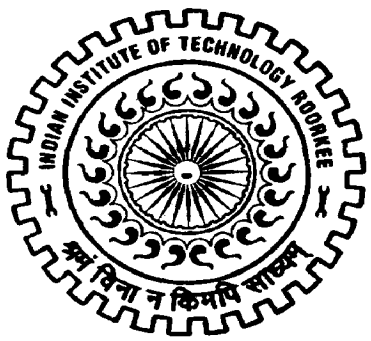
RUNTIME SCALABLE POWER-AWARE BOOTH MULTIPLIER USING 2-DIMENSIONAL PIPELINE GATING

A DISSERTATION

Submitted in partial fulfillment of the
requirements for the award of the degree
of
MASTER OF TECHNOLOGY
in
ELECTRONICS AND COMMUNICATION ENGINEERING
(With Specialization in Semiconductor Devices and VLSI Technology)

By

SRINIVAS BANDARI



DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE - 247 667 (INDIA)

JUNE, 2007

CANDIDATE'S DECLARATION

I hereby declare that the work, which is presented in this dissertation report, entitled **"RUNTIME SCALABLE POWER-AWARE BOOTH MULTIPLIER USING 2-DIMENSIONAL PIPELINE GATING"**, being submitted in partial fulfillment of the requirements for the award of the degree of **Master of Technology** in **Electronics and Communication Engineering** with specialization in **Semiconductor Devices & VLSI Technology**, in the Department of Electronics and Computer Engineering, Indian Institute of Technology, Roorkee is an authentic record of my own work carried out from July 2006 to June 2007, under guidance and supervision of Dr.**S.Dasgupta**, Assistant Professor, Department of Electronics and Computer Engineering, Indian Institute of Technology, Roorkee.

The results embodied in this dissertation have not submitted for the award of any other Degree or Diploma.

Date : 28th June, 2007

Place: Roorkee

B. Srinivas 28/6/'07

SRINIVAS BANDARI

CERTIFICATE

This is to certify that the statement made by the candidate is correct to the best of my knowledge and belief.

Date: 28th June, 2007

Place: Roorkee

Dr. S.Dasgupta,
Assistant Professor, E&CE Department,
Indian Institute of Technology Roorkee,
Roorkee – 247 667, (INDIA)

ACKNOWLEDGEMENT

I wish to express my deep sense of gratitude and sincere thanks to **Dr.S.Dasgupta,** Assistant Professor, Department of Electronics and Computer Engineering, IIT Roorkee for his valuable guidance. This work is the reflection of his thoughts, ideas, and concepts and above all, his efforts. I am highly indebted to him for his kind and valuable suggestions and his valuable time during the period of the work. The huge quantum of knowledge I had gained during his inspiring guidance would be immensely beneficial for my future endeavors. Apart from that I am also very grateful to **Prof.A.K.Saxena** for his valuable guidance during my dissertation work.

Also, I feel indebted to all those endless researchers all over the world whose work I have used in my project. Their sincerity and devotion motivates me the most.

I am also thankful to all my friends for their continuous support and enthusiastic help.

Date: 26/06/'07

Place: Roorkee **SRINIVAS BANDARI**

ABSTRACT

Power-awareness indicates the scalability of the system energy with changing conditions and quality requirements. Energy-efficient power-aware design is highly desirable for DSP functions that encounter a wide diversity of operating scenarios in battery-powered wireless sensor network systems. The DSP functions extensively make use of the multiply-and-accumulate (MAC) operation, which makes the multiplication function as most power-consuming task. Therefore it is essential to implement the power-efficient multipliers for power-aware DSPs.

Addressing power-awareness, a novel reconfigurable pipelined Booth multiplier using 2-dimensional pipeline gating scheme is proposed . This technique is to gate the clock to registers in both vertical direction (data flow direction in pipeline) and horizontal direction (within each pipeline stage). For signed multipliers using 2's complement representation, sign extension, which wastes power and causes longer delay, could be avoided by implementing this technique. Our multiplier based on the gated input signals implements a 16-bit, 8bit or 4-bit multiplication operation.

