HIGHLY SCALABLE MULTIPLIER

by

Abhijit M. Ajmera

A Thesis Submitted to the Faculty of

College of Engineering

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

Florida Atlantic University

Boca Raton, Florida

December 2003

HIGHLY SCALABLE MULTIPLIER

by
Abhijit M. Ajmera

This thesis was prepared under the direction of the candidate's thesis advisor, Dr. Ravi Shankar, Department of Computer Science & Engineering, and has been approved by the members of his supervisory committee. It was submitted to the faculty of College of Engineering and was accepted in partial fulfillment of the requirements for the degree of Master of Science.

SUPERVISORY COMMITTEE:

Thesis Advisor, Dr. R. Shankar

Dr. I. Mahgoub

Dr. M. VanHilst

Chairman, Department of Computer Engineering

Dean, College of Engineering

Division of Research and Graduate Studies

12/12/03
Date

ACKNOWLEDGEMENTS

First of all I would like to express my heartiest gratitude to Dr. Ravi Shankar whose invaluable guidance helped me write this thesis. He has been a great mentor, teacher and a friend to me. Secondly, I would like to thank Dr. Imad Mahgoub and Dr. Michael VanHilst for their support and consideration during the critical time of writing my thesis.

I am also thankful to my colleagues and friends Surya, Soham, Jigisha, Haritha, Raghu and Glenn for their kind support. I would also like to thank Division of Sponsored Research, Florida Atlantic University for funding the project.

Most of all, I would like to thank my parents whose love and encouragement has helped me reach my goals and achieve success. I would like to thank them for giving everything to me and for helping me become who I am. I am also thankful to my brother and his wife for their support and encouragement.

Finally, I would like to express gratefulness to my motherland India and to the United States of America.

ABSTRACT

Author:	Abhijit Ajmera
Title:	Highly Scalable Multiplier
Institution:	Florida Atlantic University
Advisor:	Dr. Ravi Shankar
Degree:	Master of Science
Year:	2003

High speed low power scalable multiplier is needed in computation intensive DSP applications such as video and image compression algorithms. We used an algorithm that folds the larger bit-width operands so smaller bit-width multipliers can be used. Successive folding on operands can be done until a desired threshold is reached. This method has inherent advantages of reuse of optimized building blocks, dynamic reconfiguration, and low power.

We implemented a hardware multiplier based on this algorithm using Verilog hardware description language. Our results show that this multiplier exhibited significant power advantage over Array and Wallace Tree multipliers for comparable speeds, but had higher gate counts.

TABLE OF CONTENTS

Tables

Figures

Maa and Bapu

CHAPTER 1

INTRODUCTION TO HARDWARE MULTIPLIERS

A hardware multiplier is a basic building block used for numeric processing in digital signal processors, microcontrollers, and most general purpose processors, for multi-media and other applications. Multipliers carry out a product of two numbers – a multiplier and a multiplicand. A multiplication is computation intensive and utilizes large amount of power. High speed low power circuits are needed for mobile and handheld devices. For computation intensive applications such as DSP and multimedia running on these portable devices, where the speed of multiplication can be in the critical delay path for speed, we need low power high speed multipliers [Veen00].

This chapter explains the need for the hardware multipliers and its applications. It also discusses performance requirements, tradeoffs, and power consideration of hardware multipliers. We provide an introduction to scalable multipliers, the main focus of this thesis, at the end of this chapter.

1.1 Need for Hardware Multipliers

Many applications today use digital signal processing. Some examples are video and image compression and analysis algorithms, speech recognition, echo cancellation, convolutions, filtering, etc. These algorithms are very computation intensive and have the multiplication operation as one of their basic operations used repeatedly. The

multiplication can be performed either by the software route as in the case of multiplication using Intel 8085 processor or by using a hardware multiplier. Hardware multipliers operate much faster than the software counterpart. In the algorithms mentioned above, the multiplication lies in the critical delay path and determines the speed of the algorithm [ElAb97]. For these applications to meet their timing requirement faster hardware multiplier is needed to carry out the multiplication operation.

1.2 Examples of Devices that use Hardware Multipliers

Hardware multipliers are used by a variety of devices such as DSP cores, multimedia co-processors, microcontrollers, and many general purpose application processors. The following are examples of the devices that use hardware multipliers:

1. DSP Cores
    a. Motorola DSP5685x family (16 x 16 → 32 multiplier)
    b. Motorola DSP56300 family (24 x 24 → 48 multiplier)
    c. Starcore SC110 (16 x 16 → 32 multiplier)
2. Microcontrollers
    a. NEC 32-bit V850E (32 x 32 → 64 multiplier)
    b. TI 16-bit MSP430 (16 x 16 → 32 multiplier)
3. Microprocessors
    a. ARM ARM926EJ-S (16 x 32 multiplier)

1.3 Design Parameters and Tradeoffs

There are three design parameters for any hardware design – area, speed, and power. Based on the type of application the tradeoffs are done between these three parameters. For example, say the consideration for multiplier is the area and not the speed. In that case, multiplication can be performed using a Sequential Add-Shift Multiplication algorithm, whose implementation will be of small size having a simple architecture, with low-end of performance. Now, if the consideration is speed, then more complex algorithms such as a Wallace multiplier can be used whose implementation will be of large size and have a complex architecture. But the speed of multiplication would be high. Similarly tradeoffs can be done between power and area [Cava84].

Therefore, in order to choose an optimum design for an application, tradeoffs have to be done between the three design parameters – area, power, and speed based on the requirements of the application.

For multipliers, the bit-width of the inputs also affects the choice of the algorithm. A particular algorithm may be more suitable for a particular range of bit-widths of the inputs as compared to others but may not be suitable for bit-width not within the optimum range.

1.4 Need for a Low Power Multiplier

According to the International Technology Roadmap for Semiconductors (ITRS), design for low power has been identified as one of the greatest design challenges in IC design over the next 15 years [ITRS01]. Design for low power was not

an issue until the late eighties. It has become increasingly a hot topic in IC design because of the following reasons [Veen00]:

1 **Packaging:** The number of components on advanced ICs is quadrupling every three years following Moore's Law. The result of this increase in number of components is the increase in the power consumed by the IC. More expensive packaging material has to be used so that they can tolerate high power dissipation.

2 **Demand for Mobile Computation Power:** The need for more computation power on mobile devices is another major driving force for low-power designs. Examples are laptops and PDAs.

3 **Demand for Portable Devices:** The demand of portable and handheld devices is increasing rapidly. One of the main requirements for portable and handheld applications is longer battery life. By using low-power algorithms and circuits not only the batter life is extended but a lighter battery can be used which would result in a lighter product. Examples here are cell phone, portable CD/DVD player etc.

4 **Demand for Portable Multimedia Applications**: Multimedia applications are now widely incorporated in portable devices such as cell phones, PDAs and portable DVD players. These applications are computation intensive and consume large amount of battery power. To meet the market needs low-power implementation of circuits is essential.

The implementation of all the advanced applications mentioned above has a hardware multiplier as one of the building blocks. For this reason, this research is done

on the design and implementation of a low power multiplier algorithm which trades off area for power.

1.5 Scalable Multiplier

As per our definition, scalability in a multiplier refers to a fixed-area smaller bit-width multiplication module that can handle operands of any size at runtime based on the speed and power needs. It provides the capability to choose the precision of multiplication to be carried out on the same multiplier at runtime.

Our multiplier denoted as HSM (Highly Scalable Multiplier) is a highly scalable architecture. The algorithm, design, and implementation of the HSM are discussed in the subsequent chapters.

CHAPTER 2

ALGORITHMS FOR HARDWARE MULTIPLIER

In Chapter 1, the need for the algorithm for a high performance low-power multiplier is explained. This chapter briefly explains some of the well-known algorithms for multiplication such as Sequential Add-Shift, Table-Lookup, Booth Algorithm, Wallace Tree and Modular Array.

2.1 Sequential Add-Shift Multiplication

This method uses a sequential digital circuit to carry out the multiply operation. It carries out multiplication by using successive add and shift operations. It is just like multiplication of two binary numbers done with paper-and-pencil. If the multiplication of n-bit operands is carried out, for every clock cycle the LSB of the multiplier is checked, if it is '1' then the value of the multiplier is added to the upper n-bits of a register and then the values of the register and the multiplier are shifted to the right. This process is repeated for n-clock cycles. At the end of it, the register will give the product [Cava84].

This method requires n clock cycles to compute the product of two numbers n-bit wide. The latency can be very high if n is large which makes this method unsuitable for applications such as DSP, multimedia etc [Cava84].

2.2 Table-Lookup Multiplication

This method of multiplication uses Random Access Memory (RAM) to store different versions of the multiplicand. This method is a variation of standard add-shift method but unlike standard add-shift method it shifts partial product by multiple number of bits. The multiplication using this method is faster than the add-shift multiplication technique. The multiplication is still slow because there is overhead of loading versions of multiplicand from the RAM. Further, the operation of the multiplier is controlled by microcode [Cava84].

2.3 Booth Algorithm

In the add-shift algorithm as discussed in section 2.1, each multiplier bit generates one multiple of the multiplicand to be added to the partial product. If the multiplier is large then the number of addition operations also becomes large which increases the latency. Booth algorithm uses a process called "skipping over 0s" which shifts across the 0s through recoded version of the multiplier and thus reduces the number of addition operations. Therefore, the greater the number of zeros in the multiplier, the faster the multiplication is, using this method.

The advantages of the Booth algorithm is that it treats both, positive and negative operands, as the same, and it increases the speed of multiply operation when the multiplier block has long strings of 1s and 0s. The disadvantage is that the speed of this multiplier is data dependent.

2.4 Wallace Tree Multiplication

Wallace tree multipliers use multiple levels of full adders. Rather than adding the partial products in pairs it adds three bits with the same weight (place value) reducing the result to 2 bits. It uses multiple-levels of such adders until the number of bits vectors of same weight reduces to two. The final product is then calculated by the conventional adder. This reduces the overhead of adding the partial products because of the compression of bit-vectors by 3:2.

A Wallace tree in conjunction with modified Booth Encoding is used by fast multipliers. But Wallace tree results in large power dissipation and area because of the interconnect wires. As a result, it has limited application in battery operated devices [ElAb97].

2.5 Array Multiplier Algorithm

This algorithm involves generation of a matrix of partial products in parallel as against generation of one partial product every cycle. A two-dimensional array of full adders then sums the rows of partial products in parallel to give the product.

The advantage of this algorithm is that it has a regular structure and every cell is connected to the adjoining cells only. The disadvantage of this algorithm is that the delay is proportional to the operand size [RaPe96].

2.6 Montgomery multiplication

It is the basic building block for the modular exponentiation operation required in RSA (Rivest, Shamir, and Adleman) public-key cryptosystems. Most of the

exponentiation chips perform exponentiation as a series of modular multiplications. A scalable architecture for Montgomery multiplication is shown in [TeKo99]. The total time to compute the Montgomery multiplication for a given precision of the operands depends on the available area and the chosen pipeline configuration. The upper limit on the precision of the operands is dictated by the memory available to store the operands and internal results.

2.7 Scalable Hybrid Multiplier Architecture

This multiplier architecture is shown in [KoKi03]. It has the regularity of linear array multipliers and the performance of tree multipliers. It is highly scalable to higher order multiplication. A technique called *Hybrid tree addition using 8-to-2 compressors*, is used which implements a hybrid structure using a combination of linear sub-array addition and tree addition. This hybrid tree addition divides the summation array into 8-bit sub-arrays and then adds them using a tree structure. It increases the scalability of its tree structure significantly.

2.8 MASAR (Multiplication Algorithm for Switching Activity Reduction)

This algorithm as proposed in [ItCh03] reduces switching activity through operand decomposition. This algorithm can be used in conjunction with Array or Tree multipliers to reduce the switching activity. It causes substantial reduction in the number of logic transitions and a corresponding power savings, with only a small delay and area increase.

CHAPTER 3

HIGHLY SCALABLE MULTIPLIER

This algorithm of the scalable multiplier is as described in Shankar [Shan01]. This algorithm uses the following expression for multiplication of two values, X and Y,

$$X \times Y = \left(\frac{X + Y}{2} \right)^2 - \left(\frac{X - Y}{2} \right)^2 \qquad ...(3.1)$$

The proof of the equation (3.1) can be easily obtained by expanding the right-hand side of the equation. This expanded multiplication method is commonly used in implementing analog multipliers because this multiplication method reduces the multiplication process to merely producing the difference of two squared numbers. This type of expanded multiplication process will have intensive memory requirements when the operands i.e. multiplier and multiplicand, are large values. It also results in high power consumption. This makes it impractical to use this method in applications that require repeated multiplications such as in digital signal processing of audio, and video applications [Shan01].

3.1 HSM Algorithm

Our multiplier overcomes these problems with a folding concept. We can evolve a high speed, low power scalable multiplier which has a folding multiplier configured to

fold multiplicands and multipliers whenever they exceed a folding threshold. 'Folding' refers to programmatically/dynamically reducing the size of the multiplicand and multiplier bit widths until the reduced multiplicand and multiplier are below a certain threshold of bit widths.

Equation (3.1) can be rewritten as below to give product of X and Y:

$$X \times Y = (A)^2 - (B)^2 \qquad \text{---(3.2)}$$

where

$$A = \left(\frac{X+Y}{2} \right)$$

$$B = \left(\frac{X-Y}{2} \right)$$

Now 'A' can also be expressed as:

$$A = (A - k + k) \qquad \text{--- (3.3)}$$

Where

k = a folding factor

Therefore,

$$A^2 = (A - k + k)^2$$

$$= [(A - k) + k]^2$$

$$= [(A - k)^2 + 2 \, . \, k \, . \, (A - k) + k^2] \qquad \text{--- (3.4)}$$

Equation (3.4) has a significant advantage as compared to just calculating A^2 in a normal way. Take for example that A is an 8-bit value. If k is a 4-bit value, the size of each term in equation (3.4) becomes a 4-bit operation as compared to the 8-bit squaring operation. It significantly reduces the memory requirement which will become clear when we take an example.

Just as in equation (3.4), 'B' can also be expressed as:

$$B^2 = [\,(B-l)^2 + 2\,.\,l\,.\,(B-l) + l^2\,] \qquad \text{--- (3.5)}$$

Where

l = another folding factor

From equations (3.2), (3.4) and (3.5) we have:

$$X \times Y = [\,(A-k)^2 + 2\,.\,k\,.\,(A-k) + k^2\,] - [\,(B-l)^2 + 2\,.\,l\,.\,(B-l) + l^2\,] \qquad \text{--- (3.6)}$$

It is worth observing that all the terms in equation (3.6) can be folded further in a similar fashion. The terms that are squared can be folded again, as with equations (3.4) and (3.5) and the terms is product of terms can be calculated by using another folding multiplier of half the size of X and Y. We keep on folding the terms until the size of the sub-terms becomes less than or equal to the threshold.

3.2 Example

Consider two 8-bit wide unsigned integers, X and Y, having values 125 and 75 respectively.

$X = 125 = 94_{16} = 10010100_2$

$Y = 75 = 19_{16} = 00011001_2$

From equation (3.2) we have,

$$A = \left(\frac{X+Y}{2}\right) = \left(\frac{125+75}{2}\right) = 100 = 64_{16} = 01100100_2$$

$$B = \left(\frac{X-Y}{2}\right) = \left(\frac{125-75}{2}\right) = 25 = 19_{16} = 00011001_2$$

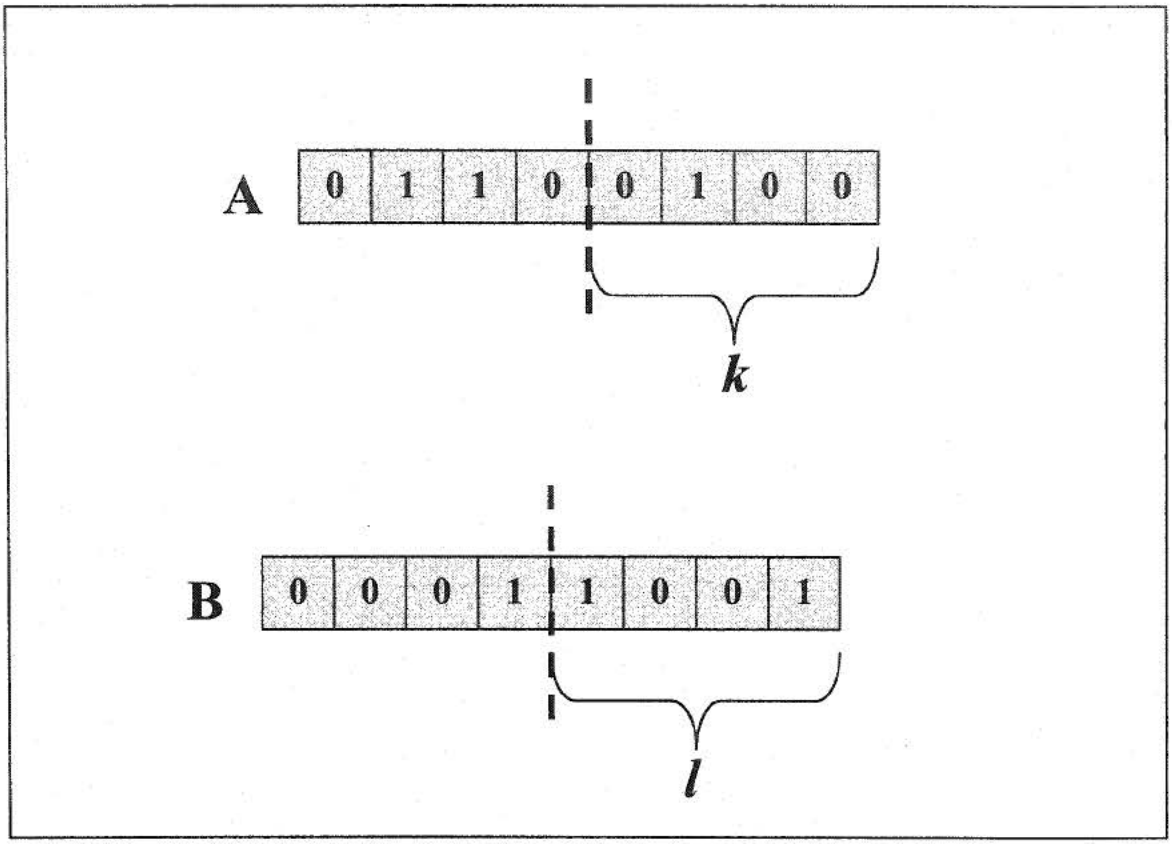

Figure 3.1 Folding of A and B

Now we can fold A and B as shown in Figure 3.1. A can be folded at the mid-point such that $k = 0100$ and $(A - k) = 01100000$. For this example we have chosen the mid-point for folding, but this method can use other point for folding based on the type of value. Now $(A - k)$ can be expressed as:

$$(A - k) = 01100000_2 = \{0110, 0000\}_2 \qquad \text{--- (3.7)}$$

$$\Rightarrow \quad (A - k)^2 = \{(0110)^2, 00000000\}_2 \qquad \text{--- (3.8)}$$

Hence, the operation $(A - k)^2$ is essentially a four bit squaring operation as the size of trailing zeros can be doubled and then concatenated at the end of the result of square of the upper non-zero term. From equation (3.8) we have,

$$(A - k)^2 = \{(0110)^2, 00000000\}_2$$

$$= \{00100100, 00000000\}_2$$

From equation (3.4) we have,

$$A^2 = [\,(A - k)^2 + 2 \,.\, k \,.\, (A - k) + k^2\,]$$

$$= [\,\{(0110)^2, 00000000\}_2 + 2 * (0100)_2 * \{0110, 0000\}_2 + (0100)^2{}_2\,] \quad \text{--- (3.9)}$$

Again, the terms 2.k.(A – k) and k^2 are 4-bit operations. For all the three operations, trailing zeros can be compensated for when adding the terms to get the value of A. From equation (3.9) we have:

$$A^2 = \{000100100, 00000000\}_2 + 2 * \{00011000, 0000\}_2 + 00010000_2$$

Here the multiplication by 2 operations can be translated to a left shift by one position.

$$\therefore \quad A^2 = 0001001000000000_2 + 0001100000000_2 + 00010000_2$$

$$= 0010011100010000_2$$

$$= 10000 \quad \text{--- (3.10)}$$

Similarly,

$$B^2 = 625 \quad \text{--- (3.11)}$$

From equation (3.2), (3.10) and (3.11), we have:

$125 \times 75 = 10000 - 625 = 9325$ (Hence proven)

From now onwards as a convention, terms such as $(A - k)$ and $(B - l)$ would refer to the upper non-zero part. We assume that the trailing zeros would be taken care of during the accumulation of the values of the six terms to give the final product.

