithms to achieve $\frac{1}{2}$-bit precision is discussed.

# 1. Introduction

Several algorithms for fast computation of the square root of binary numbers have been proposed in the literature. They can be broadly classified into *restoring* and *non-restoring* algorithms [1]. Some of them use redundant representations, mainly to avoid the rippling of carry associated with addition/subtraction [2]. Other techniques for speed-up include processing multiple bits at a time [3]. In most cases, the speed-up is achieved at the expense of relatively complex control schemes; also, the VLSI implementation is far from regular and lacks a modular structure. In this paper, a VLSI implementation of the basic non-restoring square root algorithm is presented. We describe techniques for optimization of the overall area-time complexity of the implementation by judicious storage and manipulation of operands without sacrificing simplicity. The entire hardware is built from slices, resulting in a modular design; the control circuitry needed is extremely simple. We also present a hardware algorithm for microprogram implementation. Techniques for rounding the result to the nearest integer value are given.

The basic non-restoring algorithm computes the square root in a series of approximation steps. For a $2n$-bit number $x$, the algorithm usually makes an initial guess $y = 2^{n-1}$ and tries to approach the actual value of the square root in a series of $n-1$ refinement steps; during each iteration, the sign of the difference between $x$ and the current value of the *square* of $y$ is used to compute a better approximation for $y$. This algorithm can be expressed as follows:

```
$y := 2^{n-1}$;
for $i := 1$ to $n-1$ do
```

```
        if $(x - y^2) > 0$ then $y := y + 2^{n-i-1}$
        else if $(x - y^2) < 0$ then $y := y - 2^{n-i-1}$
    end.
```

The squaring operation in the algorithm is expensive in terms of area and/or time and can be avoided by a modified scheme [2]. This scheme replaces squaring by a series of shifts and add/subtract operations.

## 2. The Algorithm

Let $x = x_{2n-1} \cdots x_0$ be the $2n$- bit binary number whose square root is to be found. We start with an initial approximation $y = 2^{n-1}$ and refine it in a series of $n - 1$ steps; at the $(n - i - 1)th$ step $(n-2 \geq i \geq 0)$, the new value of $y$ is computed as

$$\hat{y} = y \pm 2^i$$

During the $(n - i)th$ iteration, the value of $x - \hat{y}^2$ can be computed as:

$$\begin{aligned} x - \hat{y}^2 &= x - (y \pm 2^i)^2 \\ &= (x - y^2) \mp (2^{i+1}.y \pm 2^{2i}) \\ &= (x - y^2) \mp \Delta_i \end{aligned} \tag{1}$$

Thus, if we store the value of $x - y^2$ during each iteration, the new value of $x - y^2$ can easily be computed from its old value in a straightforward way. The computation of $\Delta_i = (2^{i+1}.y \pm 2^{2i})$ involves a shifting of $y$ by $i+1$ places. Addition or subtraction of the constant $2^{2i}$ can be achieved by setting or resetting a pair of bits of the shifted operand.

The following are the basic operations involved in the algorithm:

(i) Computation of $\Delta_i = 2^{i+1}.y \pm 2^{2i}$.

(ii) Addition/Subtraction of $\Delta_i$ to $(x - y^2)$ to generate the new value of $(x - y^2)$.

(iii) Refinement of $y$ by addition/subtraction of $2^i$.

In this paper, we describe an implementation that achieves operations (i) and (iii) in $O(n)$ area and constant time. We do not consider optimization of the addition/subtraction operation.

## 3. VLSI Implementation

In this section, we will describe two possible implementations of the algorithm. The first scheme uses a barrel shifter for the generation of $\Delta_i$. The second design eliminates the barrel shifter by clever storage and manipulation of operands, and is therefore more area-efficient than the first.

### 3.1 Implementation Using Barrel Shifter

A straightforward way to generate the correction $\Delta_i$ is by means of a barrel shifter and additional logic to manipulate individual bits at the output of the shifter. As a matter of convenience of implementation, the logic can be moved to the input of the barrel shifter without affecting the result of the computation; this reduces the number of gates by half.

During the $(n - i)th$ iteration, the value of $\Delta_i$ is given by

$$\begin{aligned} \Delta_i &= 2^{i+1}.y \pm 2^{2i} \\ &= 2^i.(2y \pm 2^i) \end{aligned} \qquad (2)$$

Since $y$ is of the form $y_{n-1}\, y_{n-2} \cdots y_{i+2}\, 1\, 0 \cdots 0$,

$$2y + 2^i = y_{n-1}\, y_{n-2} \cdots y_{i+2}\, 1\, 0\, 1\, 0 \cdots 0,$$

$$\text{and, } 2y - 2^i = y_{n-1}\, y_{n-2} \cdots y_{i+2}\, 0\, 1\, 1\, 0 \cdots 0$$

Thus we need to modify only a maximum of three bits at the input of the shifter during every iteration. The value of the bits depend on the sign of $x - y^2$ in the last iteration. Multiplication of $y$ by 2 is achieved by a skewed connection at the input of the shifter.