The proposed reconfigurable pipelined Booth multiplier was first modeled in VHDL and functionally verified using Mentor Graphics ModelSim simulator. After functional validation, the architecture was synthesized for appropriate time and area constraints using Synopsys Design Compiler. TSMC 90nm CMOS technology and standard cell library were used. The power analysis of the gate level structure was done using Synopsys VCS and PrimePower tool.

For the 8-bit and 4-bit computations, the proposed Booth multiplier leads to a 61% and 87% power consumption reduction over a non-scalable Booth multiplier, respectively. The proposed scalable pipelined Booth multiplier proves to be globally 48% more power efficient than a non-scalable pipelined Booth multiplier, and also it has fast speed due to pipelining.

CONTENTS

Chapter 1
INTRODUCTION

With the increasing popularity of battery-powered portable applications and the dramatic decrease in feature size, demanding the chips that consume smallest amount of power. Even in the case of non-portable applications the amount of power consumed has become so high that they require expensive packaging and heat sinks. Thus, power has become one of the major design constraints along with area and timing.

Many low power techniques have been developed to match different circuits and conditions [1]-[2]. Bhardwaj et al., [3] introduced a new measurement, power-awareness, to indicate the ability of the system power to scale with changing conditions and quality requirements. Scalability is an important figure-of-merit since it allows the end user to implement operational policy [3], just like the user of mobile multimedia equipment needs to select between better quality and longer battery operation time. The examples include that a well-designed system must gracefully degrade its quality and performance as the available energy resources are depleted [4]. In such systems like digital camera, users are allowed to select certain parameters like resolution. After user selects a resolution, there will be a short period of time to allow the system to set up. During this period, the CPU will configure itself and set up the control to the whole system. Such parameters will not change frequently. After each change, the new value will remain stable for sometime. So for a power aware system in these applications, on-the-fly control is not needed.

The power dissipation in CMOS circuit has three components: switching power, short-circuit power, and leakage power. Among these components, switching power is the dominant figure. When a node in circuit is switching, the load capacitance on this node will dissipate power due to the charging/discharging operation. If the switching activity could be reduced, the total power dissipation will be saved. For Boolean non-pipelined multipliers, starting from reset-to-zero state, low input precision calculation (like 0001×0001) dissipates much less power than high input precision calculation (like

1111×1111) because there are much less switching activities in internal nodes. Here the input precision is defined as the number of useful input bits (without padded 0's in high order bits) during the calculation. For example, the input precision of 0101 is 3, while the input precision of 1000 is 4. So, Boolean non-pipelined multipliers are said to have natural power awareness to the changing of input precisions.

1.1 Previous Work

Several techniques have been developed to reduce the power dissipation in multipliers. Huang et al., [5] introduced a 2-dimensional signal gating method for low power array multiplier design. This approach provides gating lines for both multiplicand and multiplier operands. By deactivated different regions in the multiplier, power dissipation could be reduced. This approach is for non-pipelined array multiplier and cannot be extended to pipelined design because it cannot reduce the switching activities in registers. Bhardwaj et al., [3] introduced a selective method to design power-aware multiplier. This method is also for non-pipelined designs and brings high area cost. Meier et al., [6] introduced a polarity-inversion technique for the adders in signed multiplier. This technique does not solve the sign extension problem so that the multiplicands in lower precision still cannot be processed directly. Lee et al, [7] introduced a reduced architecture based on the redundancy of lower order bits in some DSP applications. This technique is not for general use and does not solve the sign extension problem in signed multiplier.