Note the following with regard to the folding operation:

1. Multiplication of 8-bit numbers is reduced to six four-bit arithmetic operations.

2. Each time we fold an N-bit multiplication operation, we get four terms which involve N/2-bit squaring operations and two terms which involve N/2-bit multiplications.
3. N/2-bit squaring can be achieved using a table lookup ROM of $N^2/4$ size (a significantly smaller table) because of folding. Note at the 4-bit squaring level, it actually becomes economical to implement with gates.
4. An N/2-bit multiplication can be implemented using another folding multiplier of half the size.

3.3 Advantages of Folding

The advantage of folding can be understood from Figures 3.2 and 3.3. From Figure 3.2 it is clear that if we calculated A^2 and B^2 using table-lookup, the size of the table would be 256 × 16 for each of A and B. Now as shown in Figure 3.3, if we calculate all the six terms using table-lookup, we would need six tables of size 16 × 8.

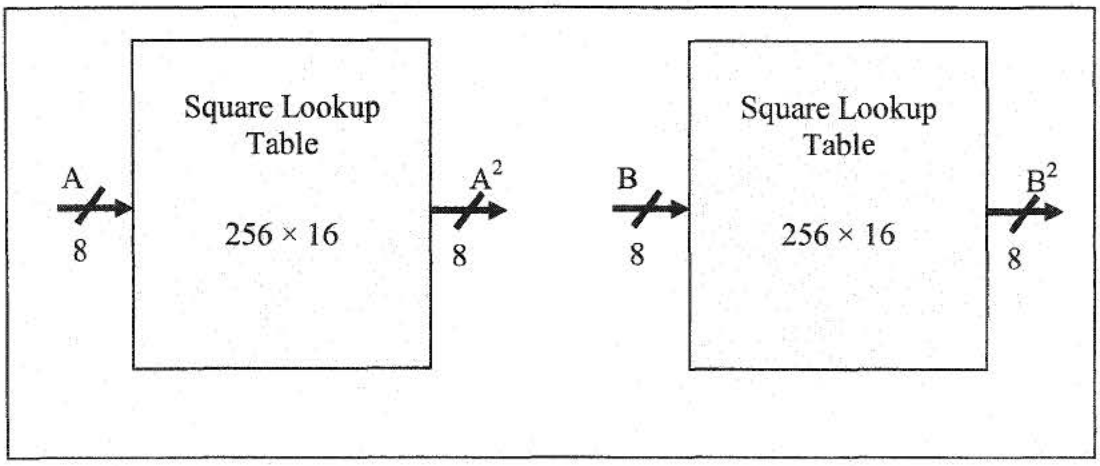

Figure 3.2 Shows implementation of multiplication without folding using square lookup table

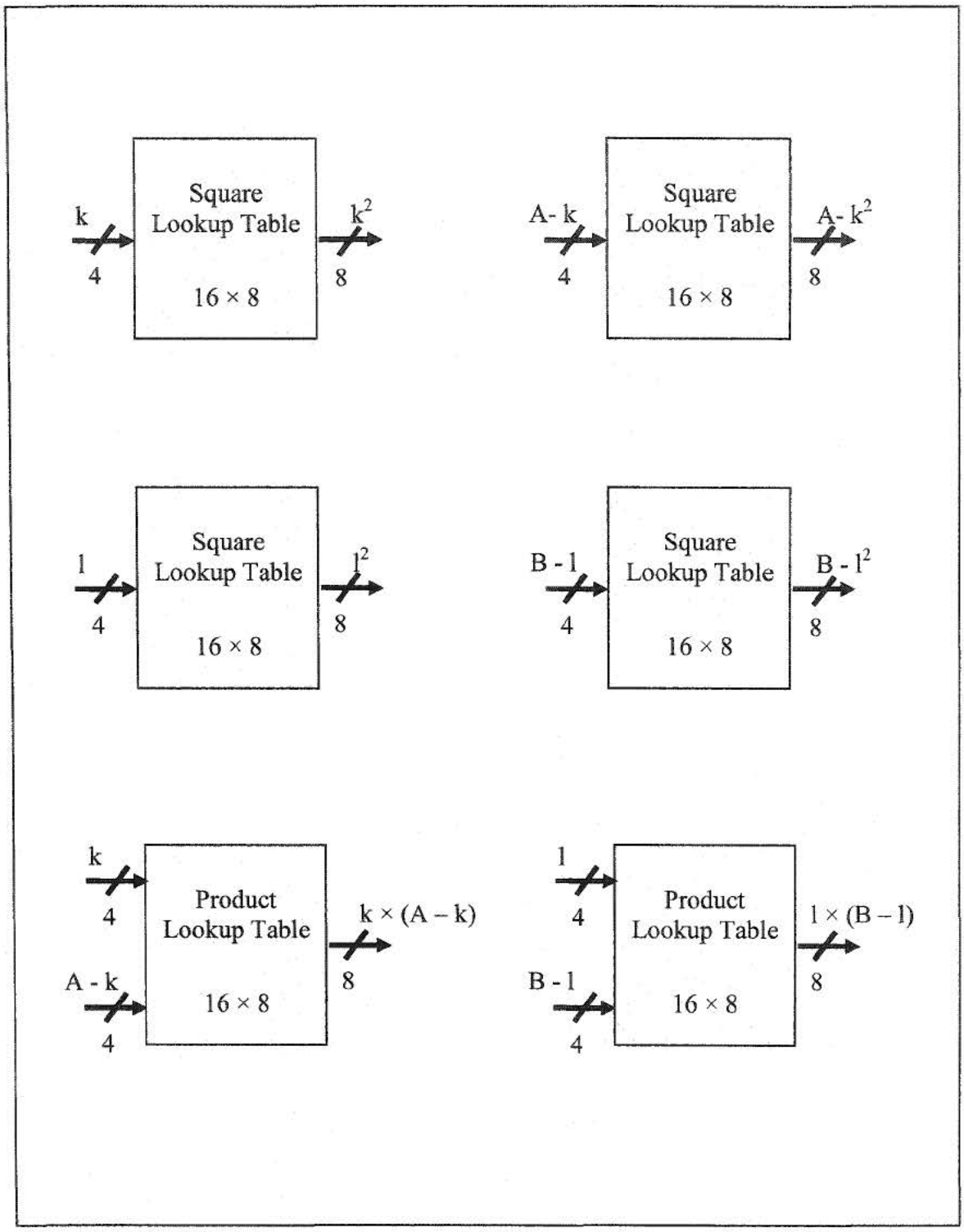

Fig. 3.3 Shows implementation of multiplication using lookup tables with folding

CHAPTER 4

IMPLEMENTATION PLAN OF THE HSM

We explained the HSM Algorithm in Chapter 3. The next step is the implementation of the algorithm in hardware. This chapter explains the implementation plan of the HSM algorithm. Top-down design methodology has been used in the design and implementation of HSM. The main stages in the design are the behavioral simulations of the algorithm as a proof of concept, and the design, implementation, and simulation of the HSM architecture. All the work done for each stage is comprehensively explained in the subsequent chapters.

4.1 Behavioral Simulation of HSM Algorithm

The HSM algorithm was described in Chapter 3. We performed high level simulations for proof of concept and to perform exhaustive testing to find any deviation from the expected behavior. Both, C & the Verilog Hardware Description Language (HDL) were used to develop the behavioral models. We limit ourselves here to the Verilog simulations.

Verilog provides the ability to describe hardware at the following levels of abstraction:

1. Behavioral
2. Structural

3. Gate Level
4. Transistor Level

The behavioral simulation exploited the behavioral level modeling capability of Verilog. Behavioral model just captures the functionality of the block [Paln96].

The following behavioral models were created:

1. HSM: This module implemented the HSM algorithm as specified in Chapter 3
2. Test: This module generates test vectors to test the HSM module and check for any unexpected output
3. Top: This top-level module instantiates HSM and Test modules to perform the simulation

Exhaustive tests using brute force method were done. Deviation from expected behavior was detected and analyzed mathematically. Appropriate changes in the implementation were made to account for the deviation. Behavioral implementation is explained in detail in chapter 5.

4.2 Design and Simulation of HSM Architecture

At the end of behavioral simulation of HSM algorithm, the algorithm for the implementation of HSM is finalized. The algorithm for the implementation is then broken down into steps. Each step is then mapped to digital logic blocks that would carry out that operation. After all the algorithmic steps have been mapped to digital logic blocks, Verilog behavioral models for each of the blocks were developed.

Exhaustive testing was done by using the brute force method, that is, testing the output for every possible combination of inputs. Details of the design and simulation are explained in chapter 6.

4.3 Results: Gate and Toggle Count

Gate count is the measure of the size of the chip. It determines how much silicon area would be needed for hardware implementation of the design. Gate count for each digital logic block was determined. The gate count of all the blocks were then added to give the gate count of the implementation of the entire design.

Toggle count or transition count is an effective surrogate for the power consumed by the circuit [MeRu96]. The dynamic power is the major source of power consumption in static or dynamic CMOS circuits. Dynamic power is directly proportional to the number of transitions in the circuit [ElAb97]. Toggle count of the design was calculated analytically by adding up toggle counts of individual blocks.

The algorithm used to estimate the gate and toggle counts are presented in Chapter 7.

CHAPTER 5

BEHAVIORAL DESCRIPTION AND SIMULATION OF HSM ALGORITHM

In Chapter 3 the algorithm for the HSM was explained. Chapter 4 discusses the implementation plans with Verilog HDL. The algorithm can be described and verified by writing abstract behavioral model using Verilog. Verilog supports multiple abstraction levels from extremely abstract behavioral description, to more detailed structural and gate-level implementation [Paln96]. This chapter explains the behavioral model of the algorithm implemented using Verilog, simulation using Cadence® NC-Verilog and results.

5.1 Modeling Scheme

The modeling scheme consists of three modules:

1. Top: It is the top-level module that instantiates other modules
2. HSM_32: This module implements the HSM algorithm. It has two inputs 'X' and 'Y', the multiplicand and multiplier, respectively. It has one output 'Product' which is the product of X and Y.
3. Test: This module generates the test vectors to test the HSM_32 module. It implements the brute-force method to test the output. It has one input 'Product' and two outputs X and Y.

Figure 5.1 shows the modeling scheme of behavioral description of the HSM algorithm.

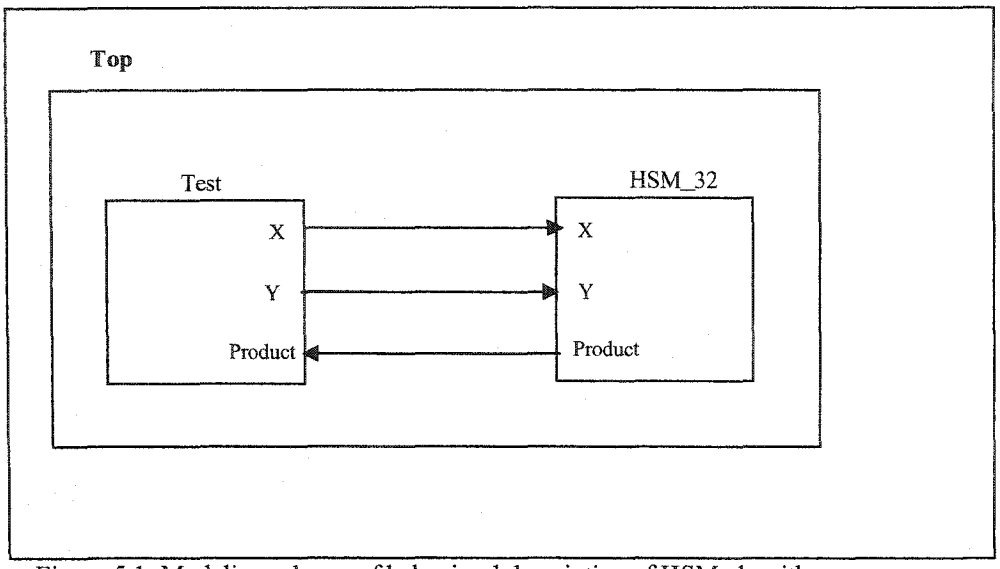

Figure 5.1: Modeling scheme of behavioral description of HSM algorithm

5.2 Behavioral Model of HSM Algorithm

The flowchart for the implemented algorithm is shown in Figure 5.2. The flowchart is very simple. The condition box in the flowchart is used to calculate the absolute value of 'B'. Figures 5.3 (a) and (b) show the behavioral description of HSM algorithm for 32 bit multiplication. The name of the module is HSM_32. Line 1 in Figure 5.3 declares the module name as HSM_32 and the name of the ports as X, Y, and Product. Lines 4-5 declare the port directions of X and Y as input and Product as output. Lines 6-7 declare the port type of all the ports of the module. Port type of X and Y is 32-bit vector and that of Product is a 64-bit register. Lines 11-21 declare internal variables required to implement the algorithm. Lines 25-34 initialize all the variables and ports whose data type is 32-bit register. The code between the lines 37 and 90 implements the

actual HSM algorithm and is executed only when there is any change in the value of X and Y.