Figure 1 shows the block diagram of this implementation. It consists of the following registers:

(i) An $n$- bit register Y to store the current approximation to the square root $y$.

(ii) An $n$- bit shift register **b** to store the constant $2^i$ to be added/subtracted during each iteration.

(iii) A $2n$- bit register X which stores the current value of $(x - y^2)$.

During every iteration cycle, the current value of $y$ is used to compute $\Delta_i$ at the output of the barrel shifter. This value of $\Delta_i$ is then added to or subtracted from the contents of X to obtain the new value of $(x - y^{2)}$ for the next iteration. The carry flip-flop stores the sign of $(x - y^2)$ which is used to determine whether an add or subtract operation is to be carried out in the next cycle.

The logic to set/reset individual bits at the input of the shifter is simple and can be incorporated at the output of the Y register. Let $a_i = a_{i,n}\, a_{i,n-1} \cdots a_{i,0}$ denote the value of $2y \pm 2^i$ during the $(n - i)th$ iteration. During this cycle, the $i$th bit of **b** is 1 and the rest of the bits are zero. The individual bits of $a_i$ can be computed using the Boolean expressions:

$$a_{i,j+1} = (y_j + b_{j+1} + b_j.\overline{ADD})\,(\overline{b}_{j-1} + ADD)\,;\ 0 \leq j \leq n-1 \text{ with } b_n = 0.$$

$$a_{i,0} = b_0.ADD \tag{3}$$

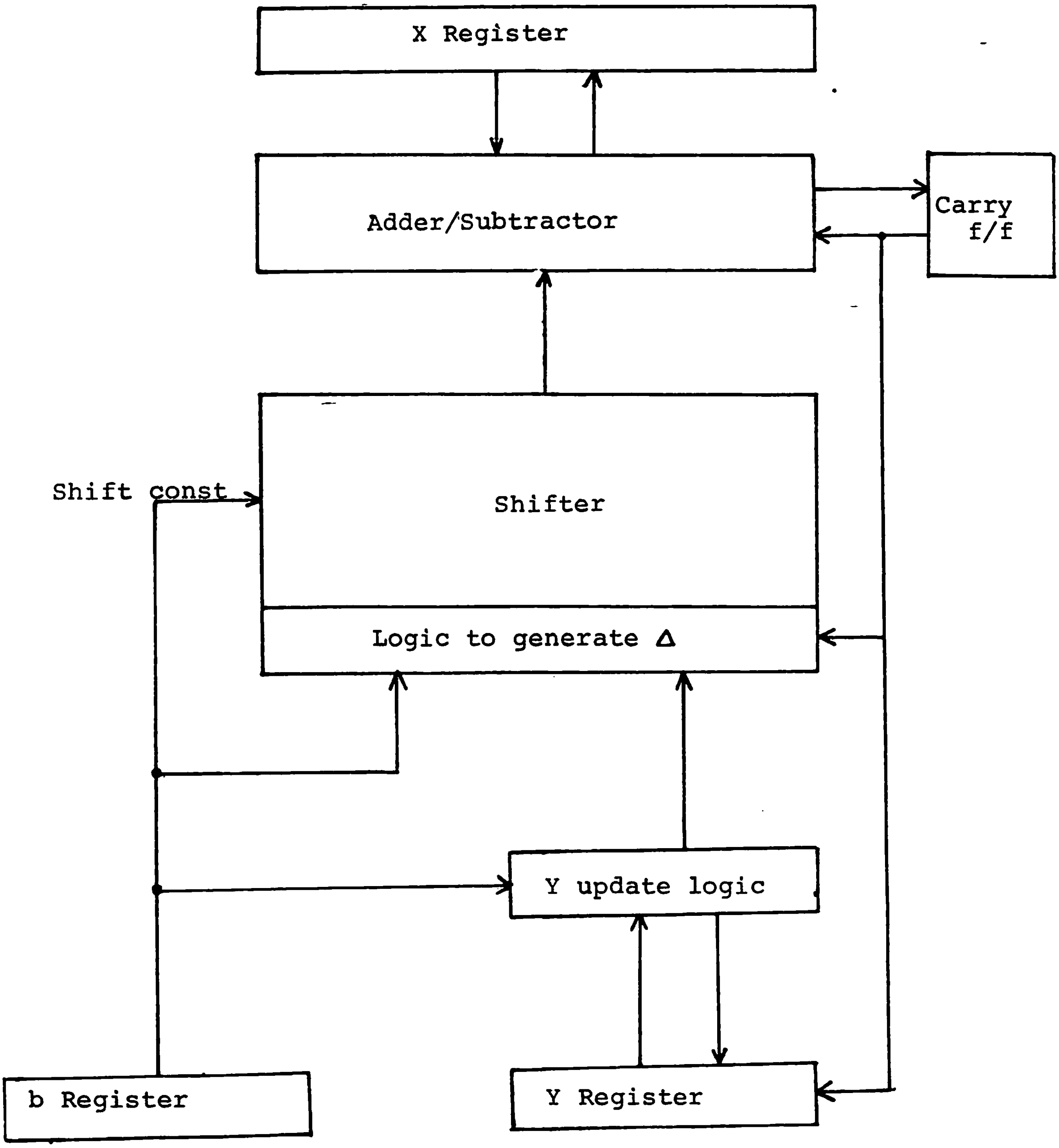


Figure 1
Implementation of the Square Root Algorithm
Using Barrel Shifter

*ADD* is the output of the carry flip-flop. Since each bit of $a_i$ is dependent only on three adjacent bits of $b$, registers **b**, Y, and the logic to generate $a_i$ can be combined and implemented in single-bit slices.