Kim et al., [8] introduced a clock gating method to design reconfigurable multiplier. This method is to selectively disable pipeline stages by gating clocks and to select correct results by multiplexers. Very little additional area cost is needed (only several AND2 gates and multiplexers) to implement this technique. Good power and latency saving can be achieved due to the reduced switching activities of registers in corresponding pipeline stages. The outputs of the multiplier are selected from different stages to ensure the correctness and obtain latency reduction. This technique can be seen as 1-dimensional pipeline gating because it only considers gating clocks to unnecessary stages along data

flow direction. As the computational width of multiplier growing from 4-bit, 8-bit, to 32-bit and 64-bit, 1-dimensional pipeline gating is far from enough.

Jia Di, [9]-[10] has proposed a 2-dimensional clock gating, that is, gate the clock to the registers in both vertical direction (data flow direction in pipeline) and horizontal direction (within each pipeline stage) and applied it to an array multiplier and used the multiplier for FIR application.

H. Lee, [11] has applied 2-dimensional signal gating to the booth multiplier, but is area inefficient as the partial product reduction tree is not sharable. Also it consumes more power in case of 16-bit multiplication.

1.2 Statement of the problem

Deeply pipelined multipliers are used in systems that need either high throughput or accurate timing control, like retimed FIR filters [12]. In pipelined multipliers, each pipeline stage contains a number of registers. Clock is connected to each register. In each clock cycle, a transition will occur on the clock input node of each register. This transition is independent of input data and will cause power dissipation even when the current input data of the register is the same as the current data output. Since in deeply pipelined designs, the number of registers is much larger than that of other elements, these designs do not have the natural power awareness to the changing of input precision due to the large portion of power dissipated on clock input nodes.

To solve these problems and improve the power awareness of deeply pipelined multipliers, a technique, 2-dimensional pipeline gating, is applied to the Booth multiplier. This technique is to gate the clock to the registers in both vertical direction (data flow direction in pipeline) and horizontal direction (within each pipeline stage). The additional area cost to implement this technique to design array multipliers is small.

1.3 Organization of the thesis

The thesis is divided into six chapters.

The first chapter is specifies the problem and previous work carried out in the low power multiplier area.
In the second chapter we deal with the basics of multiplication and various types of multipliers.

In the third chapter we explain various sources of power consumption in CMOS circuits and give some methods of reducing power.

The fourth chapter elaborates the architectures of various multipliers designed.

In the fifth chapter we present the analysis procedure, output waveforms after synthesis and the analysis of the results.

In the last chapter the thesis is concluded.

2.2 Partial product generation

Partial product generation is the very first step in binary multiplier. These are the intermediate terms which are generated based on the value of multiplier. If the multiplier bit is '0', then partial product row is also zero, and if it is '1', then the multiplicand is copied as it is. From the 2nd bit multiplication onwards, each partial product row is shifted one unit to the left as shown in the above mentioned example. In signed multiplication, the sign bit is also extended to the left. Partial product generators for a conventional multiplier consist of a series of logic AND gates as shown in Figure 3. Careful optimization of the partial-product generation can lead to some substantial delay and area reduction.

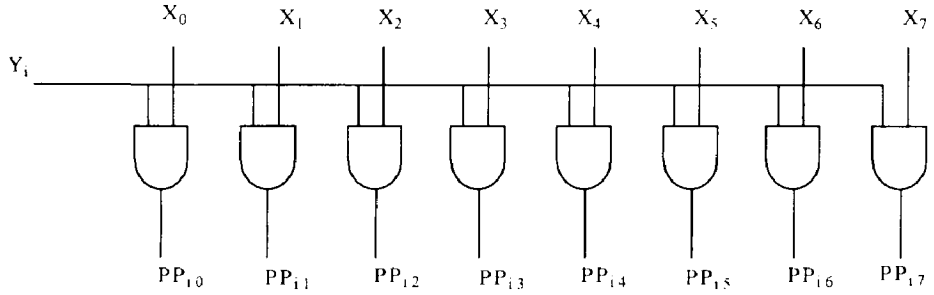


Figure 2.2 Partial product generation logic

2.2.1 Booth Encoding

Booth encoding is a method used for the reduction of the number of partial products proposed by A.D. Booth in 1951 [13]. A binary number X consisting of m bits represented in 2's complement format can be described as