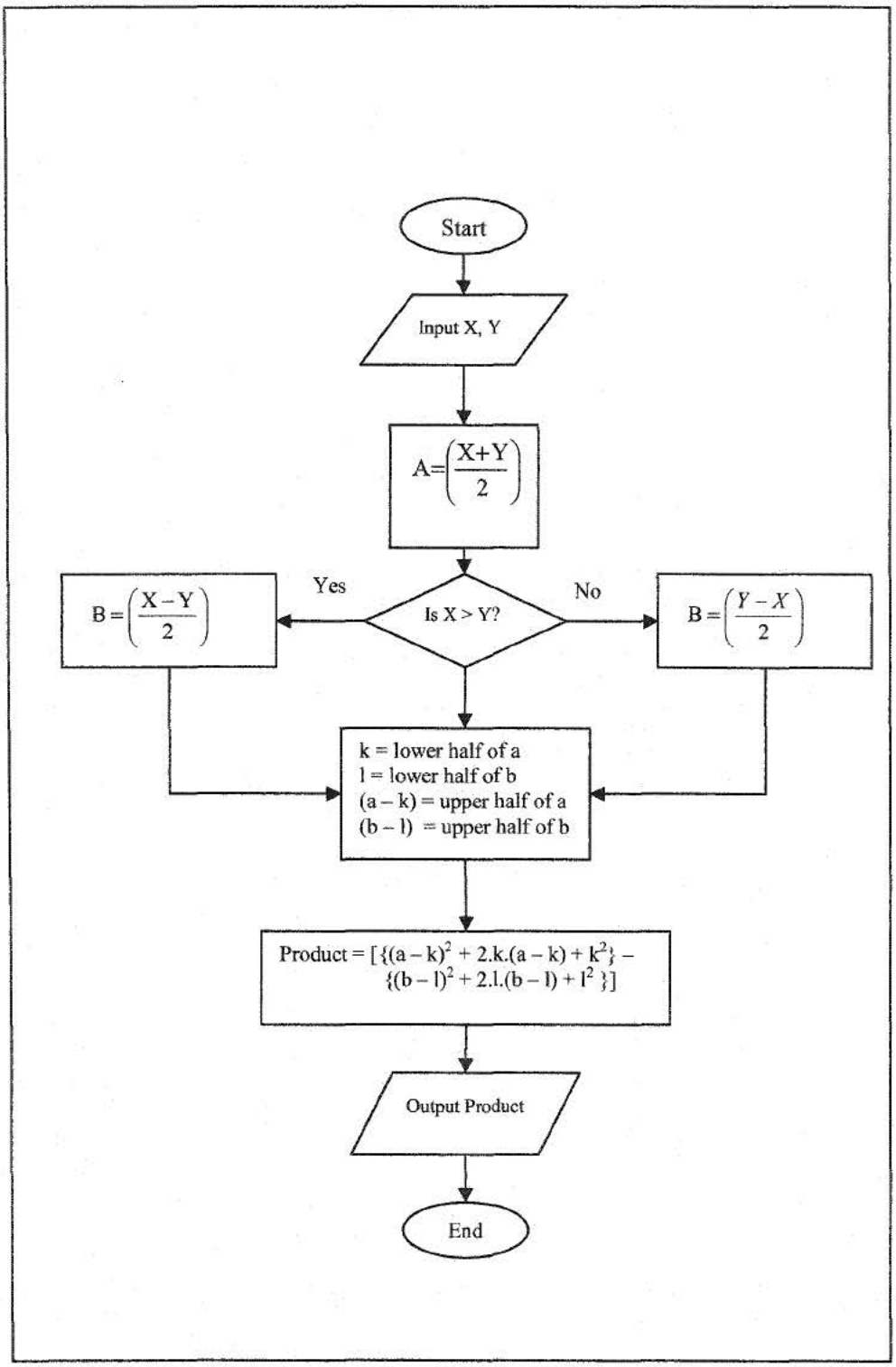

Figure 5.2 The flowchart of implementation of the HSM algorithm

```
module HSM_32 (Product, X, Y);

      // Port Declaration
      output Product;        // Product of X and Y
      input  X, Y;           // Input Values

      reg   [63:0]     Product;
      wire  [31:0]     X, Y;

      // Declaration of internal variables
      reg   [31:0]     a, b;                        // a = (X + Y)/2
                                                    // b = (X - Y)/2
      reg   [15:0]     k, l;                        // k = a Folding Factor
                                                    // l = another Folding Factor
      reg   [15:0]     a_k, b_l;                    // a_k = a - k as in equation 3.6
                                                    // b_l = b - k as in equation 3.6
      reg   [31:0]     k_square, l_square;          // k_square = k * k
                                                    // l_square = l * l
      reg   [31:0]     a_k_square, b_l_square;      // a_k_square = (a - k)(a - k)
                                                    // b_l_square = (b - l)(b - l)
      reg   [31:0]     k2_times_a_k, l2_times_b_l;  // k2_times_a_k = 2.k.(a - k)
                                                    // l2_times_b_l = 2.l.(b - l)

      // Initialization of internal registers
      initial
      begin
            Product = 0;
            a = 0; b = 0;
            k = 0; l = 0;
            a_k = 0; b_l = 0;
            k_square = 0; l_square = 0;
            a_k_square = 0; b_l_square = 0;
            k2_times_a_k = 0; l2_times_b_l = 0;
      end

      // Execute the block if any of the input X or Y changes
      always @(X or Y)
      begin

            a = (X + Y)/2;    //    a = average of sum of X and Y

            // Find the absolute difference between X and Y
            if( X >= Y)
                  b = (X + (~Y) + 1)>>1;
            else
                  b = ((~X) + Y + 1) >> 1;
```

Figure 5.3 (a) Code of HSM_32 module that gives the behavioral description of HSM Algorithm

(Continued to Fig 5.3 (b)).

```
                // Folding a and b at mid-point.
                // k = lower half of a and a - k = upper half of a
                // l = lower half of b and b - l = upper half of b
                k = a[15:0];
                l = b[15:0];
                a_k = a[31:16];
                b_l = b[31:16];


                // Calculate the six terms as in equation 3.6
                k_square = k * k;             // k_square       = square of k
                l_square = l * l;             // l_square       = square of l
                a_k_square = a_k * a_k;       // a_k_square     = square of (a - k)
                b_l_square = b_l * b_l;       // b_l_square     = square of (b - l)

                k2_times_a_k = 2 * k * a_k;   // k2_times_a_k = 2 times product of k and (a - k)
                l2_times_b_l = 2 * l * b_l;   // l2_times_b_l = 2 times product of l and (b - l)


                // Accumulate the six terms in equation 3.6. Calculate the product.
                // The trailing zeros are added at this stage.
                Product = (k_square + {a_k_square, 16'b0} + {k2_times_a_k, 8'b0})
                          - (l_square  + {b_l_square, 16'b0} + {l2_times_b_l, 8'b0});

                /*$display("X \t\t\t = %-b  (%d)\n", X, X);
                $display("Y \t\t\t = %-b  (%d)\n", Y, Y);
                $display("a \t\t\t = %-b  (%d)\n", a, a);
                $display("b \t\t\t = %-b  (%d)\n", b, b);
                $display("k \t\t\t = %-b  (%d)\n", k, k);
                $display("l \t\t\t = %-b  (%d)\n", l, l);
                $display("(a - k) \t\t = %-b  (%d)\n", a_k, a_k);
                $display("(b - l) \t\t = %-b  (%d)\n", b_l, b_l);
                $display("k * k \t\t = %-b  (%d)\n", k_square, k_square);
                $display("l * l \t\t = %-b  (%d)\n", l_square, l_square);
                $display("(a - k)(a - k)\t = %-b  (%d)\n", a_k_square, a_k_square);
                $display("(b - l)(b - l)\t = %-b  (%d)\n", a_k_square, a_k_square);
                $display("2.k.(a - k) \t = %-b  (%d)\n", k2_times_a_k, k2_times_a_k);
                $display("2.l.(b - l) \t = %-b  (%d)\n", l2_times_b_l, l2_times_b_l);
                $display("Product \t\t = %-b  (%d)\n", Product, Product);*/

        end

endmodule
```

Figure 5.3 (b) Code of HSM_32 module that gives the behavioral description of HSM Algorithm

Figure 5.4 shows the code for the module "Test". This module generates the test vectors to test the HSM_32 module. It implements brute force method of testing generating every possible combination of the inputs. It checks the correctness of the output. In case of error, it logs the output in the file "file1.out".

Figure 5.5 shows the code for the module "Top". It is the top-most module which instantiates the Test and HSM_32 modules and connects them.

The simulation was done using the Cadence NC-Verilog simulator.

```
module Test( Product, X, Y );

      // Port Declaration
      output      X, Y;
      input       Product;

      // Port type declaration
      reg   [31:0]      X, Y;
      wire  [63:0]      Product;

      // Internal variable declaration
      real        i, j;       // variables for loop structures
      integer     file1;      // file pointer to dump the simulation output

      // Initialize the variables and open an ASCII file
      initial
      begin
            X = 0;
            Y = 0;
            i = 0;
            j = 0;
            file1 = $fopen("file1.out");
      end

      always
      begin
            for( i = 0; i < pow(2.0, 32.0) - 1.0; i = i + 1 )
            begin
                  X = i;
                  for( j = 0; j < pow(2.0, 8.0) - 1.0; j = j + 1 )
                  begin
                        #1 Y = j;
                        // If Product is not equal to the product of X and Y, Generate Error Message
                        if( Product != ( X * Y ) )
                              $fdisplay(file1, "X = ", X, "\tY = ", Y, "\tProduct = ", Product, "\tError!!");
                              //$display("X = %d \t Y = %d \t Product = %d\t\tError", X, Y, Product);
                  end
                  $finish;
            end
            $finish;
      end

endmodule
```

Figure 5.4 Code of the Test module which generates test vectors for HSM_32 module

```
module Top;

    // Internal Variable declaration
    wire  [63:0]     product;
    wire  [31:0]     x, y;

    // Module Instantiation
    HSM_32 hsm1( product, x, y);
    Test   test1( product, x, y);

endmodule
```

Figure 5.5 Code of the Top module which instantiates HSM_32 and Test modules

5.3 Analysis of Results

The simulation generated errors. A small segment of error log is as shown below:

```
X =          1    Y =          1    Product =          0    Error!!
X =          1    Y =          2    Product =          1    Error!!
X =          1    Y =          3    Product =          1    Error!!
X =          1    Y =          4    Product =          3    Error!!
X =          1    Y =          5    Product =          3    Error!!
X =          1    Y =          6    Product =          5    Error!!
X =          1    Y =          7    Product =          5    Error!!
X =          1    Y =          8    Product =          7    Error!!
X =          1    Y =          9    Product =          7    Error!!
X =          1    Y =         10    Product =          9    Error!!
```

Now let us look at the detailed log as shown below of one of the transaction that generated the error when X = 1 and Y = 6.

```
X          = 00000000000000000000000000000001 (      1)
Y          = 00000000000000000000000000000110 (      6)
a          = 00000000000000000000000000000011 (      3)
b          = 00000000000000000000000000000010 (      2)
k          = 0000000000000011 (   3)
l          = 0000000000000010 (   2)
(a - k)    = 0000000000000000 (   0)
(b - l)    = 0000000000000000 (   0)
k * k      = 00000000000000000000000000001001 (      9)
```

```
I * I          = 00000000000000000000000000000100 (      4)

(a - k)(a - k) = 00000000000000000000000000000000 (      0)

(b - I)(b - I) = 00000000000000000000000000000000 (      0)

2.k.(a - k)    = 00000000000000000000000000000000 (      0)

2.I.(b - I)    = 00000000000000000000000000000000 (      0)

Product=
00000000000000000000000000000000000000000000000000000000000000101 (5)
```

Now let us analyze the calculations step by step.

$$a = (X + Y) \; / \; 2 = (1_2 + 110_2) / 10_2 = 111.1_2 = 3.5$$

As we can see the value of 'A' in the simulation equals the floor of the average of X and Y. Therefore, instead of 3.5, 'A' equals 3. It is this loss of fractional value when we calculated the value of 'A' and 'B' that caused the error in our first set of behavioral simulations. The next section discusses the mathematical analysis of this loss of accuracy.

5.4 Analysis of the Loss of Accuracy

Now as discussed in section 5.3, the fractional value is lost when calculating the values of a and b. The fractional value of 0.1_2 is lost when we have a '1' in the Least Significant Bit (LSB) position of the results of operations (X + Y) and (X – Y) and the values are then shifted one bit to the right in order to calculate the value of a and b. The only way we can get '1' in LSB when we calculate (X + Y) and (X – Y) is that only one of the two values, X and Y, has a 'one' in the LSB. Also, the loss in the accuracy equals 0.1_2.

Now let us try to find the mathematical expression of the value that will compensate for this loss in accuracy. Let Δa be the loss in accuracy of 'A' and Δb be the loss in accuracy of 'B'. Now as explained before,

$$\Delta a = \Delta b = 0.1_2 \qquad \text{--- (5.1)}$$

From equation 3.2 we have:

$$X \times Y = a^2 - b^2 \qquad \text{--- (5.2)}$$

Now replacing 'A' with "$a + \Delta a$" and 'B' with "$b + \Delta b$" in equation (5.2) for the condition when we have loss in the accuracy, we have:

$$X \times Y = (a + \Delta a)^2 - (b + \Delta b)^2$$

$$= (a^2 + (10)_2 . a . \Delta a + \Delta a^2) - (b^2 + (10)_2 . b . \Delta b + \Delta b^2)$$

On rearranging we have,

$$X \times Y = (a^2 - b^2) + (10)_2 . (a . \Delta a - b . \Delta b) + (\Delta a^2 - \Delta b^2) \qquad \text{--- (5.3)}$$

From equation (5.1) and (5.3) we have:

$$X \times Y = (a^2 - b^2) + (10)_2 . (0.1)_2 . (a - b)$$

$$= (a^2 - b^2) + (a - b) \qquad \text{--- (5.4)}$$

From equation 5.4 it is clear that the error in the product of X and Y is:

$$\Delta_{Product} = a - b$$

$$= [\{ (X + Y) / 2 \} - \{ (X - Y) / 2 \}]$$

$$= Y \qquad \text{--- (5.5)}$$

Now we have made one assumption in this derivation. 'B' will always be a positive value as we are taking the absolute value of b. In that case $\Delta_{Product}$ will be equal to the smaller of the two numbers X and Y.

Hence, we can summarize the analysis of the loss of accuracy as follows:

1. The accuracy is lost when only one of the two values X and Y has a '1' in the LSB position.
2. The accuracy can be adjusted by adding the smaller of the two values X and Y to the calculated product of X and Y.

5.5 Modification in the Implementation of the Algorithm

Figure 5.6 shows the new flowchart that adjusts the accuracy whenever there is deviation from the ideal result. Figures 5.7 (a) and (b) show the implementation of HSM_32 module with the accuracy adjustment. The simulation results with this modification resulted in zero errors.

5.6 Conclusion

In this chapter a behavioral simulation of the HSM algorithm was done in Verilog. The simulation results detected a discrepancy in the implementation of the algorithm that resulted in a deviation from the desired result. Mathematical analysis was done in order to derive the value of the error. The algorithm was modified such that an added term automatically adjusts the accuracy so there is no longer a loss in accuracy.

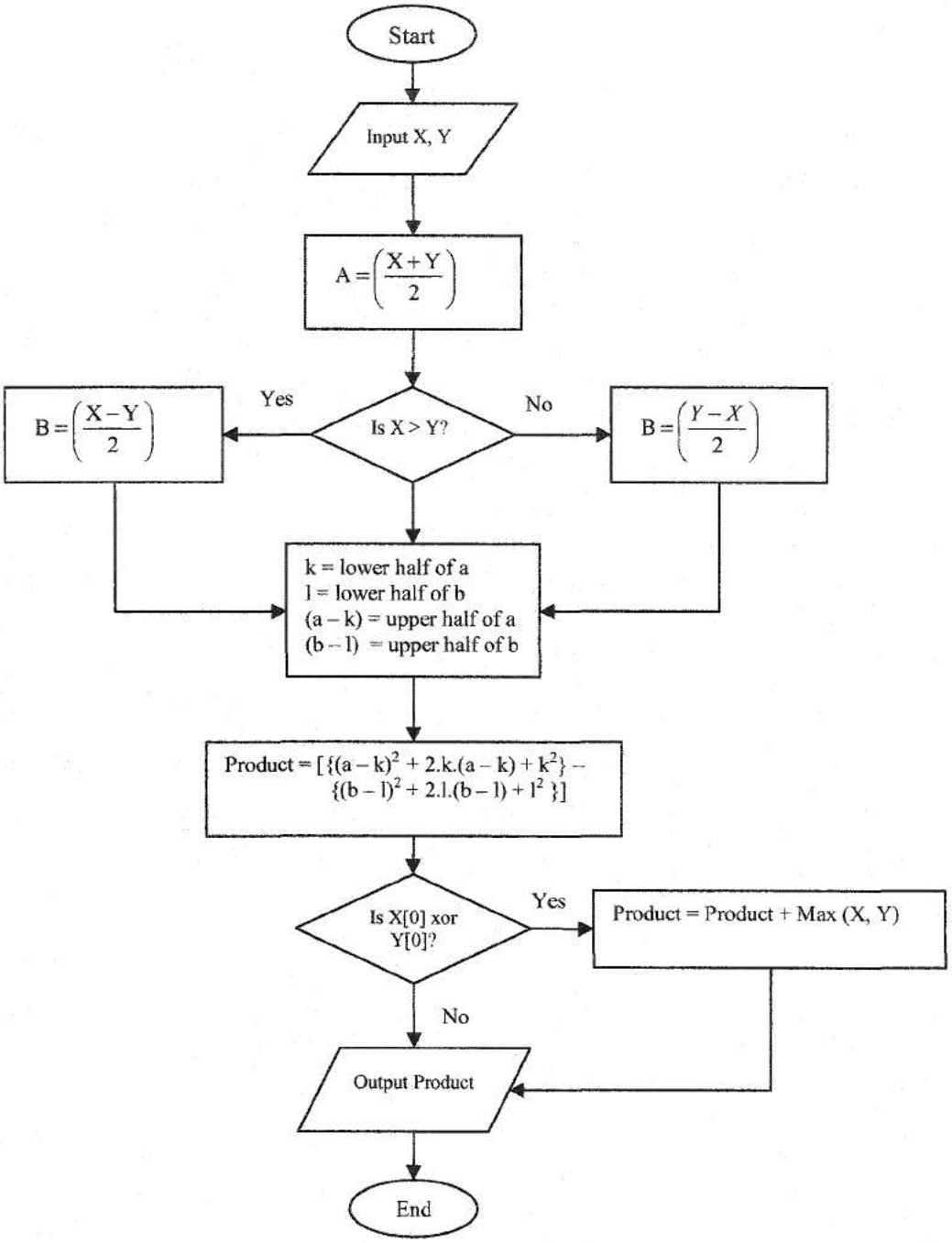

Figure 5.6 The flowchart of implementation of the HSM algorithm with accuracy adjustment

```
module HSM_32 (Product, X, Y);

      // Port Declaration
      output Product;         // Product of X and Y
      input  X, Y;            // Input Values

      reg   [63:0]      Product;
      wire  [31:0]      X, Y;

      // Declaration of internal variables
      reg   [31:0]      a, b;                         // a = (X + Y)/2
                                                      // b = (X - Y)/2
      reg   [15:0]      k, l;                         // k = a folding factor
                                                      // l = another folding factor
      reg   [15:0]      a_k, b_l;                     // a_k = a - k as in equation 3.6
                                                      // b_l = b - k as in equation 3.6
      reg   [31:0]      k_square, l_square;           // k_square = k * k
                                                      // l_square = l * l
      reg   [31:0]      a_k_square, b_l_square;       // a_k_square = (a - k)(a - k)
                                                      // b_l_square = (b - l)(b - l)
      reg   [31:0]      k2_times_a_k, l2_times_b_l;   // k2_times_a_k = 2.k.(a - k)
                                                      // l2_times_b_l = 2.l.(b - l)

      // Initialization of internal registers
      initial
      begin
            Product = 0;
            a = 0; b = 0;
            k = 0; l = 0;
            a_k = 0; b_l = 0;
            k_square = 0; l_square = 0;
            a_k_square = 0; b_l_square = 0;
            k2_times_a_k = 0; l2_times_b_l = 0;
      end

      // Execute the block if any of the input X or Y changes
      always @(X or Y)
      begin

            a = (X + Y)/2;    //    a = average of sum of X and Y

            // Find the absolute difference between X and Y
            if( X >= Y)
                  b = (X + (~Y) + 1)>>1;
            else
                  b = ((~X) + Y + 1) >> 1;



            // Folding a and b at mid-point.
            // k = lower half of a and a - k = upper half of a
            // l = lower half of b and b - l = upper half of b
            k = a[15:0];
            l = b[15:0];
            a_k = a[31:16];
            b_l = b[31:16];


            // Calculate the six terms as in equation 3.6
```