The value of $y$ is updated at the end of every iteration by adding or subtracting $b = 2^i$. Since $y$ is of the form $y_{n-1}\, y_{n-2} \cdots y_{i+2}\, 1\, 0 \cdots 0$ at the beginning of the $(n - i)th$ iteration, the new value of $y$ is given by:

$$\hat{y} = y + b = y_{n-1}\, y_{n-2} \cdots y_{i+2}\, 1\, 1\, 0 \cdots 0 \quad \text{if } ADD = 1$$

$$\hat{y} = y - b = y_{n-1}\, y_{n-2} \cdots y_{i+2}\, 0\, 1\, 0 \cdots 0 \quad \text{if } ADD = 0$$

Only a pair of adjacent bits are modified during every iteration; this property enables us to use simple logic to update the Y register in place of a subtractor. The individual bits of $\hat{y}$ can be generated using the Boolean expression

$$\hat{y}_j = (y_j + b_j)\,(\bar{b}_{j-1} + ADD) \tag{4}$$

This logic can easily be incorporated into a bit-slice along with register bits $y_j$ and $b_j$, and the logic to form $a_{i,j}$.

The initial value of $x - y^2$ can be generated easily from $x$ by performing an extra iteration initially. This iteration makes use of the unused bit $b_7$ of the **b** register. If we start with $b_7 = 1$, $y = 0$, and carry f/f $= 0$, the first iteration will compute $\Delta = (2^{n-1})^2$, which is the initial value of $y^2$. The X register is thus correctly updated to the initial value of $x - y^2$ at the end of the first iteration.

The carry flip-flop which generates the *ADD* signal stores the sign bit of $x - y^2$, and can be considered as the extension of the X register; correct operation is achieved by complementing it every time a carry/borrow occurs from the adder/subtractor during the computation of $x - y^2$ and leaving it unchanged otherwise.

Figure 2 shows the floorplan of the chip for a 16-bit implementation of this scheme. Circuits were designed based on the 2- *micron* *NMOS* technology. A bit slice of type 1 covers one bit each of the **b** and Y registers, Y-update logic, and logic to generate $a_i$. Bit slices of type 2 incorporate individual bits of register X, adder/subtractor, and the precharged output buffers of the barrel shifter. The adder/subtractor is a straightforward implementation of the Mead&Conway ALU [4], and was adapted from [5]. The zero detector decodes the all-zero condition of the X register to terminate the algorithm. Control circuitry and the carry flip-flop are not shown. The internal organization of the bit-slices is shown in Figure 3.

Control of the chip is achieved with a four-phase clock. During initialization, the X register is loaded with the 16- bit operand; the Y register is loaded with $2^7$, and the carry flip-flop is reset to zero. The execution of the algorithm takes a maximum of eight iterations. Register **b** serves the additional function of the iteration counter also. Table 1 lists the four phases of activity during each iteration:

| Phase | Operation |
|---|---|
| 1 | Compute $a_i$;<br>Precharge shifter o/p buffer. |
| 2 | Shift operation;<br>stop iteration if X = 0. |
| 3 | First phase of addition/subtraction;<br>update Y register. |
| 4 | Second phase of addition/subtraction;<br>update **b**, X registers and carry f/f. |