$$X=-2^{m}X_{m} + 2^{m-1} X_{m-1} + 2^{m-2}X_{m-2} + \quad ..... \qquad (2.1)$$

Rewriting Eq.2.1 using $2^{a} = 2^{a+1} - 2^{a}$ leads to

$$X=-2^{m}(X_{m-1}-X_{m}) + 2^{m-1}(X_{m-2}-X_{m-1}) + 2^{m-2}(X_{m-3}-X_{m-2}) + \quad ..... \qquad (2.2)$$

Considering the first 3 bits of X, we can determine whether to add Y, 2Y or 0 to partial product.

Chapter 2
MULTIPLIERS

2.1. Multiplier Background

2.1.1. Basic binary multiplier

The operation of multiplication is rather simple in digital electronics. It has its origin from the classical algorithm for the product of two binary numbers. This algorithm uses addition and shift left operations to calculate the product of two numbers. Two examples are presented below.

10 X 8 =80	-6 X 4 =-24
1010	1010
X1000	X 0100
0000	0000
0000	0000
0000	111010
1010	00000
1010000	11101000

Figure 2.1 Basic binary multiplication

The left example shows the multiplication procedure of two unsigned binary digits while the one on the right is for signed multiplication. The first digit is called Multiplicand and the second Multiplier. The only difference between signed and unsigned multiplication is that we have to extend the sign bit in the case of signed one, as depicted in the given right example in Partial product row 3. Based upon the above procedure, we can deduce that any multiplication had three basic steps.

1) Partial product generation.

2) Partial product accumulation.

3) Final addition.

2.2.2 Modified Booth Encoding (MBE)

Modified booth encoding was invented by O.L. Macsorley in 1961[14]. MBE is an enhanced form of Booth encoding. A binary number $X = x_{m-1}, x_{m-2},\ldots, x_0$ consisting of m bits represented in 2's complement form can be mathematically expressed as

$$X=-2^m x_{m-1}+\Sigma x_i 2^i \quad , 0<i<m-2 \tag{2.3}$$

Equivalently, representation of X in base 4 is as follows:

$$X=d_i 4^i \quad , 0<i<m/2-1 \tag{2.4}$$

The digits d_i are chosen from the ensemble {-2,-1, 0, 1, 2} according to Table 2.1.

X_{2i+1}	X_{2i}	X_{2i-1}	Increment
0	0	0	0
0	0	1	Y
0	1	0	Y
0	1	1	2Y
1	0	0	-2Y
1	0	1	-Y
1	1	0	-Y
1	1	1	0

Table 2.1 Modified Booth encoding table [24]

For each step i, three bits of multiplier X i.e. x_{2i-1}, x_{2i}, x_{2i+1} are considered and the corresponding value of d_i is obtained from Table 2.1. There are few points to remember here[23].

1. Zero must always be concatenated to the right of X, i.e. x_{-1} is considered to be 0.
2. m must always be even.

There are two unavoidable consequences when utilizing MBE as *sign extension prevention* and *negative encoding*16]. The combination of these two results in the formation of one additional partial product row, which requires more hardware and the system, also becomes slower. The advantage of using MBE is that the number of partial

products are reduced to m/2. This, in turn, reduces the hardware burden and increases the speed of multiplier.

2.3 Carry Save Adder Tree (CSAT)

Carry Save Adder (CSA) can be used to reduce the number of addition cycles as well as to make each cycle faster. Carry save adder is also called a compressor. A full adder takes 3 inputs and produces 2 outputs i.e. sum and carry, hence it is called a 3:2 compressor. In CSA, the output carry is not passed to the neighboring cell but is saved and passed to the cell one position down. In order to add the partial products in correct order, Carry save adder tree (CSAT) is used. An example to understand the operation of CSAT is shown in Figure 2.3. Suppose we have 4 partial products, each consisting of 4 bits.