Figure 5.7 (a) The modified behavioral description of HSM_32 module in Verilog which implements the dynamic accuracy adjustment

```
        // Calculate the six terms as in equation 3.6
        k_square = k * k;           // k_square       = square of k
        l_square = l * l;           // l_square       = square of l
        a_k_square = a_k * a_k;     // a_k_square     = square of (a - k)
        b_l_square = b_l * b_l;     // b_l_square     = square of (b - l)

        k2_times_a_k = 2 * k * a_k;  // k2_times_a_k = 2 times product of k and (a - k)
        l2_times_b_l = 2 * l * b_l;  // l2_times_b_l = 2 times product of l and (b - l)


        // Accumulate the six terms in equation 3.6. Calculate the product.
        // The trailing zeros are added at this stage.
        Product = (k_square + {a_k_square, 16'b0} + {k2_times_a_k, 8'b0})
                  - (l_square  + {b_l_square, 16'b0} + {l2_times_b_l, 8'b0});

        // Check if there is loss of accuracy. If so, adjust the accuracy.
        if ( X[0] ^ Y[0] == true )
        begin
              if ( X >= Y)
                    Product = Product + Y;
              else
                    Product = Product + X;
        end

        $display("X \t\t\t = %-b  (%d)\n", X, X);
        $display("Y \t\t\t = %-b  (%d)\n", Y, Y);
        $display("a \t\t\t = %-b  (%d)\n", a, a);
        $display("b \t\t\t = %-b  (%d)\n", b, b);
        $display("k \t\t\t = %-b  (%d)\n", k, k);
        $display("l \t\t\t = %-b  (%d)\n", l, l);
        $display("(a - k) \t\t = %-b  (%d)\n", a_k, a_k);
        $display("(b - l) \t\t = %-b  (%d)\n", b_l, b_l);
        $display("k * k \t\t = %-b  (%d)\n", k_square, k_square);
        $display("l * l \t\t = %-b  (%d)\n", l_square, l_square);
        $display("(a - k)(a - k)\t = %-b  (%d)\n", a_k_square, a_k_square);
        $display("(b - l)(b - l)\t = %-b  (%d)\n", a_k_square, a_k_square);
        $display("2.k.(a - k) \t = %-b  (%d)\n", k2_times_a_k, k2_times_a_k);
        $display("2.l.(b - l) \t = %-b  (%d)\n", l2_times_b_l, l2_times_b_l);
        $display("Product \t\t = %-b  (%d)\n", Product, Product);

    end

endmodule
```

Figure 5.7 (b) The modified behavioral description of HSM_32 module in Verilog which implements the dynamic accuracy adjustment

CHAPTER 6

DESIGN AND SIMULATION OF HSM ARCHITECTURE

In the previous chapter we discussed the behavioral description and the simulation of the HSM algorithm using Verilog. A behavioral model of the HSM algorithm and the corresponding flowchart were developed. This chapter explains the use of the developed abstract model to map each step to the corresponding digital logic blocks. Then for every digital logic block, behavioral models are developed. And finally, the behavioral model of the HSM architecture is developed and simulated.

6.1 Design of HSM Architecture

The HSM requires two inputs, the multiplier and the multiplicand in unsigned integer form. Using the HSM algorithm implemented as in Figure 5.6, HSM folds the input based on the folding factor and calculates the product of the two inputs. If the multiplier and the multiplicands are in the sign-magnitude form or in 2's complement form, then the magnitude of the operands needs to be calculated by converting them from signed form to the unsigned form. Figure 6.1 shows how absoluting logic is used to give absolute value X and Y of 32-bit operands P and Q respectively. The output of 32-bit HSM is the 64-bit product of P and Q. The product is then complemented if only one of the two operands P and Q was negative i.e. the Most Significant Bit (MSB) of P or Q is '1'.

Now let us look at the flowchart shown in Figure 5.6 which implements the HSM.

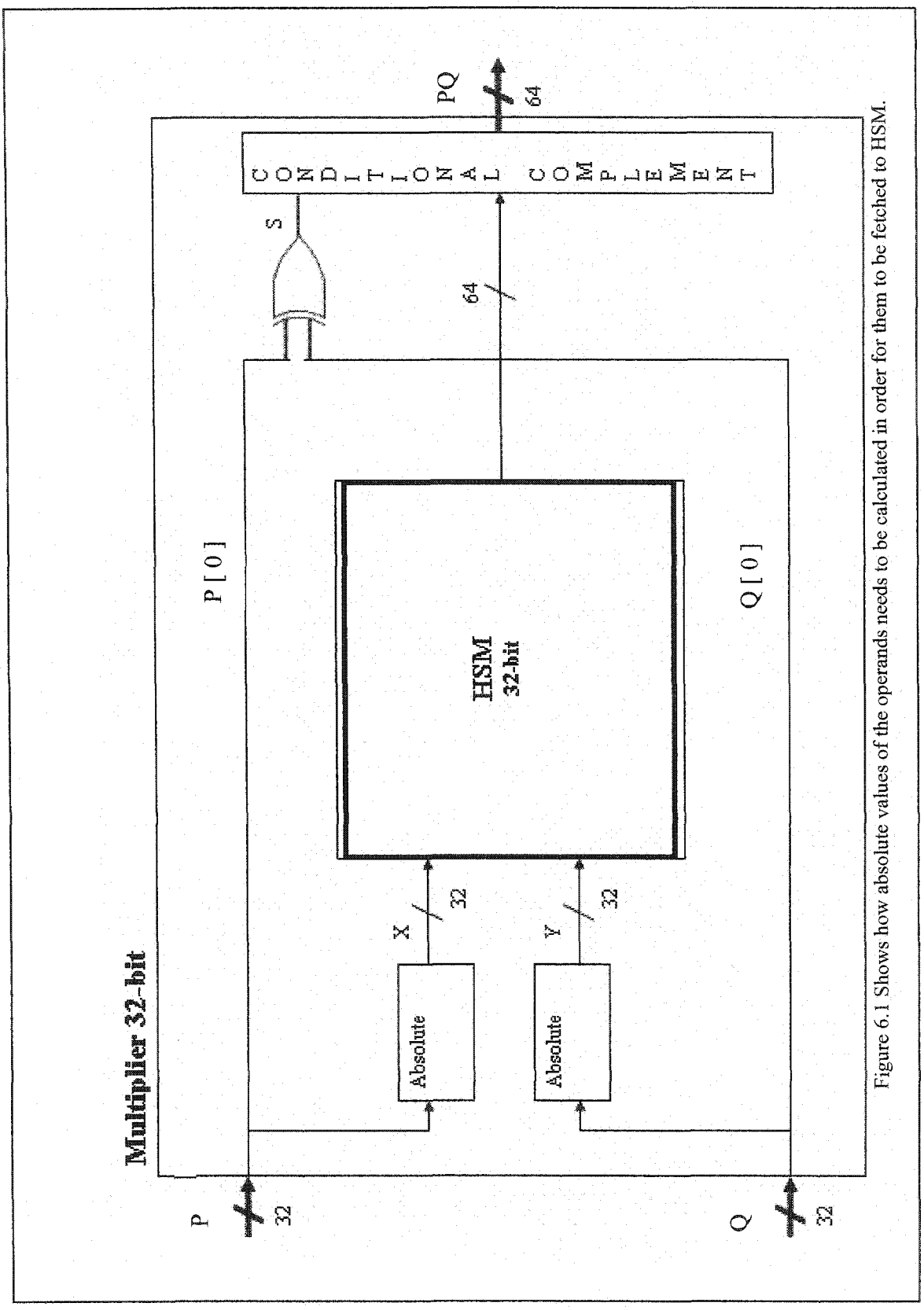

Figure 6.1 Shows how absolute values of the operands needs to be calculated in order for them to be fetched to HSM.

algorithm and take each step and map it to the corresponding digital logic block. The steps involved in HSM algorithm are as follows:

1. Calculate the value of 'A'
2. Calculate the absolute value of 'B'
3. Calculate the accuracy adjustment value
4. Calculate the value of $(a - k)^2$
5. Calculate the value of k^2
6. Calculate the value of $(b - 1)^2$
7. Calculate the value of 1^2
8. Calculate the value of $2.b.(b - 1)$
9. Calculate the value of $2.a.(a - k)$
10. Accumulate the values calculated in step 3 through 9 and add the trailing zeros before accumulating

Figure 6.2 shows the architecture of 32-bit HSM. The main logical blocks of the architecture are ROMs, 16-bit multipliers, adders, inverters, and multiplexers. Now let us take each step in the HSM algorithm as explained above and try to understand which blocks in Figure 6.2 perform those steps.

In step 1 the value of 'A', i.e. the average of X and Y, is calculated. The 32-bit adder "P5 S1" performs the operation (X + Y). Division by 2 operations can be achieved by right shifting the number. Therefore, upper 31-bits of 'A' will give the average of X and Y.

In step 2 the absolute value of 'B', i.e. the difference of X and Y, is calculated. Subtraction is performed by the addition of 2's complement of the value that is to be

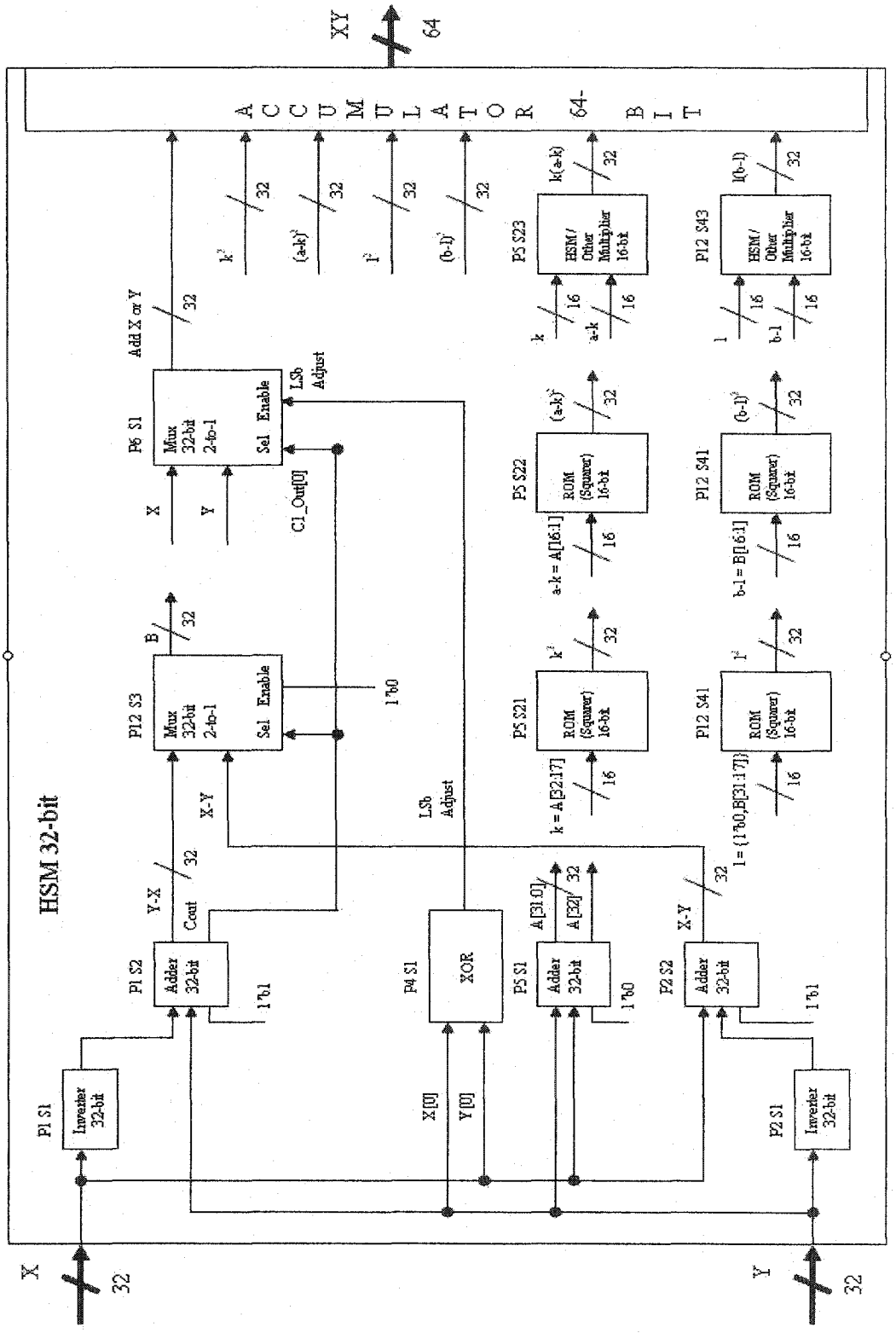

Figure 6.2 The architecture of 32-bit HSM

subtracted. Because this is a combinational circuit, the 2's complement of both X and Y is needed for this purpose. The inverters "P1 S1" and "P2 S2" give 1's complement of X and Y respectively. But 2's complement of a number is equal to 1 added to the 1's complement of the number. This addition of the value '1' to the 1's complement can be combined with the operations (X – Y) and (Y - X) which are performed by adders "P1 S2" and "P2 S2". '1' that needs to be added to (X – Y) and (Y - X) appears as the carry-in signal to the adders performing these operations. The carry-out signal of adders "P1 S2" and "P2 S2" can be used to determine which of the operations gives the absolute value of the difference of X and Y. If the carry-out signal of "P1 S2" is high, it means subtraction (Y – X) resulted in a negative value and if it is low, the subtraction (X – Y) resulted in a positive value. Therefore, we would use this carry-out signal as the select line for 32-bit 2-to-1 multiplexer "P12 S3". "P12 S3" has two inputs (X – Y) and (Y – X). It will select the positive value out of the two inputs and hence carry-out signal of adder "P1 S2" is used as the select line. We could have also used the inverted carry-out signal of adder "P2 S2" as the select line. The upper 31-bits of the output of the multiplexer "P12 S3" will give the value of 'B'.

In step 3, the accuracy adjustment value is calculated. As explained in the section 5.4, this value is equal to the smaller of the two inputs X and Y. Just like in step 2 we can use the carry-out signal from the adder "P1 S2" as the select line for the multiplexer "P6 S1", which will select smaller of the two inputs X and Y.

Steps 4 through 7 involve calculating squares of 16-bit values. The three ways in which we can perform this operation are as follows:

1. Using ROMs
2. Using combinational logic
3. Using 16-bit multiplier with both inputs equal to the value to be squared

In our case ROMs are used for input values having bit-width greater than 4-bit and combinational logic for input values having bit-width less than or equal to 4-bit.

Steps 8 and 9 involve multiplication of two 16-bit numbers and then doubling the product. The multiplication operation can be performed in the following three ways:

1. Using ROMs
2. Using combinational logic
3. Using 16-bit multiplier

In our case HSM is used to carry out multiplication when operand bit-width is greater than 2-bit and combinational logic is used when operand bit-width is less than or equal to 2-bit.

Step 10 involves the addition of all the values calculated in steps 4 through 9. It is not just direct addition of all the terms. While accumulating the values we will have to account for the trailing zeros. Figure 6.3 shows the 64-bit accumulator. The output of the accumulator will be the product of the two inputs X and Y. As shown in the Figure 6.3, the accumulator uses four 64-bit adders.

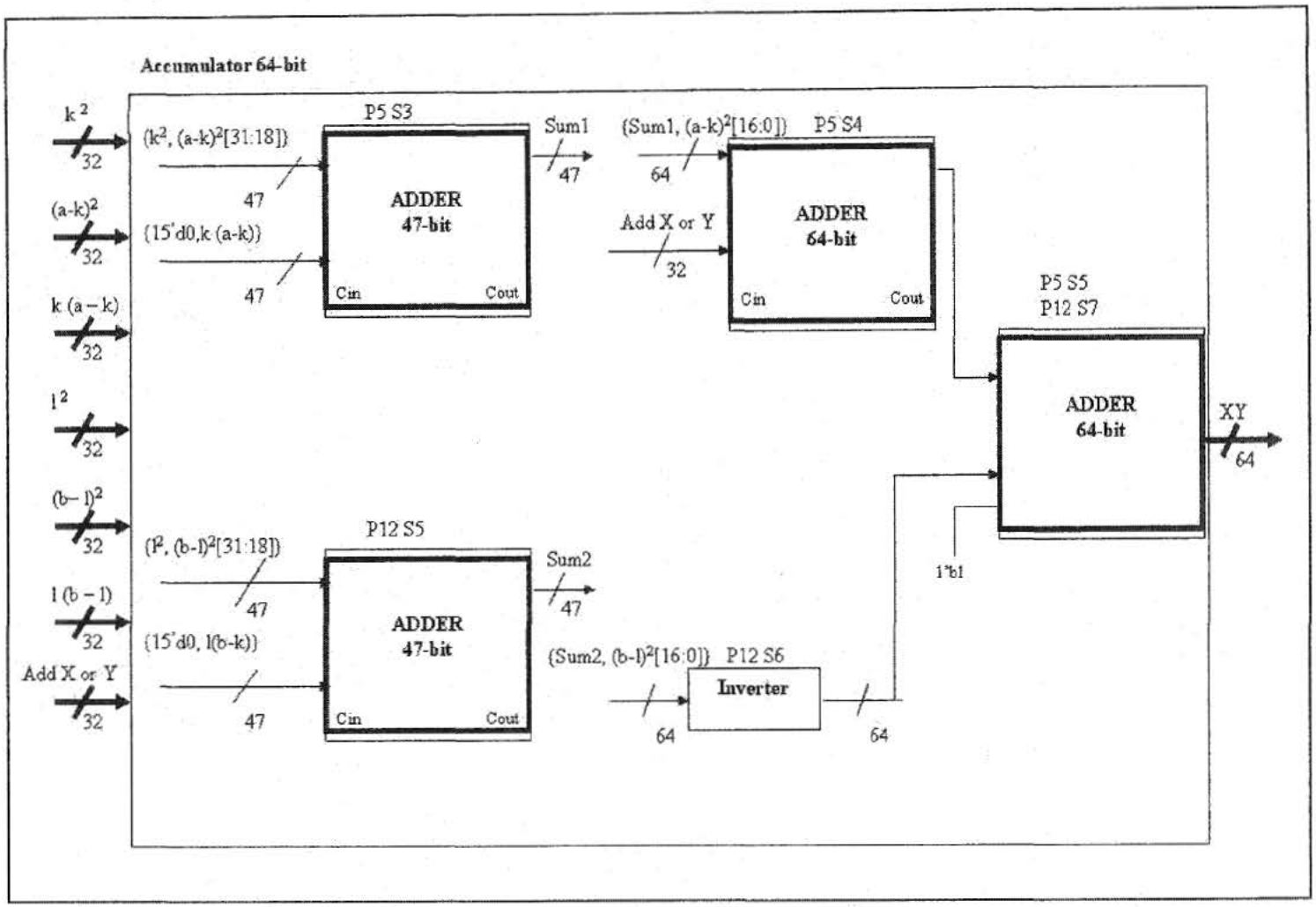

Figure 6.3 The architecture of 64-bit Accumulator

6.2 Scalability of the Architecture

The multiplier is scalable which means that a fixed area multiplication module can handle operands of any size and also the word size can be selected based on the performance requirements [SaTe00]. HSM architecture is a highly scalable architecture because of the following reasons:

1. It uses smaller multipliers as building blocks that have the same architecture. For example, 32-bit HSM has two 16-bit HSMs that have the same architecture as the 32-bit HSM. 16-bit HSMs in turn have 8-bit HSMs as the building blocks as shown in Figure 6.4 and 6.5. For folding factor other than half, the size of the smaller HSMs will vary but their architecture will still remain the same.
2. All the building blocks are scalable: ROMs, HSMs and Accumulators
3. Folding capability is the reason why this architecture can handle operands of any size to suit the area and performance needs
4. The building blocks can be implemented as ROMs, multipliers, and/or combinational logic giving tremendous flexibility in optimizing the architecture for area, speed, or power. Further, this method can take advantage of any advances in hardware implementations and algorithms, so far as the building blocks are concerned. Thus an 8-bit Wallace Multiplier and a 16-bit CLA (Carry Look-Ahead) adders were used in our benchmark comparisons.