Table 1

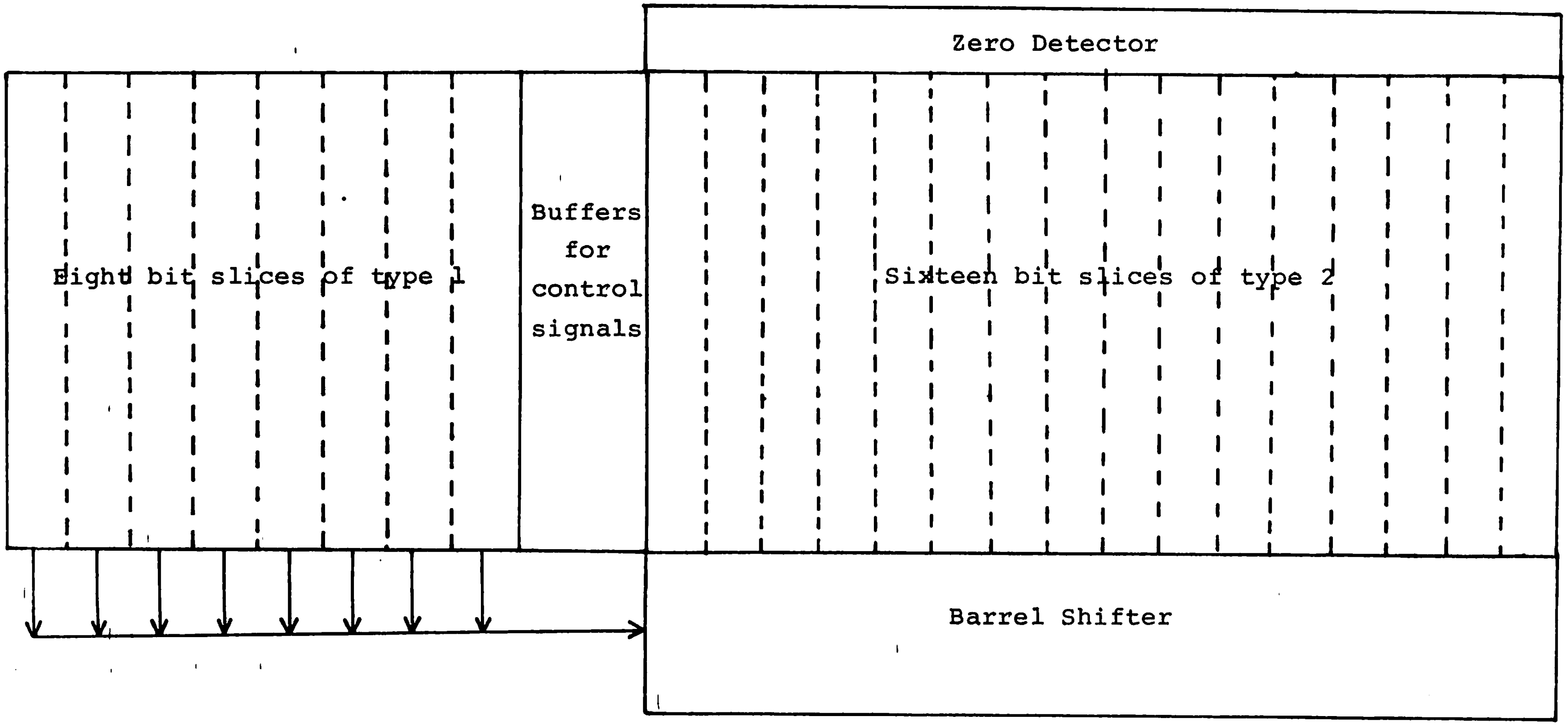
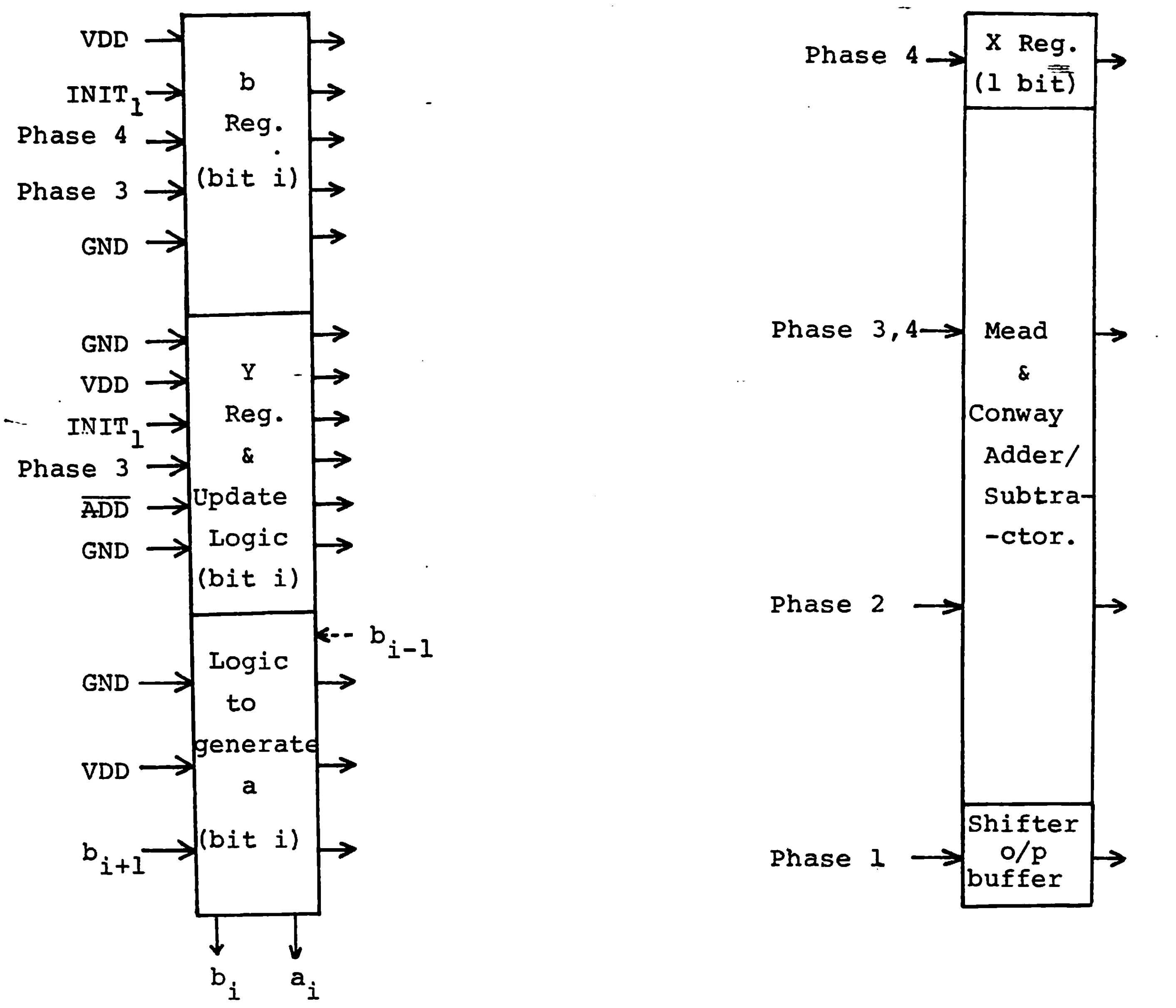