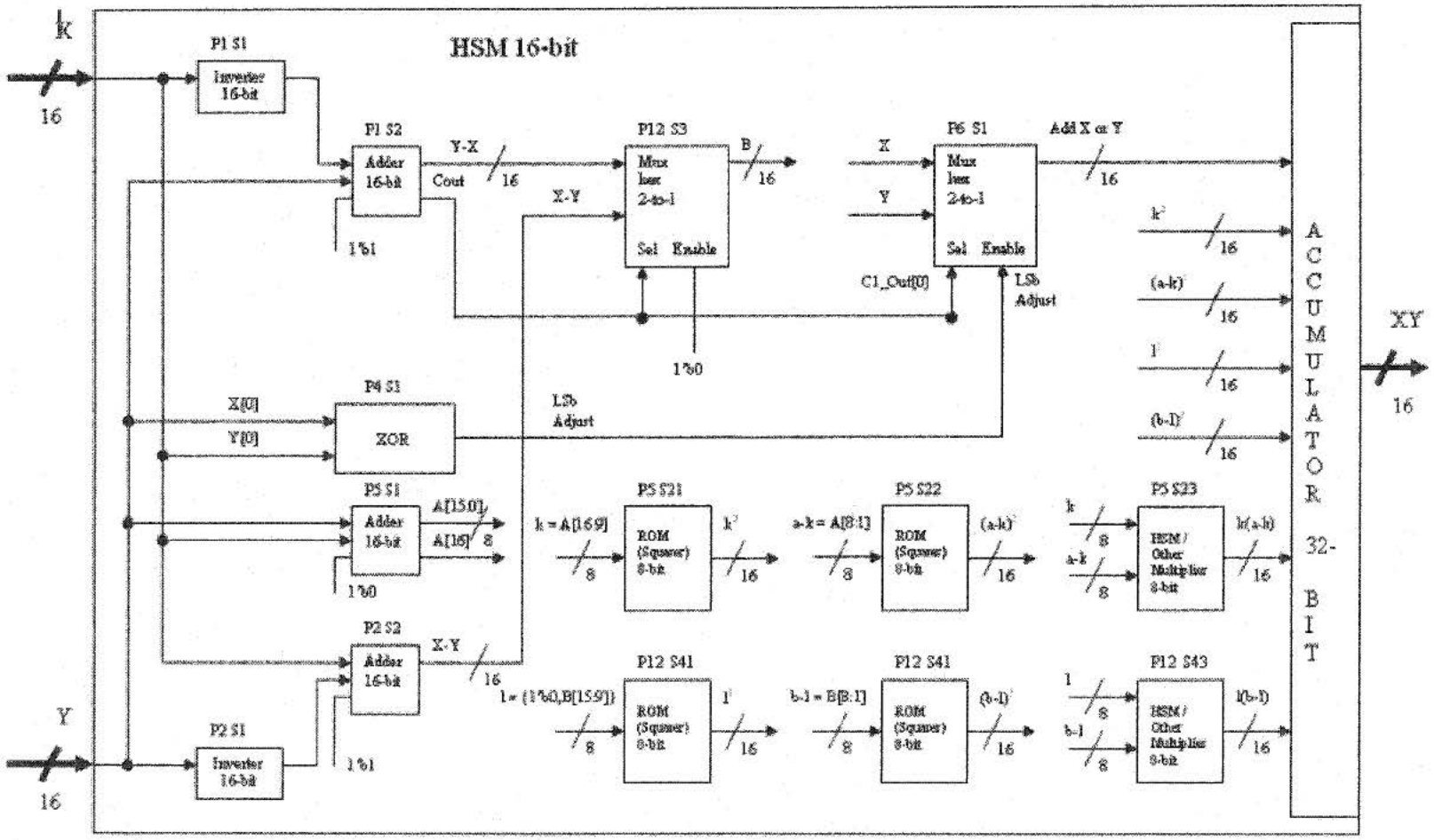

Figure 6.4 The architecture of 16-bit HSM

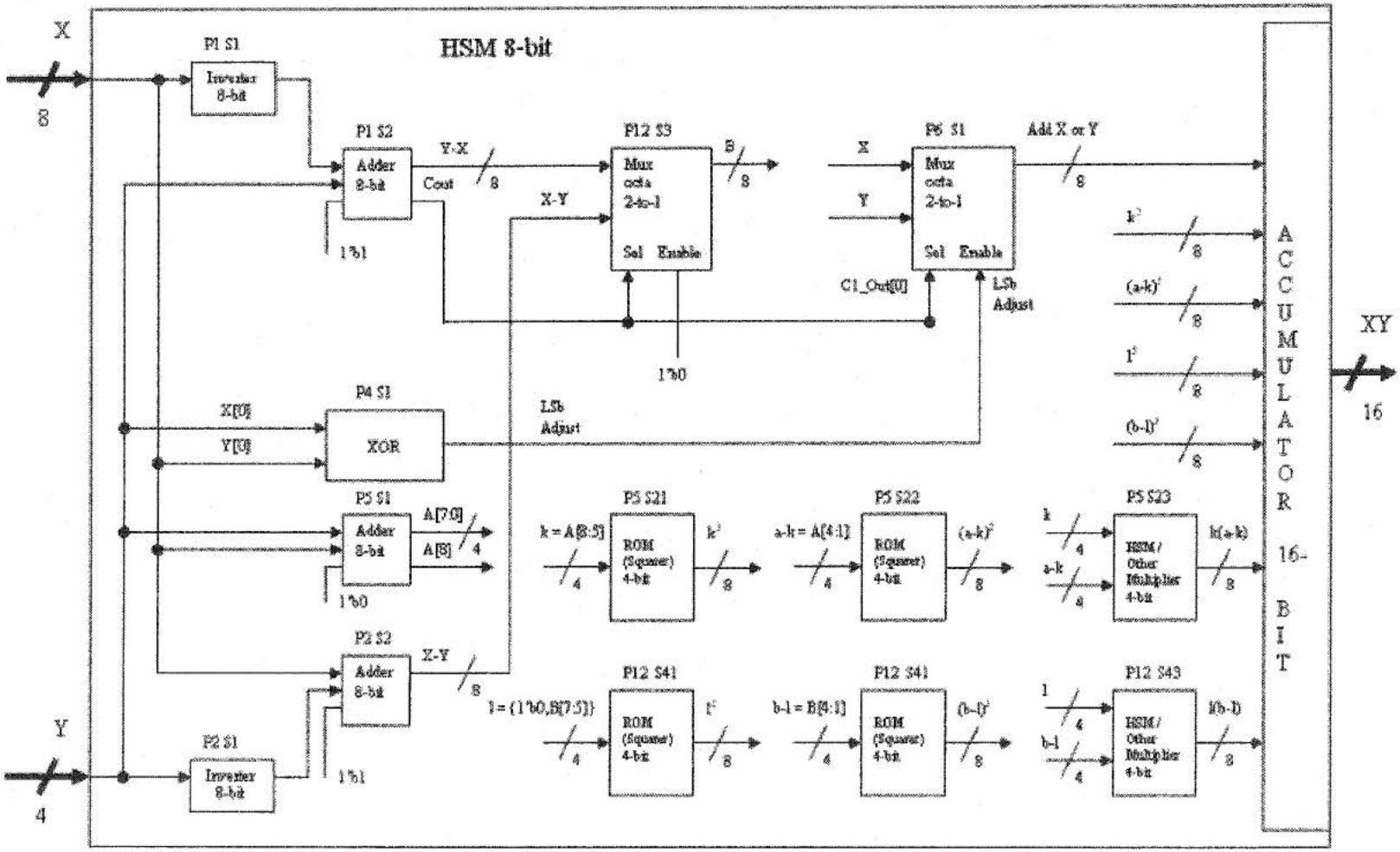

Figure 6.5 The architecture of 8-bit HSM

6.3 Explanation of 8-bit HSM Code

Figure 6.5 shows the architecture of 8-bit HSM. It comprises of the following major building blocks:

1. 4-bit square ROMs
2. 8-bit 2-to-1 multiplexer
3. 16-bit Accumulator
4. 8-bit Inverter
5. 4-bit HSM

Behavioral models for all of the building blocks were developed. The code for each of them is explained in the following subsections.

6.3.1 4-bit Square ROM

The behavioral description of 4-bit square ROMs is as shown in the Figure 6.6. The details of the module are as shown in Table 6.1.

Name of the module	ROM_4bit
Input Ports	In (4-bit wide vector)
Output Ports	Out (8-bit wide vector)
Function	Squares the input

Table 6.1 Specifications of 4-bit Square ROM module

Note that the behavior of the 4-bit square ROM was captured by the equation instead of using writing detailed structural details:

Out = In * In

```
/*=====================================================
      4-bit Square ROM Model
      File name: ROM_4bit.vb
      Module name: ROM_4bit
      Input: In (4-bit)
      Output: Out (8-bit)
      Instances:
            None

      It is the Square ROM 4-bit wide that is it takes
      in 4-bit wide input (In) and outputs its square (Out)
      It is behavioral code.
=====================================================*/


module ROM_4bit ( Out, In );


      // Port declaration
      output      Out;
      input [3:0] In;

      reg   [7:0] Out;


      // Variable Initialization
      initial
            Out = 0;

      // Executed when there are any changes in the value
      // of In
      always @(In)
      begin
            Out = In * In;
      end

endmodule
```

Figure 6.6 Verilog code of 4-bit ROM module (ROM_4bit)

6.3.2 8-bit 2-to-1 multiplexer

Behavioral description of the 8-bit 2-to-1 multiplexer is as shown in the Figure 6.7. The details of the module are as shown in Table 6.2.

Name of the module	Mux_octa_2_to_1
Input Ports	In1 (8-bit wide vector) In1 (8-bit wide vector) Select (1-bit signal) Enable (1-bit signal)
Output Ports	Out (8-bit wide vector)
Function	If Enable is true then: • If Select = 1 select In1 else select In2

Table 6.2 Specifications of 8-bit 2-to-1 Multiplexer

6.3.3 16-bit Accumulator

Behavioral description of the 16-bit Accumulator is as shown in the Figure 6.8. The accumulator calculates the product by adding the 6 terms in equation (3.6) and the accuracy adjustment value. But it is not direct addition of the seven input terms because trailing zeros of all the terms have to be taken into consideration while adding the values as explained in the section 3.3 besides 2's complement has to be performed to carry out subtraction. Every term that has (a – k) or (b – k) as a factor, 4 trailing zeros have to be added to the term for each instance of these factors because we chopped off these zeros during the folding operation. For instance, the term $(a - k)^2$ will have 8 trailing zeros

whereas the term k . (a – k) will have 4 trailing zeros in addition to one for left shifting the value in order to multiply the number by 2. The details of the trailing zeros added to each input is as shown in Table 6.4. The details of the module are in Table 6.3.

```
module Mux_octa_2_to_1(Out, In1, In2, Select, Enable)

      output            Out;
      reg   [7:0] Out;
      input [7:0] In1, In2;
      input       Select, Enable;

      parameter   gate_count  = 27;

      always @(In1 or In2 or Select or Enable)
      begin
            if(Enable==1)
                  Out = 0;
            else
            begin
                  if(Select == 1)
                        Out = In1;
                  else
                        Out = In2;
            end
      end


endmodule
```

Figure 6.7 Verilog code of 8-bit 2-to-1 multiplexer

```
1   module Accumulator_16bit (Out, ak2, kak, k2, bl2,
2                             lbl, l2, error_adjustment);
3
4         output      [15:0]      Out;
5         input       [7:0]       ak2;
6         input       [7:0]       kak;
7         input       [7:0]       k2;
8         input       [7:0]       bl2;
9         input       [7:0]       lbl;
10        input       [7:0]       l2;
11        input       [7:0]       error_adjustment;
12
13        assign      Out =  ak2 + {kak, 3'd0} + {k2, 4'd0} + (~bl2) +
14                    (~{lbl, 3'd0}) + (~{l2, 4'd0}) + 3 + error_adjustment;
15
    endmodule
```

Figure 6.8 Verilog code of 16-bit accumulator

Name of the module	Accumulator_16bit
Input Ports	ak2 (8-bit wide vector) kak (8-bit wide vector) k2 (8-bit wide vector) bl2 (8-bit wide vector) lbl (8-bit wide vector) l2 (8-bit wide vector) error_adjustment (8-bit wide vector)
Output Ports	Out (16-bit wide vector)
Function	Add the values of all the seven inputs after adding the trailing zeros as per the specification from Table 6.4

Table 6.3 Specifications of 16-bit Accumulator

Term in equation (3.6)	Input name in the program	Number of trailing zeros
$(a-k)^2$	ak2	8
$(b-1)^2$	bl2	8
$2.k.(a-k)$	kak (Note: multiplication by 2 is still not performed)	4 + 1 (For multiplication by 2) = 5
$2.1.(b-1)$	blb (Note: multiplication by 2 is still not performed)	4 + 1 (For multiplication by 2) = 5

Table 6.4 Shows number of trailing zeros to be added to each input in 16-bit Accumulator

6.3.4 Inverters

The inverters are used to calculate 1's complement of the input number. The Verilog code of the inverter is as shown in Figure 6.9. The details of the module are shown in Table 6.5.

```
1     module      Inverter_8bit(Out, In);
2
3           output      [7:0] Out;
4           input [7:0] In;
5
6           parameter   gate_count  = 8;
7
8           assign      Out = ~In;
9
10    endmodule
11
```

Figure 6.9 Verilog code of 8-bit Inverter (Inverter_8bit module)

Name of the module	Inverter_8bit
Input Ports	In (8-bit wide vector)
Output Ports	Out (8-bit wide vector)
Function	Add the values of all the seven inputs after adding the trailing zeros as per the specification from Table 6.4

Table 6.5 Specifications of 8-bit inverter

6.3.5 4-bit HSM

The Verilog code of 4-bit HSM is as shown in Figure 6.10. As discussed earlier, 4-bit HSM and 8-bit HSM have similar architecture except bit-width of operands of the building blocks in 4-bit HSM is of half the size as compared to 8-bit HSM. The details of the 4-bit HSM module are as shown in Table 6.6.

6.3.6 8-bit HSM

The Verilog code of 8-bit HSM is as shown in Figure 6.11. This module instantiates all the modules described in sections 6.3.1 through 6.3.5. The details of the module are as shown in Table 6.7.

6.4 Brute-Force Testing

The test module for testing the HSM_8bit module employed brute-force method of testing as explained in section. A part of the result is as shown in the Figure 6.12.

Name of the module	HSM_4bit
Input Ports	X (4-bit wide vector) Y (4-bit wide vector)
Output Ports	Out (8-bit wide vector)
Function	Outputs the product of the two input
Sub-modules	Square_ROM_2bit (2-bit square ROM) Multiply_ROM_2bit (2-bit ROM for multiplication) Accumulator_8bit (8-bit accumulator) Adder_4bit (4-bit adder) Inverter_4bit (4-bit inverter) Mux_quad_2_to_1 (4-bit 2-to-1 multiplexer)

Table 6.6 Specifications of 4-bit HSM (HSM_4bit module)

Name of the module	HSM_8bit
Input Ports	X (8-bit wide vector) Y (8-bit wide vector)
Output Ports	Out (8-bit wide vector)
Function	Outputs the product of the two input
Sub-modules	Square_ROM_4bit (4-bit square ROM) HSM_4bit (4-bit HSM for multiplication) Accumulator_16bit (8-bit accumulator) Adder_4bit (8-bit adder) Inverter_4bit (8-bit inverter) Mux_octa_2_to_1 (8-bit 2-to-1 multiplexer)

Table 6.7 Specifications of 8-bit HSM (HSM_8bit module)

```
module HSM_4bit ( Out, X, Y );

      // Port Declaration
      output      [7:0] Out;
      input [3:0] X;
      input [3:0] Y;

      //Internal variables
      wire  [4:0] ai;
      wire  [4:0] bi;
      wire  [3:0] k2;
      wire  [3:0] ak2;
      wire  [3:0] l2;
      wire  [3:0] bl2;
      wire  [3:0] kak;
      wire  [3:0] lbl;
      wire  [1:0] C1_Out;

      // aliases
      `define     k     ai[4:3]
      `define     ak    ai[2:1]
      `define     l     bi[4:3]
      `define     bl    bi[2:1]

      // Internal variables
      wire  [3:0] error_adjustment;
      wire  [3:0] xbar, ybar;
      wire  x0bar, y0bar, error, a1, a2, a3, a4, o1, enable, select, n3;
      wire  errorbar;
      wire  [3:0] x_minus_y, y_minus_x;
      wire        Cout;

      //Module instantiations
      Square_ROM_2bit         Square_ROM_2bit_1(k2,`k);
      Square_ROM_2bit         Square_ROM_2bit_2(ak2, `ak);
      Square_ROM_2bit         Square_ROM_2bit_3(l2,`l);
      Square_ROM_2bit         Square_ROM_2bit_4(bl2, `bl);
      Multiply_ROM_2bit       Multiply_ROM_2bit_1(kak, `k, `ak);
      Multiply_ROM_2bit       Multiply_ROM_2bit_2(lbl, `l, `bl);
      Accumulator_8bit        Accumulator_8bit1(Out, ak2, kak, k2, bl2,
                                                lbl, l2, error_adjustment);
      Adder_4bit              Adder_4bit1(ai[3:0],ai[4],X[3:0],Y[3:0],1'b0);
                                    //=== ai = (X + Y)
      Inverter_4bit           Inverter_4bit1(xbar, X);
      Inverter_4bit           Inverter_4bit2(ybar, Y);
      Adder_4bit              Adder_4bit2(x_minus_y, , X, ybar, 1'b1);
      Adder_4bit              Adder_4bit3(y_minus_x, Cout, xbar, Y, 1'b1);

      // Check if error is there. X[0] xor Y[0]
      not   Not1(x0bar, X[0]);
      not   Not2(y0bar, Y[0]);
      and   And1(a1, x0bar, Y[0]);
      and   And2(a2, X[0], y0bar);
      or    Or1(error, a1, a2);
      not   Not3(errorbar, error);
      Mux_quad_2_to_1 Mux_quad_1(error_adjustment, X, Y, Cout, errorbar);
      Mux_quad_2_to_1 Mux_quad_2(bi[3:0], y_minus_x, x_minus_y, Cout, 1'b0);
      assign      bi[4] = 1'b0;

endmodule
```

Figure 6.10 Verilog code of 4-bit HSM (HSM_4bit module)

```
X=192, Y=  6 Out= 1152
X=193, Y=  6 Out= 1158
X=194, Y=  6 Out= 1164
X=195, Y=  6 Out= 1170
X=196, Y=  6 Out= 1176
X=197, Y=  6 Out= 1182
X=198, Y=  6 Out= 1188
X=199, Y=  6 Out= 1194
X=200, Y=  6 Out= 1200
X=201, Y=  6 Out= 1206
X=202, Y=  6 Out= 1212
X=203, Y=  6 Out= 1218
X=204, Y=  6 Out= 1224
X=205, Y=  6 Out= 1230
X=206, Y=  6 Out= 1236
X=207, Y=  6 Out= 1242
X=208, Y=  6 Out= 1248
X=209, Y=  6 Out= 1254
X=210, Y=  6 Out= 1260
X=211, Y=  6 Out= 1266
X=212, Y=  6 Out= 1272
X=213, Y=  6 Out= 1278
X=214, Y=  6 Out= 1284
X=215, Y=  6 Out= 1290
X=216, Y=  6 Out= 1296
X=217, Y=  6 Out= 1302
X=218, Y=  6 Out= 1308
X=219, Y=  6 Out= 1314
X=220, Y=  6 Out= 1320
X=221, Y=  6 Out= 1326
X=222, Y=  6 Out= 1332
X=223, Y=  6 Out= 1338
X=224, Y=  6 Out= 1344
X=225, Y=  6 Out= 1350
X=226, Y=  6 Out= 1356
X=227, Y=  6 Out= 1362
X=228, Y=  6 Out= 1368
X=229, Y=  6 Out= 1374
X=230, Y=  6 Out= 1380
```

Figure 6.11 Section of simulation result of 8-bit HSM

CHAPTER 7

RESULTS

In Chapter 6 the HSM was implemented and simulated in order to verify the architecture. This chapter shows how the HSM architecture was analyzed to get the gate count, the toggle (transition) count, and the delay.

7.1 Gate Count

The gate count is the number of equivalent nand or nor gates required for the implementation. The following are the basic building blocks of HSM:

1. Adders
2. Multiplexers
3. Inverters
4. Xor gate
5. Square ROMs
6. Product ROMs

Table 7.1 through 7.6 shows the gate counts of these building blocks for different input sizes [Mano00]. These are example tables with Ripple-Carry Adder. Similar tables exist for Carry Look-Ahead adders. The following are the building blocks of accumulator of HSM:

1. 4 Adders

2. 1 Inverter

Adder	Gate Count
5-bit	45
8-bit	72
11-bit	99
16-bit	144
23-bit	207
32-bit	288
47-bit	423
64-bit	576

Table: 7.1 Gate count of ripple-carry adders

Multiplexer	Gate Count
4-bit 2-to-1	15
8-bit 2-to-1	27

Table: 7.2 Gate count of multiplexer

Xor	Gate Count
2-input Xor	4

Table: 7.3 Gate count of Xor gate

Product ROM	Gate Count
2-bit	17

Table: 7.4 Gate count of Product ROM

Inverter	Gate Count
2-bit	2
4-bit	4
8-bit	8
16-bit	16
32-bit	32
64-bit	64

Table: 7.5 Gate count of Inverters

Square ROM	Gate Count
2-bit	5
4-bit	22
8-bit	360
16-bit	1575

Table: 7.6 Gate count of square ROMs

The gate count of the accumulators is calculated by adding up the gate counts of the building blocks from Table 7.1 through 7.6, and is shown in Table 7.7.

Size of Accumulator	Building blocks of Accumulator		
	Adders	**Inverters**	**Gate Count**
8-bit	2 5-bit Adders 2 8-bit Adders	1 8-bit Inverter	242
16-bit	2 11-bit Adders 2 16-bit Adders	1 16-bit Inverter	502
32-bit	2 23-bit Adders 2 32-bit Adders	1 32-bit Inverter	1022
64-bit	2 47-bit Adders 2 64-bit Adders	1 64-bit Inverter	2062

Table 7.7 Shows the calculation of gate count for accumulators using carry look-ahead adders

Now that gate counts for all the building blocks are available, the gate count for HSM can be calculated as shown in Table 7.8 (a). The gate counts for HSM implemented with CLA, and 8-bit Wallace multiplier as building blocks is shown in Table 7.8 (b).

Figure 7.1 shows the comparison of gate count of HSM (implemented with CLA, 8-bit Wallace multiplier as building blocks) and with that of two industry standard multipliers, Wallace and Array multipliers [CaSh97].

As we can see that the gate count of HSM is almost double as compared to that of Array and Wallace multipliers.

HSM	Gates		Adders		Inverters		Square ROMs		Product ROMs		Multiplexers		Accumulator		Gate Count
Size	Size	#	Size	#	Size	#	Size	#	Size	#	Size	#	Size	#	
4-bit	2-input Xor	1	4-bit	3	4-bit	2	2-bit	4	2-bit	2	4-bit 2-to-1	2	8-bit	1	448
8-bit	2-input Xor	1	8-bit	3	8-bit	2	4-bit	4	4-bit	2	8-bit 2-to-1	2	16-bit	1	1778
16-bit	2-input Xor	1	16-bit	3	16-bit	2	8-bit	4	8-bit	2	16-bit 2-to-1	2	32-bit	1	6596
32-bit	2-input Xor	1	32-bit	3	32-bit	2	16-bit	4	16-bit	2	32-bit 2-to-1	2	64-bit	1	22688

Table 7.8 (a) Shows the calculation for the gate count of HSM

HSM	Gates		Adders		Inverters		Square ROMs		Product ROMs		Multiplexers		Accumulator		Gate Count
Size	Size	#	Size	#	Size	#	Size	#	Size	#	Size	#	Size	#	
16-bit	2-input Xor	1	16-bit	3	16-bit	2	8-bit	4	8-bit	2	16-bit 2-to-1	2	32-bit	1	4716
32-bit	2-input Xor	1	32-bit	3	32-bit	2	16-bit	4	16-bit	2	32-bit 2-to-1	2	64-bit	1	19278

Table 7.8 (b) Shows the calculation for the gate count of HSM with CLA and 8-bit Wallace Tree Multiplier as building blocks

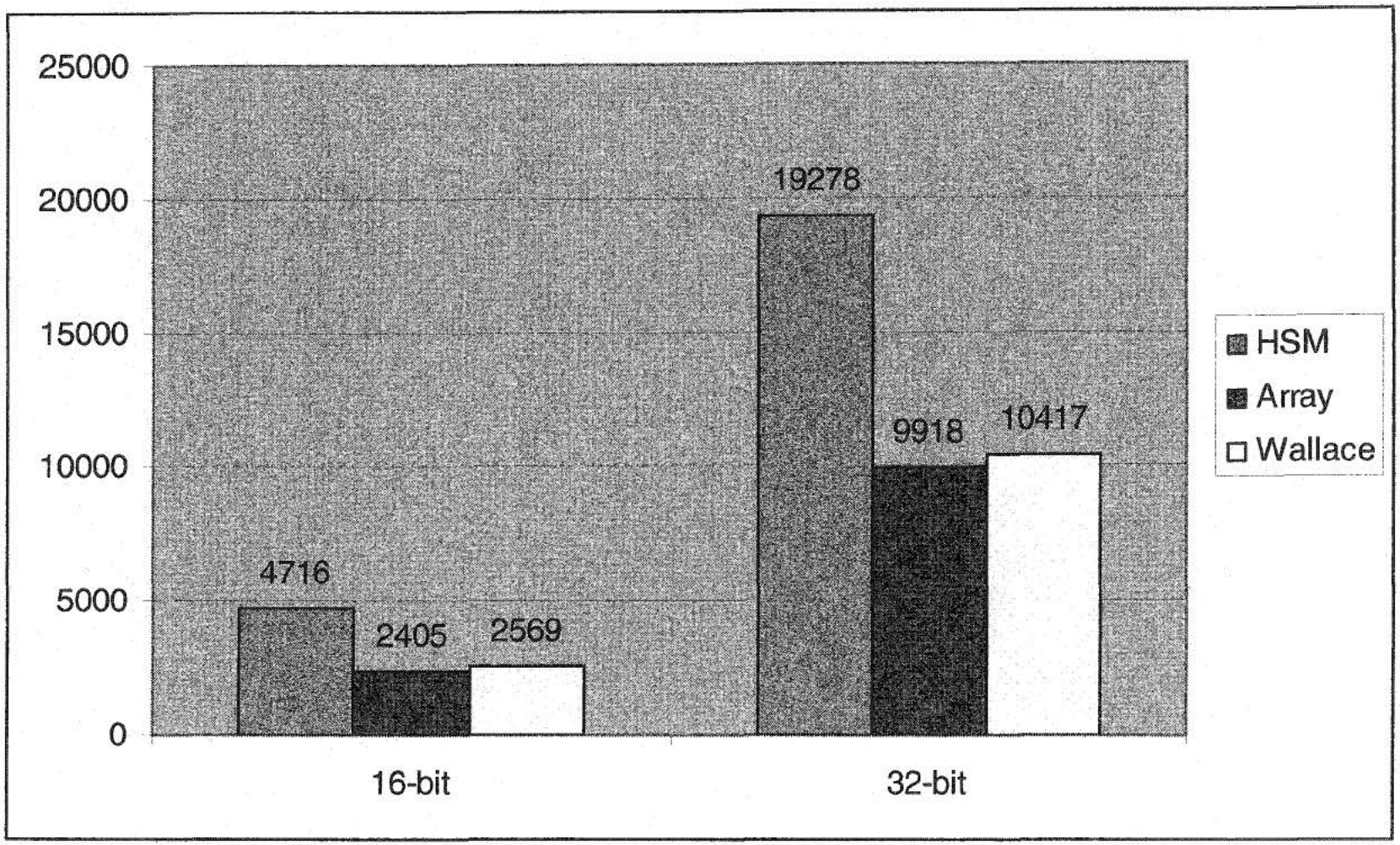

Figure 7.1 Comparison of gate counts of HSM, Wallace Multiplier and Array Multiplier

7.2 Toggle Count

Toggle count measures the number of '1 to 0' and '0 to 1' transitions. It refers to the toggle from '0' to '1' or vice versa. The Average Toggle Count (ATC) of HSM was calculated analytically. The ATC for each building blocks of HSM was determined analytically and was added up to give the ATC of HSM. The method of determining ATC for each of the building block is explained below:

1. Inverter: Inverter inverts each bit of the input to give the inverted output. Therefore the ATC is equal the bit-width of the input. For 4-bit and 8-bit inverters it is equal to 4 and 8 respectively.
2. Adder: The ATC for a 16-bit Ripple Carry adder is 90 [CaSh97]. Based on the size of the adder, ATC can be estimated. For 4-bit adder ATC is given by:

   ATC (4-bit Adder) = 4/16 * ATC (16-bit Adder) = (4 / 16) * 90 = 23.

   Similarly ATC of 8-bit Adder is given by:

   ATC (8-bit Adder) = 8 / 16 * ATC (16-bit Adder) = (8 / 16) * 90 = 45

   The CLA adder transitions were obtained from Callaway & Swatzlander [CaSh97].
3. Multiplexer: The average toggle is taken as the average of the maximum and minimum number of toggles. For 4-bit 2-to-1 multiplexer the maximum and minimum number of toggles possible is 4 and 2 respectively. Therefore, ATC of 4-bit 2-to-1 multiplexer is given by:

   ATC (4-bit 2-to-1 multiplexer) = (5 + 2) / 2 = 3.5
4. Product ROM: The ATC of 8-bit Modular Array multiplier is 583 [CaSh97]. The ATC of 16-bit Modular Array (MA) multiplier is 16 times that of 8-bit

multiplier. Therefore, factor of 1/12 and 1/44 is used for 4-bit and 2-bit multipliers respectively. For 2-bit multipliers ATC is given by:

ATC (2-bit Multiplier) = (1 / 12) * (1 / 12) * ATC (16-bit MA)

= (1 / 144) * 583 = 4

5. Square ROM: ROM implemented as (a+b)**2, where a is the MSB and b is the LSB. The squaring requires two half-sized ROM and one half-sized multiplier in the first stage, and a full sized adder to add a**2 and 2*a*b, and a double sized adder to add this partial sum to b**2. Adding toggle for all those components will give toggles for Square ROM. For 2-bit square ROM ATC is given by:

ATC (2-bit Square ROM) = 2 * ATC (1-bit Square ROM) + ATC (1-bit Multiplier) + ATC (2-bit Adder) + ATC (4-bit Adder)

= 2 * 0 + 0.5 + 12 + 24 = 39.0

As we can see that for 2-bit Square ROM ATC of 39 is very high. If we implement 2-bit squaring, the gate count is 5 and the maximum and minimum number of transitions is 3 and 0. Therefore, the ATC of 2-bit Squarer implemented as combinational logic is the average of maximum and minimum number of transitions and is given by:

ATC (2-bit $\text{Squarer}_{\text{combinational}}$) = (0 + 3) / 2 = 1.5

But for larger bit-width Square ROM the implementation as combinational logic will be much larger and will have more toggles. In that case, we can use

the former method of implementation using Square ROM and calculating its ATC.

6. Xor: It can be implemented with 5 nand gates. The maximum and minimum number of toggles in xor gate is 5 and 2 respectively the average of which will give ATC. Therfore, ATC of xor gate is given by:

   ATC (Xor) = (5 + 2) / 2 = 3.5

Table 7.9 shows the ATC of sub-components of 4-bit HSM. It also shows the multiplicity of these sub-components in a 4-bit HSM. The total ATC of all the sub-components gives the ATC of 4-bit HSM.

Table 7.9 shows the calculation of toggle count of HSM.

Sub-Component	Multiplicity	ATC / instance	Sub-total
4-bit Inverter	2	4	8
8-bit Inverter	1	8	8
4-bit Adder	3	23	69
Xor gate	1	3.5	3.5
4-bit 2-to-1 Multiplexer	2	2	4
2-bit Square ROM	4	1.5	6
2-bit Multiplier / Product ROM	2	4	8
5-bit Adder	2	30	60
8-bit Adder	2	45	90
4-bit HSM			257

Table 7.9 Calculation of average toggle count of 4-bit HSM

Using the same method, ATC of HSM with higher bit-width is calculated. Instead of using square ROMs or HSMs to carry out multiplication to produce partial product, we can use any optimized multiplier instead. For example in 16-bit HSM we can use two 8-bit Wallace multipliers instead of square ROMs.

Table 7.10 shows the ATC of HSM of various bit-widths and having optimized Wallace multiplier as a core. Figure 7.2 shows the comparison of toggle count of HSM with 8-bit Wallace multiplier core with two other standard multipliers, Array and Wallace

multipliers [CaSh97]. As we can see in figure 7.2, HSM consumes about half the power as compared to Wallace multipliers and $1/9^{th}$ as compared to Array multiplier.