Zero Detector
Eight bit slices of type 1
Buffers for control signals
Sixteen bit slices of type 2
Barrel Shifter

Figure 2.
Floorplan for a 16-bit implementation



(a)

Bit Slice of type 1

(b)

Bit Slice of type 2

(GND and VDD not shown)

Figure 3
Internal Organization of the Bit Slices

The critical path in the implementation is the path from the output of the shifter through the ripple-carry adder/subtractor to the carry flip-flop and the X register. A SPICE simulation revealed a delay of approximately 100 $nS$ for every 4-bit group of the adder/subtractor. The output of the shifter is available at the beginning of phase 3 and the carry flip-flop is updated at the end of phase 4. This enables the 8-bit module to be clocked with a 300 $nS$ clock.

### 3.2 Alternate Implementation Without Barrel Shifter

The previous scheme used a barrel shifter to generate the value of $2^{i+1}.y$ during every iteration. A straightforward implementation of the barrel shifter takes $O(n^2)$ area; besides affecting the modularity of the design, this makes the implementation unwieldy for large $n$. In this section, we describe an alternate design that eliminates the barrel shifter — the function of the barrel shifter is achieved by extending registers Y and **b** and shifting them during every cycle.

In this design, we use $2n$-bit registers for storage of both $b$ and $y$. (In fact, only the even bits of register **b** are used, so the odd bits need not be present). Figure 4 shows the block diagram of this implementation. Registers X, Y, and **b** are all $2n$-bits wide. Initially $y$ is stored in register Y in its most significant half. It is shifted right once during every iteration so that its contents represent $2^{i+1}.y$ at the beginning of the $(n-i)$th iteration $(n-1 \geq i \geq 0)$. The **b** register is initially loaded with $(2^{n-1})^2$ and is shifted right twice during every iteration so that it contains $2^{2i}$ at the beginning of the $(n-i)$th iteration. Thus, according to Equation (2), the value of $\Delta_i$ can be computed by adding or subtracting the contents of register **b** and register Y. The addition/subtraction can be achieved, as in the previous implementation, by means of simple logic to set/reset individual bits, thereby eliminating the need for carry propagation. The individual bits of $\Delta_i = \Delta_{i,n-1}, \cdots, \Delta_{i,0}$ can be computed by the following