Bit-Width	ATC
16-bit	2831
32-bit	11499

Table 7.10 Average toggle count of HSMs with 8-bit Wallace multiplier core

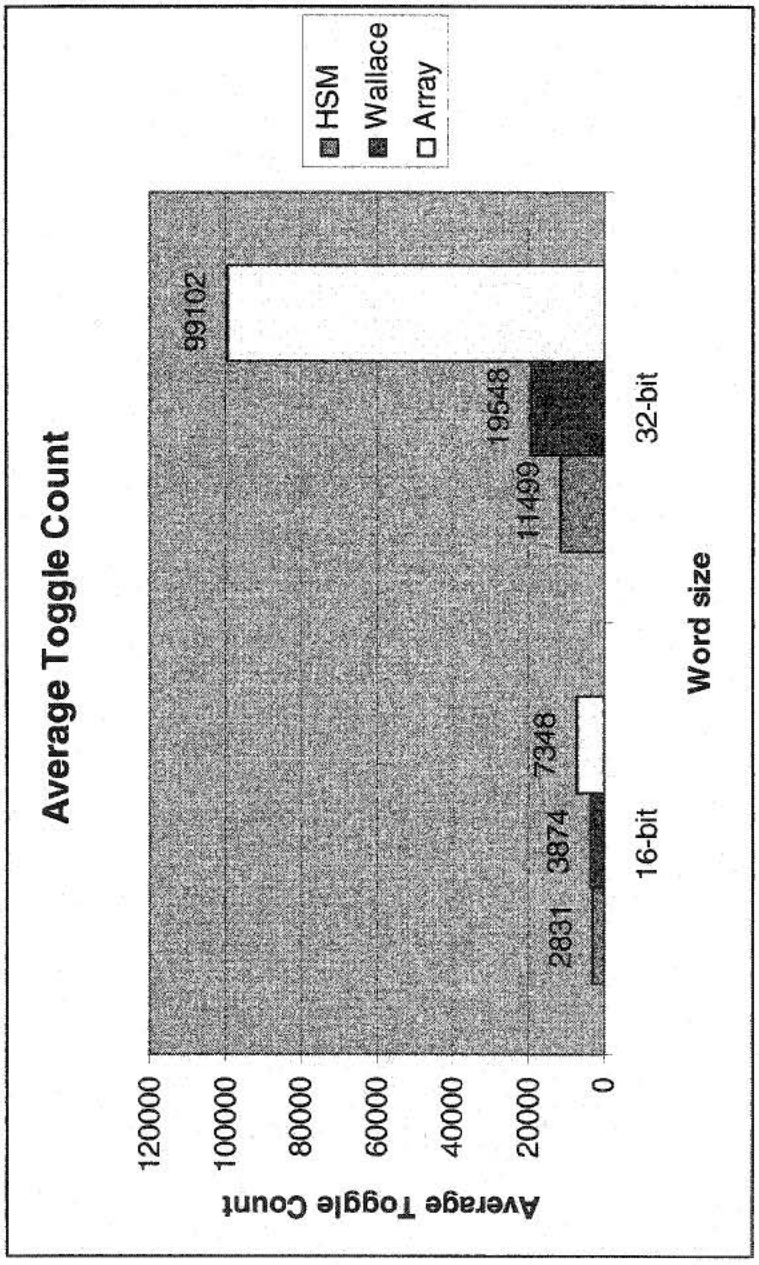

Figure 7.2 Comparison of toggle counts of HSM, Wallace Multiplier and Array Multiplier

7.3 Delay

The steps to calculate the delay of HSM are as follows:

1. Calculate delay of each individual sub-component
2. Calculate largest delay path

For the purpose of explaining the method, delay of 4-bit HSM is estimated. In calculating the delay of each individual sub-component, unit gate delay is assumed for every gate. The delays of sub-components of 4-bit HSM are tabulated in Table 7.11.

Sub-Component	Delay
4-bit Inverter	1
8-bit Inverter	1
4-bit Adder	12
4-bit 2-to-1 Multiplexer	3
Xor	3
2-bit Squarer	2
2-bit Multiplier	4
5-bit Adder	15
8-bit Adder	32

Table 7.11 Delays of sub-components of 4-bit HSM

Figure 7.3 and 7.4 shows the block diagram of 4-bit HSM and 8-bit accumulator respectively. Analysis of the delay every path was done. The delay of the longest path is as shown below:

$$D_{longest} = D(P5\ S1) + D(P5\ S23) + D(P5\ S3) + D(P5\ S4) + D(P5\ S5)$$

$$= 12 + 4 + 15 + 24 + 24 = 79$$

Figure 7.5 shows the comparison of delay of HSM which has 8-bit Wallace multiplier as the core, with Wallace Tree and Array multipliers [CaSh97]. The delay of HSM is intermediate between that of Wallace Tree and Array multipliers where Array multiplier is the slowest and Wallace Tree is the fastest among all three. Note that Carry look ahead adders were assumed in the calculation of the delay of the HSM.

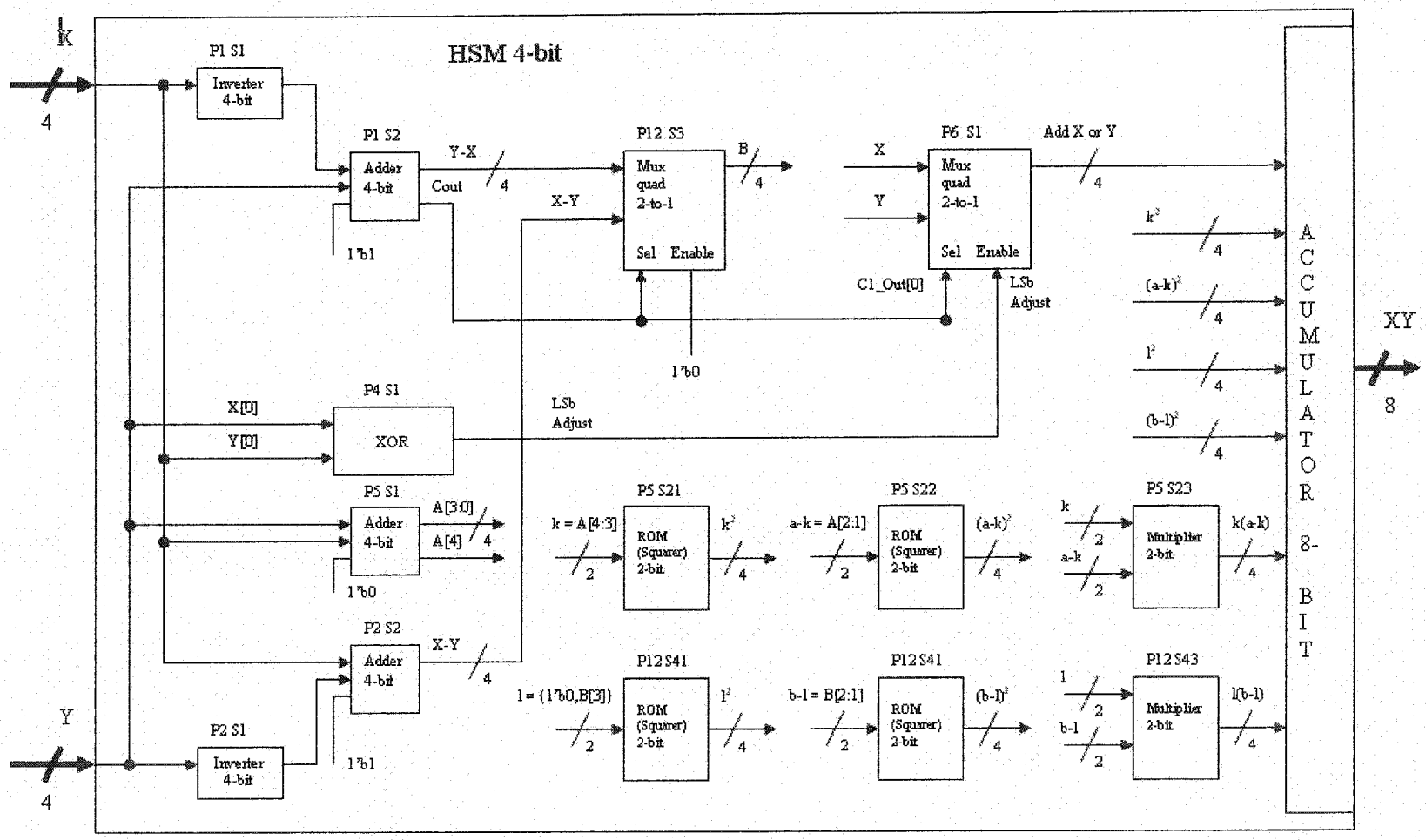

Figure 7.3 Block diagram of 4-bit HSM

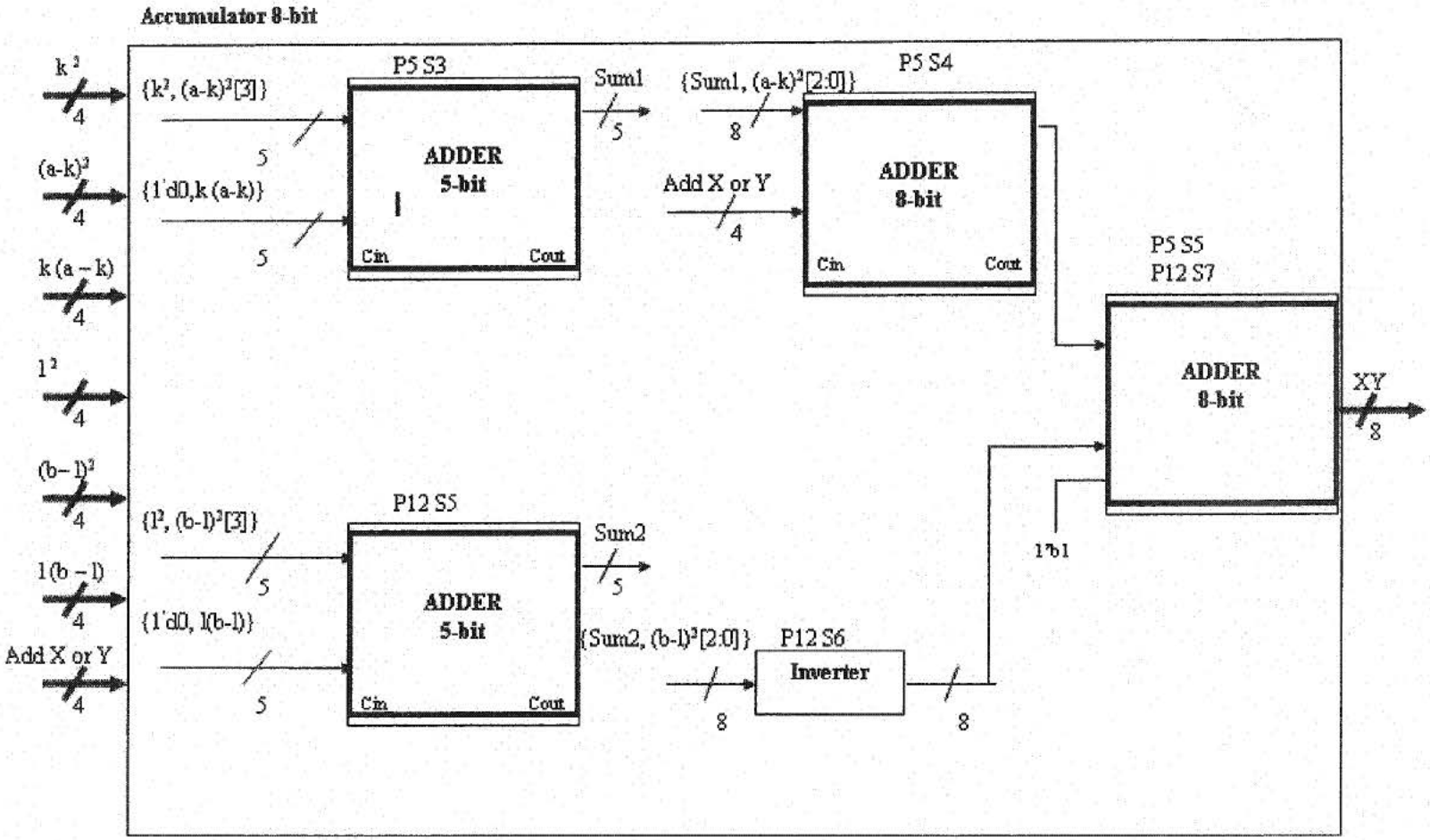

Figure 7.4 Block diagram of 8-bit Accumulator

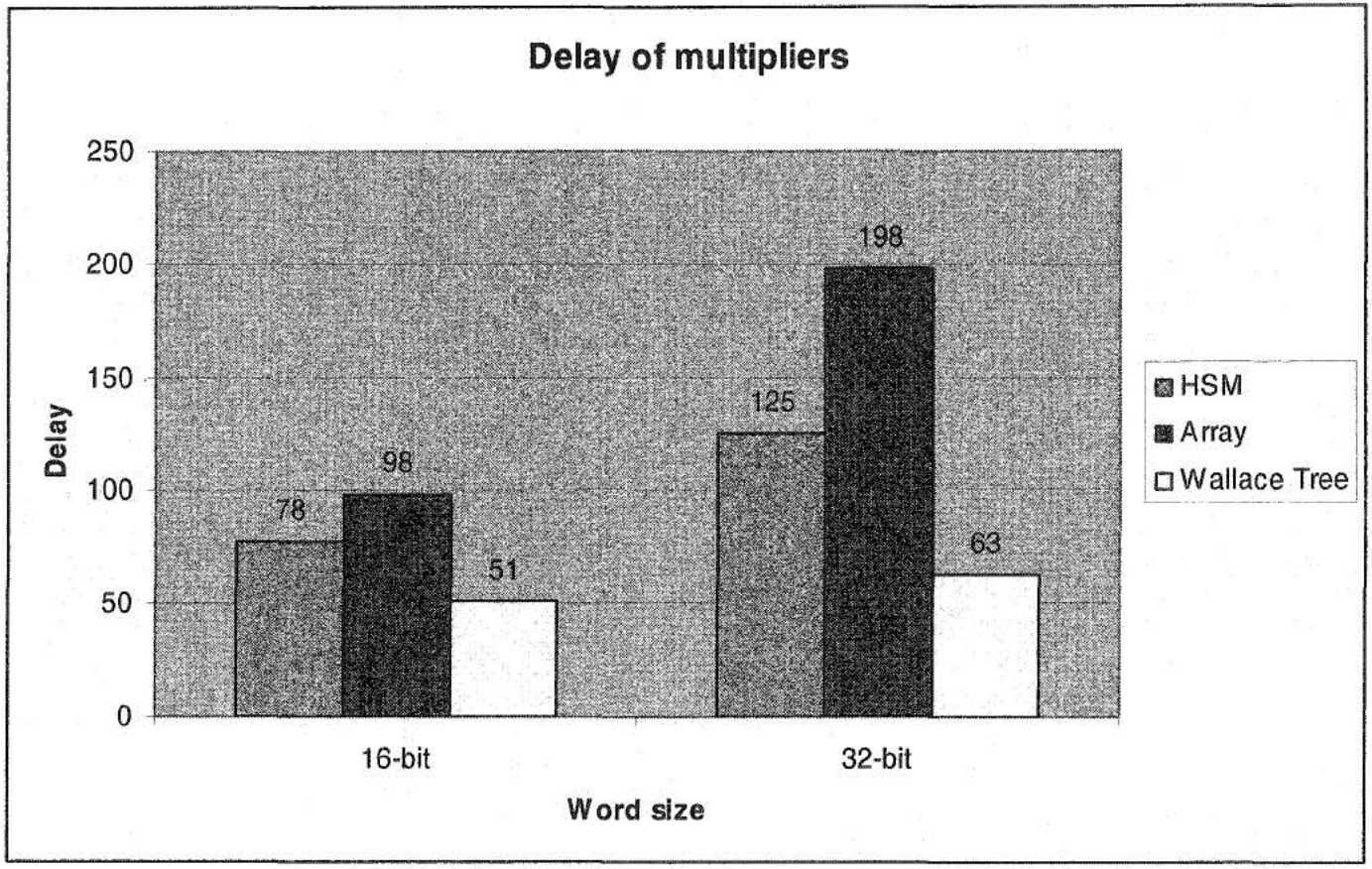

Figure 7.5 Delay of HSM, Wallace Tree and Array multipliers

CHAPTER 8

CONCLUSIONS

8.1 Discussion

16-bit and 32-bit HSM have significantly low power dissipation relative to the corresponding Wallace and Array Multiplier implementations. This multiplier is highly scalable because of its capability to compute on operands of variable precision which is attributed to the folding capability. Scalability of this multiplier is a great advantage if used in mobile devices because the precision of the multiplication can be controlled at runtime to optimize the limited resources and to do trade offs between performance and power.

The delay of this multiplier is intermediate to that of Array and Wallace Tree multipliers. The delay can be reduced further by using carry lookup adders rather than ripple carry adder.

The gate Count, however, is twice the other implementations. This can be reduced substantially, but our emphasis was on re-usable regular

localized structures. This overhead of object oriented methodology can be addressed.

HSM has a very regular structure and therefore we are expecting it to give a very dense implementation on IC.

8.2 Future Work

The following future work needs to be done on HSM:

1. Synthesize the Verilog code.
2. Estimate the toggle count by performing extensive simulations
3. Perform floorplanning to estimate the approximate silicon area needed for the implementation
4. Calculate the delay of the architecture
5. Fine tune the design by using logic optimizers to reduce the gate count and the delay

REFERENCES

[Cava84] Cavanagh, J., Digital Computer Arithmatic, Design and Implementation, McGraw Hill Publisher, New York, NY, 1984.

[CaSh97] Callaway, T. K., and Shwartzlander, E. E., "Low Power Arothmetic Components " in Rabaey, J. M., and Pedram, M., Low Power Design Methodologies, Kluwer Academic Publishers, Boston, MA, 1997.

[ElAb97] Elrabaa, M. S., Abu-khater, I. S., and Elmasry, M. I., Advanced Low-Power Digital Circuit Techniques, Kluwer Academic Publishers, Norwell, MA, 1997.

[ITRS01] "International Technology Roadmap for Semiconductors", ITRS, 2001, http://www.itrs.net.

[ItCh03] Ito, M., Chinnery, D., and Keutzer, K., "Low Power Multiplication Algorithm for Switching Activity Reduction through Operand Decomposition", International Conference on Computer Design, San Jose, CA, 2003, pp. 21-27.

[KoKi03] Kolla, Y., Kim, Y., and Carter, J., "A Novel 32-bit Scalable Multiplier Architecture", Proceedings of the 13th ACM Great Lakes Symposium on VLSI, Washington, D. C., 2003, pp. 241-244.

[MeRu96] Meier, P. C. H., Rutenbar, R. A., and Carley, L. R., "Exploring Multiplier Architecture and Layout for Low Power", Custom Integrated Circuits Conference, 1996, pp. 513-516.

[Mano00] Mano, M. M., Digital Logic and Computer Design, Prentice Hall Publications, Englewood Cliffs, NJ, 2000.

[Paln96] Palnitkar, S., Verilog HDL, Prentice Hall Publications, Englewood Cliffs, NJ, 1996.

[SaTe00] Savas, E., Tenca, A. F., and Koc, C. K., "A Scalable and Unified Multiplier Architecture For Finite Fields GF(p) and GF(2^m)", Cryptographic Hardware and Embedded Systems - CHES 2000, Worcester, Massachusetts, 2000, pp. 277-292.

[Shan01] Shankar, R., "Highly Scalable Multiplier", patents filed by Florida Atlantic University, Boca Raton, FL, 2001 & 2003.

[TeKo99] Tenca, A. F., and Koc, C. K., "A Scalable Architecture for Montgomery Multiplication", CHES'99, Worcester, MA, 1999, pp. 94-108.

[Veen00] Veendrick, H., Deep-Submicron CMOS ICs, Kluwer Academic Publishers, Norwell, MA, 2000.

APPENDIX A

A.1 4-bit HSM

8-bit Accumulator

```
module Accumulator_8bit (Out, ak2, kak, k2, bl2, lbl, l2,
accuracy_adjustment);

      output      [7:0] Out;
      input [3:0] ak2;
      input [3:0] kak;
      input [3:0] k2;
      input [3:0] bl2;
      input [3:0] lbl;
      input [3:0] l2;
      input [3:0] accuracy_adjustment;

      wire  [7:0] term1, term2, term3, term3bar;


      Adder_8bit  Adder_8bit1(term1, , {k2, ak2}, {1'd0, kak, 3'd0} ,
      1'd0);
      Adder_8bit  Adder_8bit2(term2, , term1, {4'd0,
      accuracy_adjustment}, 1'd0);
      Adder_8bit  Adder_8bit3(term3, , {l2, bl2}, {1'd0, lbl, 3'd0},
      1'd0);
      Inverter_8bit     Inverter_8bit1(term3bar, term3);
      Adder_8bit  Adder_8bit4(Out, , term2, term3bar, 1'b1);


endmodule
```

4-bit Adder

```
module Adder_4bit(Out, Cout, X, Y, Cin);


      output      [3:0] Out;
      input [3:0] X, Y;
      input       Cin;
      output      Cout;

      wire  [2:0] C;
```

```
      Full_Adder  FA0(Out[0], C[0], X[0], Y[0], Cin);
      Full_Adder  FA1(Out[1], C[1], X[1], Y[1], C[0]);
      Full_Adder  FA2(Out[2], C[2], X[2], Y[2], C[1]);
      Full_Adder  FA3(Out[3], Cout, X[3], Y[3], C[2]);

endmodule
```

8-bit Adder

```
module Adder_8bit(Out, Cout, X, Y, Cin);


      output      [7:0] Out;
      input       [7:0] X, Y;
      input       Cin;
      output      Cout;

      wire  [6:0] C;


      Full_Adder  FA0(Out[0], C[0], X[0], Y[0], Cin);
      Full_Adder  FA1(Out[1], C[1], X[1], Y[1], C[0]);
      Full_Adder  FA2(Out[2], C[2], X[2], Y[2], C[1]);
      Full_Adder  FA3(Out[3], C[3], X[3], Y[3], C[2]);
      Full_Adder  FA4(Out[4], C[4], X[4], Y[4], C[3]);
      Full_Adder  FA5(Out[5], C[5], X[5], Y[5], C[4]);
      Full_Adder  FA6(Out[6], C[6], X[6], Y[6], C[5]);
      Full_Adder  FA7(Out[7], Cout, X[7], Y[7], C[6]);

endmodule
```

Full Adder

```
module Full_Adder(S, C, x, y, z);

      output      S, C;
      input x,y,z;

      wire  w1, w2;

      xor xor1( w1, x, y );
      xor xor2( S, w1, z );

      assign      C = (z & (x ^ y)) | (x & y);


endmodule
```

4-bit HSM

```
module HSM_4bit ( Out, X, Y );
```

```
output      [7:0] Out;
input       [3:0] X;
input       [3:0] Y;

//Internal variables
wire  [4:0] ai;
wire  [4:0] bi;
wire  [3:0] k2;
wire  [3:0] ak2;
wire  [3:0] l2;
wire  [3:0] bl2;
wire  [3:0] kak;
wire  [3:0] lbl;
wire  [1:0] C1_Out;

`define     k     ai[4:3]
`define     ak    ai[2:1]
`define     l     bi[4:3]
`define     bl    bi[2:1]

wire  [3:0] error_adjustment;
wire  [3:0] xbar, ybar;
wire  x0bar, y0bar, error, a1, a2, a3, a4, o1, enable, select,
      n3, errorbar;
wire  [3:0] x_minus_y, y_minus_x;

wire        Cout;

//Module instantiations
Square_ROM_2bit         Square_ROM_2bit_1(k2,`k);
Square_ROM_2bit         Square_ROM_2bit_2(ak2, `ak);
Square_ROM_2bit         Square_ROM_2bit_3(l2,`l);
Square_ROM_2bit         Square_ROM_2bit_4(bl2, `bl);
Multiply_ROM_2bit       Multiply_ROM_2bit_1(kak, `k, `ak);
Multiply_ROM_2bit       Multiply_ROM_2bit_2(lbl, `l, `bl);
Accumulator_8bit        Accumulator_8bit1(Out, ak2, kak, k2, bl2,
                                  lbl, l2, error_adjustment);
Adder_4bit              Adder_4bit1(ai[3:0],ai[4],X[3:0],
                                  Y[3:0],1'b0); //ai = (X + Y)

Inverter_4bit           Inverter_4bit1(xbar, X);
Inverter_4bit           Inverter_4bit2(ybar, Y);
Adder_4bit              Adder_4bit2(x_minus_y, , X, ybar, 1'b1);
Adder_4bit              Adder_4bit3(y_minus_x, Cout, xbar, Y,
                                  1'b1);

// Check if error is there. X[0] xor Y[0]
not   Not1(x0bar, X[0]);
not   Not2(y0bar, Y[0]);
and   And1(a1, x0bar, Y[0]);
and   And2(a2, X[0], y0bar);
or    Or1(error, a1, a2);
not   Not3(errorbar, error);
Mux_quad_2_to_1 Mux_quad_1(error_adjustment, X, Y, Cout,
                           errorbar);
Mux_quad_2_to_1 Mux_quad_2(bi[3:0], y_minus_x, x_minus_y, Cout,
                           1'b0);
```

```
      assign      bi[4] = 1'b0;

endmodule
```

4-bit Inverter

```
module      Inverter_4bit(Out, In);


      output      [3:0] Out;
      input [3:0] In;

      parameter   gate_count = 4;

      assign      Out = ~In;

endmodule
```

8-bit Inverter

```
module      Inverter_8bit(Out, In);

      output      [7:0] Out;
      input       [7:0] In;

      assign      Out = ~In;

endmodule
```

2-bit Product ROM

```
module Multiply_ROM_2bit (Out, In1, In2);

      output      [3:0] Out;
      input [1:0] In1;
      input [1:0] In2;

      `define     w     In1[1]
      `define     x     In1[0]
      `define y   In2[1]
      `define z   In2[0]


      wire  x_bar, w_bar, y_bar, z_bar;

      not   N1(x_bar, `x);
      not   N2(y_bar, `y);
      not   N3(z_bar, `z);
      not   N4(w_bar, `w);


      assign      Out[0] = `x & `z;
```

```
      assign      Out[1] = (w_bar & `x & `y) | (`x & `y & z_bar) | ( `w
                           & y_bar & `z) | ( `w & x_bar & `z);
      assign      Out[2] = (`w & x_bar & `y) | (`w & `y & z_bar);
      assign      Out[3] = `w & `x & `y & `z;

endmodule
```

4-bit 2-to-1 Multiplexer

```
module Mux_quad_2_to_1(Out, In1, In2, Select, Enable);

      output            Out;
      reg         [3:0] Out;
      input       [3:0] In1, In2;
      input             Select, Enable;

      initial
            Out = 0;

      always @(In1 or In2 or Select or Enable)
      begin
            if(Enable==1)
                  Out = 0;
            else
            begin
                  if(Select == 1)
                        Out = In1;
                  else
                        Out = In2;
            end
      end

endmodule
```

2-bit Square ROM

```
/*=====================================================
      2-bit Square ROM Model
      File name   : Square_ROM_2bit.vg
      Module name : Square_ROM_2bit
      Input       : In  (2-bit)
      Output            : Out (4-bit)
      Instances:
            None

      It is the Square ROM 2-bit wide that is it takes
      in 2-bit wide input (In) and outputs its square (Out)
      It is gate level implementation.
=====================================================*/


module Square_ROM_2bit( Out, In );
```

```
        output      [3:0]       Out;
        input       [1:0]       In;

        wire  In0_bar;

        not   Not1(In0_bar, In[0]);
        and   And1(Out[3], In[1], In[0]);
        and   And2(Out[2], In[1], In0_bar);
        assign Out[1] = 1'b0;
        assign Out[0] = In[0];

endmodule
```

Top

```
module Top;

        wire  [7:0] Out;
        reg   [3:0] X;
        reg   [3:0] Y;

        HSM_4bit HSM_4bit1( Out, X, Y );

        integer i, j;

        initial
        begin
              X = 0;
              Y = 0;

              for(i = 0; i < 16; i = i+1)
              begin
                    for(j = 0; j < 16; j = j+1)
                    begin
                          #10
                                Y = i;
                                X = j;
                    end
              end

              #50 $finish;
        end

        initial
              $monitor("X = %d, Y = %d, Out = %d", X, Y, Out);
endmodule
```

A.2 8-bit HSM

16-bit Accumulator

```
module Accumulator_16bit (Out, ak2, kak, k2, b12, lb1, l2,
                          error_adjustment);


      output      [15:0]      Out;
      input       [7:0]       ak2;
      input       [7:0]       kak;
      input       [7:0]       k2;
      input       [7:0]       b12;
      input       [7:0]       lb1;
      input       [7:0]       l2;
      input       [7:0]       error_adjustment;

      wire  [15:0]      term1, term2, term3, term3bar;

      Adder_16bit Adder_16bit1(term1, , {k2, ak2}, {3'd0, kak, 5'd0} ,
                               1'd0);
      Adder_16bit Adder_16bit2(term2, , term1, {8'd0,error_adjustment},
                               1'd0);
      Adder_16bit Adder_16bit3(term3, , {l2, b12}, {3'd0, lb1, 5'd0},
                               1'd0);
      Adder_16bit Adder_16bit4(Out, , term2, term3bar, 1'b1);
      Inverter_16bit    Inverter_16bit1(term3bar, term3);

endmodule
```

16-bit Adder

```
module Adder_16bit(Out, Cout, X, Y, Cin);

      output [15:0]     Out;
      output            Cout;

      input [15:0]      X;
      input [15:0]      Y;
      input             Cin;


      wire  [14:0]      C;


      Full_Adder  FA0(Out[0], C[0], X[0], Y[0], Cin);
      Full_Adder  FA1(Out[1], C[1], X[1], Y[1], C[0]);
      Full_Adder  FA2(Out[2], C[2], X[2], Y[2], C[1]);
      Full_Adder  FA3(Out[3], C[3], X[3], Y[3], C[2]);
      Full_Adder  FA4(Out[4], C[4], X[4], Y[4], C[3]);
      Full_Adder  FA5(Out[5], C[5], X[5], Y[5], C[4]);
      Full_Adder  FA6(Out[6], C[6], X[6], Y[6], C[5]);
      Full_Adder  FA7(Out[7], C[7], X[7], Y[7], C[6]);
      Full_Adder  FA8(Out[8], C[8], X[8], Y[8], C[7]);
      Full_Adder  FA9(Out[9], C[9], X[9], Y[9], C[8]);
      Full_Adder  FA10(Out[10], C[10], X[10], Y[10], C[9]);
      Full_Adder  FA11(Out[11], C[11], X[11], Y[11], C[10]);
      Full_Adder  FA12(Out[12], C[12], X[12], Y[12], C[11]);
```

```
      Full_Adder  FA13(Out[13], C[13], X[13], Y[13], C[12]);
      Full_Adder  FA14(Out[14], C[14], X[14], Y[14], C[13]);
      Full_Adder  FA15(Out[15], Cout , X[15], Y[15], C[14]);

Endmodule
```

8-bit HSM

```
module HSM_8bit ( Out, X, Y );

      output      [15:0]      Out;

      input [7:0] X;
      input [7:0] Y;

      //Internal variables
      wire  [8:0] ai;
      wire  [8:0] bi;
      wire  [7:0] k2;
      wire  [7:0] ak2;
      wire  [7:0] l2;
      wire  [7:0] bl2;
      wire  [7:0] kak;
      wire  [7:0] lbl;
      wire  [1:0] C1_Out;

      `define     k     ai[8:5]
      `define     ak    ai[4:1]
      `define     l     bi[8:5]
      `define     bl    bi[4:1]

      wire  [7:0] error_adjustment;
      wire  [7:0] xbar, ybar;
      wire  x0bar, y0bar, error, a1, a2, a3, a4, o1, enable, select,
            n3, errorbar, Cout;
      wire  [7:0] x_minus_y, y_minus_x;

      parameter   gate_count  = 6;

      //Module instantiation
      Square_ROM_4bit         Square_ROM_4bit_1(k2,`k);
      Square_ROM_4bit         Square_ROM_4bit_2(ak2, `ak);
      Square_ROM_4bit         Square_ROM_4bit_3(l2,`l);
      Square_ROM_4bit         Square_ROM_4bit_4(bl2, `bl);
      HSM_4bit                HSM_4bit1(kak, `k, `ak);
      HSM_4bit                HSM_4bit2(lbl, `l, `bl);



      Accumulator_16bit       Accumulator_16bit1(Out, ak2, kak, k2,
                                                 bl2, lbl, l2,
                                                 error_adjustment);
      Adder_8bit              Adder_8bit1(ai[7:0],ai[8],X[7:0],Y[7:0],
                                    1'b0); //=== ai = (X + Y)
```

```
        Inverter_8bit           Inverter_8bit1(xbar, X);
        Inverter_8bit           Inverter_8bit2(ybar, Y);
        Adder_8bit              Adder_8bit2(x_minus_y, , X, ybar, 1'b1);
        Adder_8bit              Adder_8bit3(y_minus_x,Cout , xbar, Y,
                                        1'b1);

        // Accuracy Adjustment
        not    Not1(x0bar, X[0]);
        not    Not2(y0bar, Y[0]);
        and    And1(a1, x0bar, Y[0]);
        and    And2(a2, X[0], y0bar);
        or     Or1(error, a1, a2);
        not    Not3(errorbar, error);
        Mux_octa_2_to_1 Mux_octa_1(error_adjustment, X, Y, Cout,
                                errorbar);
        Mux_octa_2_to_1 Mux_octa_2(bi[7:0], y_minus_x, x_minus_y, Cout,
                                1'b0);

        assign      bi[8] = 1'b0;


endmodule
```

16-bit Inverter

```
module      Inverter_16bit(Out, In);

        output      [15:0]      Out;
        input       [15:0]      In;

        assign      Out = ~In;

endmodule
```

8-bit 2-to-1 Multiplexer

```
module Mux_octa_2_to_1(Out, In1, In2, Select, Enable);

        output      Out;
        reg   [7:0] Out;
        input [7:0] In1, In2;
        input       Select, Enable;

        parameter   gate_count  = 27;

        always @(In1 or In2 or Select or Enable)
        begin
              if(Enable==1)
                    Out = 0;
              else
              begin
                    if(Select == 1)
                          Out = In1;
                    else
```

```
                        Out = In2;
            end
      end

endmodule
```

Top

```
module Top;

      wire  [15:0]      Out;
      reg   [7:0] X;
      reg   [7:0] Y;

      HSM_8bit HSM_8bit1( Out, X, Y );

      integer i, j, expected_output;

      initial
      begin
            X = 0;
            Y = 0;

            for(i = 0; i < 256; i = i+1)
            begin
                  for(j = 0; j < 256; j = j+1)
                  begin
                        #10
                              expected_output <= i * j;
                              Y <= i;
                              X <= j;

                              if(Out != expected_output)
                              begin
                                    $display("X = %d , Y = %d , Desired
                                    Output = %d , HSM Output = %d", X,
                                          Y, expected_output, Out);

                                    $display("Incorrect Output\n");
                              end
                  end
            end

            #50 $finish;
      end


endmodule
```

32-bit Behavioral Code of HSM Algorithm

```
module HSM_32 (Product, X, Y);

      // Port Declaration
      output Product;          // Product of X and Y
      input  X, Y;             // Input Values

      reg   [63:0]      Product;
      wire  [31:0]      X, Y;

      // Declaration of internal variables
      reg   [31:0]      a, b;
      reg   [15:0]      k, l;
      reg   [15:0]      a_k, b_l;
      reg   [31:0]      k_square, l_square;
      reg   [31:0]      a_k_square, b_l_square;
      reg   [31:0]      k2_times_a_k, l2_times_b_l;

      // Initialization of internal registers
      initial
      begin
            Product = 0;
            a = 0; b = 0;
            k = 0; l = 0;
            a_k = 0; b_l = 0;
            k_square = 0; l_square = 0;
            a_k_square = 0; b_l_square = 0;
            k2_times_a_k = 0; l2_times_b_l = 0;
      end

      // Execute the block if any of the input X or Y changes
      always @(X or Y)
      begin

            a = (X + Y)/2;     //     a = average of sum of X and Y

            // Find the absolute difference between X and Y
            if( X >= Y)
                  b = (X + (~Y) + 1)>>1;
            else
                  b = ((~X) + Y + 1) >> 1;



            // Folding a and b at mid-point.
            // k = lower half of a and a - k = upper half of a
            // l = lower half of b and b - l = upper half of b
            k = a[15:0];
            l = b[15:0];
            a_k = a[31:16];
            b_l = b[31:16];


            // Calculate the six terms as in equation 3.6
            k_square = k * k;
```

```
            l_square = l * l;
            a_k_square = a_k * a_k;
            b_l_square = b_l * b_l;
            k2_times_a_k = 2 * k * a_k;
            l2_times_b_l = 2 * l * b_l;


            Product = (k_square + {a_k_square, 16'b0} + {k2_times_a_k,
                       8'b0}) - (l_square  + {b_l_square, 16'b0} +
                       {l2_times_b_l, 8'b0});

            if ( X[0] ^ Y[0] == 1 )
            begin
                  if ( X >= Y)
                        Product = Product + Y;
                  else
                        Product = Product + X;
            end

      end

endmodule
```