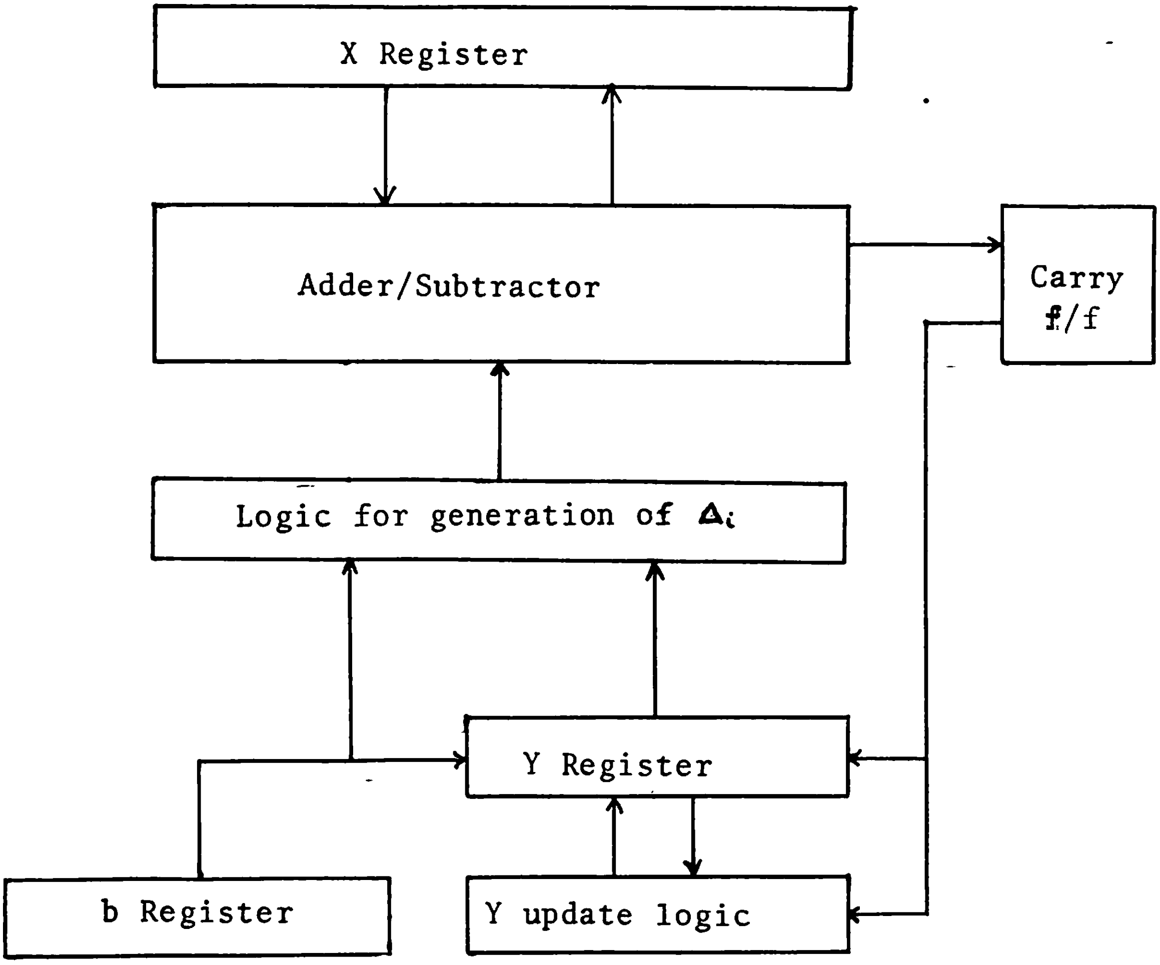


Figure 4
Implementation of the Square Root Algorithm
Without Barrel Shifter

logic:

*for even bits* :

$$\Delta_{i,2j} = (Y_{2j} + b_{2j})(\bar{b}_{2j-2} + ADD) \quad 0 < j \leq n-1$$
$$\Delta_{i,0} = \mathrm{Y}_{2j} + b_{2j} \tag{5}$$

*for odd bits* :

$$\Delta_{i,2j+1} = \mathrm{Y}_{2j+1} + b_{2j}.ADD \quad 0 \leq j < n-1$$
$$\Delta_{i,2n-1} = 0 \ \ always. \tag{6}$$

*ADD* is the output of the carry f/f. Once $\Delta_i$ is obtained, it is added to(subtracted from) the X register to get $(x - \hat{y}^2)$ as discussed in Section 3.1. The addition(subtraction) also toggles the carry f/f if there is a carry(borrow).

Refinement of $y$ can be done in a similar fashion as in the previous section. However, since the Y register now contains a shifted version of $y$, the updating logic is slightly different. The new values for the individual bits of Y are given by:

*for even bits* :

$$\mathrm{Y}_{2j} = \mathrm{Y}_{2j}.(\bar{b}_{2j-2} + ADD) \quad 0 \leq j \leq n-1 \tag{7}$$

*for odd bits* :

$$\mathrm{Y}_{2j+1} = \mathrm{Y}_{2j+1} + \bar{b}_{2j} \quad 0 \leq j \leq n-1 \tag{8}$$

This logic is used to set, reset, or leave unchanged the individual bits of Y during the Y update phase.

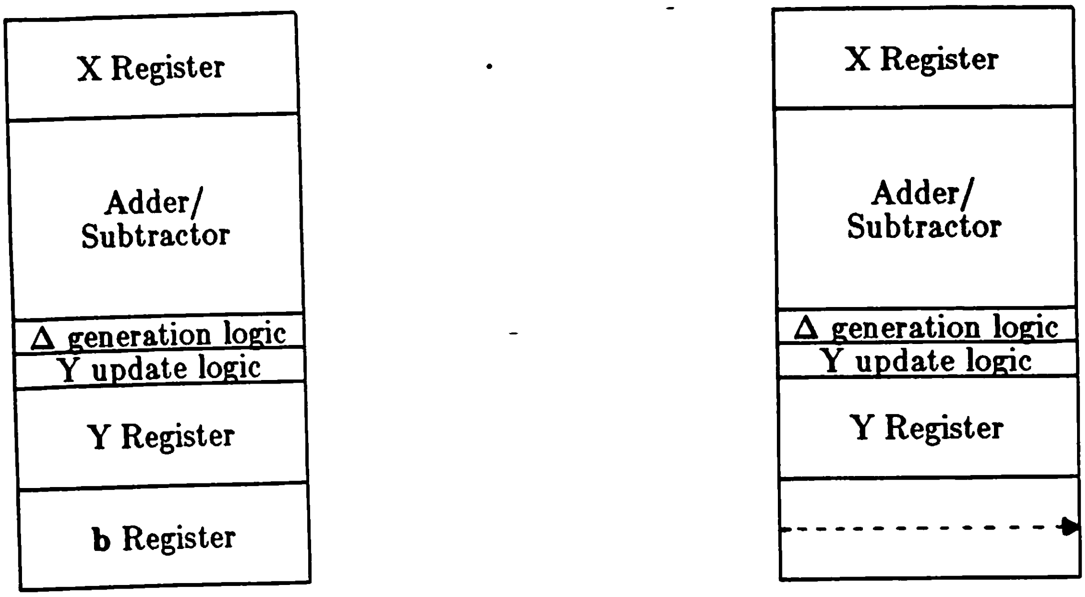
Initially, the register X is loaded with the operand $x$, **b** is loaded with $2^{2n-1}$, and Y with $2^{2n-2}$. The first iteration is equivalent to the initialization step wherein the X register gets correctly updated to $x - 2^{2n-2}$. At the beginning of the second iteration, we need the Y register to contain $2^{2n-2}$ — this requires the shifting of Y to be inhibited in the first cycle. Every iteration requires the **b** register to be shifted right by two bits. Since only the even bits of **b** are used, the same result can be achieved by building only the even bits and shifting it right once every iteration.

The number of clock phases needed can be reduced to three in this implementation due to the absence of the shifter. The following table lists the operations done during each phase of the clock.

| Phase | Operation |
|---|---|
| 1 | First phase of addition/subtraction.<br>(stop iteration if **b** = 0) |
| 2 | Second phase of addition/subtraction;<br>update Y if X $\neq$ 0. |
| 3 | Shift right Y and **b**;<br>update X if not already zero;<br>update *ADD* f/f. |

Table 2

Figure 5 shows the internal organization of the bit slices. Bit slices of type (a) occupy even positions and slices of type (b) occupy odd positions. The even bit slices contain one bit each of the X, Y, and **b** registers, adder/subtractor, logic for generation of the $\Delta_i$ bit, and logic for updating the Y bit. The odd slices are similar except that, instead of the cell for **b** register, there is a straight connection that provides the shift path for the **b** register; the logic for generation of $\Delta_i$ and update of Y are also different. The bit slices thus constitute the whole of the hardware except the control unit, result-



(a) even bit slice

(b) odd bit slice

Figure 5
Internal Organization of the Bit Slices

ing in a truly modular organization. The limit to the number of slices is evidently caused by the delay due to the rippling of carry associated with the addition/subtraction. Carry lookahead schemes may be necessary for large values of $n$.

### 3.3 Processing of Floating Point Numbers

The design in Section 3.2 can be expanded to handle floating point operands with a small overhead on hardware and time. The operation for computation of square root of a floating point number under this scheme can be summarized as follows:

(i) If the exponent is odd, shift mantissa right and increase exponent by one so that it becomes even.

(ii) Compute square root of mantissa as outlined in Section 3.2.

(iii) Shift exponent right to achieve division by two.

(iv) Normalize result, if necessary.

Observe that, if the operand is initially normalized, it takes at most one extra cycle for normalization of the result. Thus the computation of square root of a floating point number requires little extra overhead in terms of time as compared to computing the square root of its mantissa.

It is interesting to compare the efficiency of our design with some of the existing implementations. As an example, the 8087 numeric processor from Intel [6, page S-3] takes 180 cycles to compute the square root of an 80-bit real number comprised of a 64-bit mantissa and 16-bit exponent. Our implementation achieves the same function in $3 \times 32 = 96$ cycles plus a few extra cycles for shifting and adjustment of exponent.

## 4. Hardware Algorithm for Microprogram Implementation

In this section, we present a simple and efficient hardware algorithm for computation of square root of a $2n$-bit integer. This algorithm requires a minimum of hardware and can be microprogrammed into many of the existing mini- and microcomputer systems. The algorithm makes use of the following hardware:

(i) A $2n$-bit register X.

(ii) Two $2n$-bit registers, Y and **b**, with shift-right capability.

(iii) A flag $E$ that stores the bit being shifted out of register **b**.

(iv) A $2n$-bit ALU capable of addition/subtraction with flags $CY$ and $ZERO$ to record the carry and zero condition respectively.

The algorithm is illustrated in Figure 6. It is based on the same technique as the implementation in Section 3.2. Register X stores the value of $x-y^2$, register Y stores a shifted version of $y$, and register **b** stores $2^{2i+1}$ at the beginning of the $(n-i-1)th$ iteration $(n-2 \geq i \geq 0)$. The increment/decrement loop will be executed a maximum of $(n-1)$ times. This algorithm can easily be extended to handle floating point numbers as outlined in Section 3.3.

```
        (* Hardware Algorithm for Square Root *)

        X ← operand;  Y ← $2^{2n-2}$;  b ← $2^{2n-3}$;  $E$ ← 0;

        X ← X - $Y$;

        jump on zero ($ZERO = 1$) to $END$;
        jump on borrow ($CY = 1$) to $DECR$;

$INCR$ :  Y ← Y + b;

        Shift b and Y right; (* in parallel *)

        X ← X - b;

        Shift b right through $E$;
        jump if $E = 1$ to $END$;

        X ← X - $Y$;

        jump on zero ($ZERO = 1$) to $END$;
        jump on no borrow ($CY = 0$) to $INCR$;
        (* else fall through to $DECR$ *);

$DECR$ :  Y ← Y - b;

        Shift b and Y right; (* in parallel *)

        X ← X - b;

        Shift b right through $E$;
        jump if $E = 1$ to $END$;

        X ← X + $Y$;

        jump on zero ($ZERO = 1$) to $END$;
        jump on no carry ($CY = 0$) to $DECR$;
        jump to $INCR$

$END$ :  result ← Y

        stop.
```

Figure 6

Hardware Algorithm for Microprogram Implementation

## 5. Rounding Schemes

The algorithms presented in Sections 3 and 4 produce results with a maximum error of ±1 bit. The precision can be improved to $\frac{1}{2}$ bit by rounding off the result. In this section, we examine how rounding could be incorporated into our implementations.

Let us assume that the operand $x$ is not a perfect square. Application of the algorithms in Section 3 or 4 produce either $y = \lfloor\sqrt{x}\rfloor$ or $y = \lceil\sqrt{x}\rceil$ as the result, depending on whether the approximation was approached from below or above. Let $u = \lfloor\sqrt{x}\rfloor$. Then:

$$|x - y^2| < (u+1)^2 - u^2 = 2u + 1$$

To obtain $\frac{1}{2}$-bit precision, we need to set $y = u$ if $u^2 \leq x \leq u(u+1)$ and $y = u+1$ if $u(u+1) + 1 \leq x < (u+1)^2$. In the implementation, this correction can be applied from knowledge of whether the last operation was an addition or subtraction. If the last operation was an addition, the Y register will contain $y = \lceil\sqrt{x}\rceil = u+1$ at the end of the last iteration; if the last operation was a subtraction, the Y register will contain $y = \lfloor\sqrt{x}\rfloor = u$. This information can be obtained from the status of the carry f/f at the end of the last iteration. If the carry f/f is zero, the final value of $x - y^2$ is positive, implying $y = u$, and if it is 1, the final value of $x - y^2$ is negative, so that the Y register contains $y = u+1$. The following paragraphs outline how the result can be corrected in the two separate cases:

*case (i)* $y = u$ : In this case, subtract the contents of Y from X to obtain $w = (x - y^2) - y = x - u(u+1)$. If $w \leq 0$, the Y register requires no correction; else it needs to be incremented.

*case (ii)* $y = u+1$ : In this case the X register contains $2^{2n} - \left((u+1)^2 - x\right)$. Add the contents of the Y register to X to obtain $w = 2^{2n} - \left((u+1)^2 - x\right) + (u+1) = 2^{2n} - \left(u(u+1) - x\right)$, toggling the carry f/f if there is a carry from the X register. Now, if $w \geq 0$ (carry f/f = 0), we need to decrement the Y register; if $w < 0$ (no carry resulted from X), no correction is required.

The algorithms in Sections 3 and 4 can be extended easily to include this rounding procedure.

## 5. Conclusion

Area-time efficient VLSI implementations of the non-restoring square root algorithm were described. The first algorithm uses a barrel shifter. The second algorithm eliminates the barrel shifter, and is therefore more efficient than the first in terms of both area and time complexity. The design is modular and requires simple control. A hardware algorithm suitable for microprogram implementation was also presented. Extension of the algorithms to handle floating-point numbers and techniques for round-off were also discussed.

## Acknowledgements

The authors wish to thank Dr. Alice Parker for her encouragement and guidance, and David Knapp for the initial idea.

## References

[1] Flores, I., "The Logic of Computer Arithmetic," Prentice-Hall, 1963.

[2] Metze, G., "Minimal Square Rooting," *IEEE Transactions on Electronic Computers,* Vol. EC-14, April 1965, pp 181-185,

[3] Majerski, S., "Square Root Algorithms for High-Speed Digital Circuits," *Proceedings of the Sixth Symposium on Computer Arithmetic,"* April 1983, pp 99-102.

[4] Mead, C. and Conway, L., "Introduction to VLSI Systems," Addison-Wesley, 1980.

[5] Newkirk, J. and Mathews, R., "The VLSI Designers' Library," Addison-Wesley, 1983.

[6] iAPX 86,88 User's Manual, Intel Corporation, Santa Clara, July 1981.