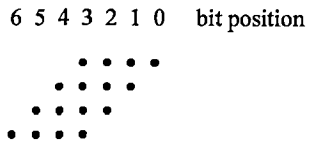


Figure 2.3 Bit positions in multiplier

The first step is to rearrange the partial products according to bit positions as shown in Figure 2.4.

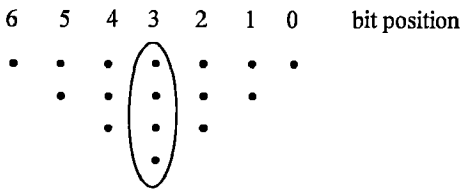


Figure 2.4 Rearranging bits in multiplier

The longest path consists of 4 terms at bit position 3. The final step is to determine the number of half and full adders required to complete the addition operation. A 9-input CSAT is shown in Figure 2.5.

If in the level j of the tree, the number of bits is n, then k= n/3 full-adders should be used for the summation. The k generated carry signals are sent to the level j+1 of the tree i+1.

Since the number of bits to sum has been reduced by three fold at each level, the depth of the Wallace tree is O (logN)[15], where N is the initial number of bits.

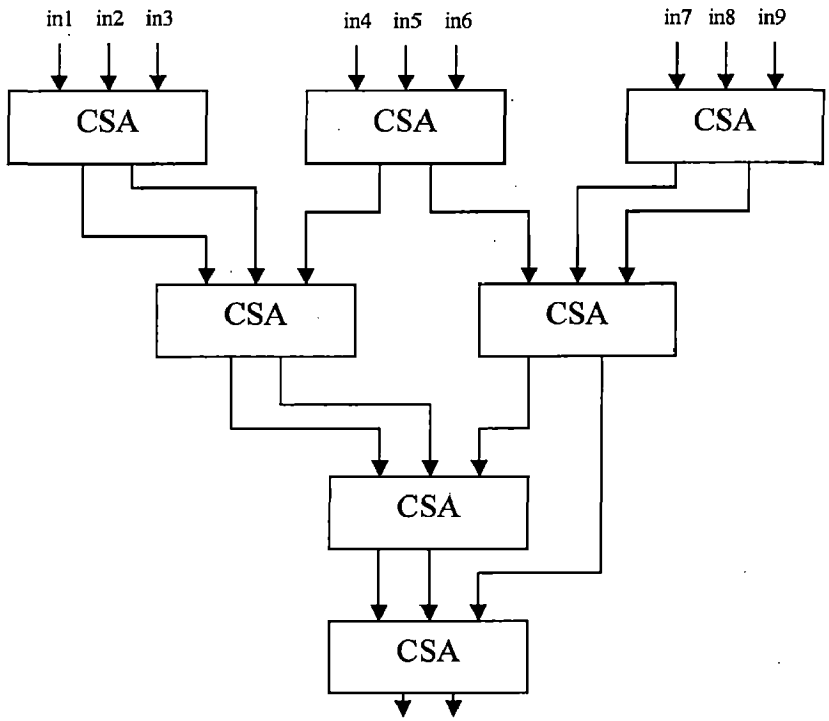


Figure 2.5 9-input reduction tree[22]

2.4. Fast Adders

The final step in completing the multiplication procedure is to add the final terms in the final adder. This is normally called "Vector-merging" adder. The choice of the final adder depends on the structure of the accumulation array[15]. Following is a list of fast adders which are normally used.

1. Carry look-ahead adder
2. Simple carry skip adder
3. Multilevel carry skip adder
4. Carry- select adder
5. Conditional sum adder
6. Hybrid adder
7. Ripple carry adder

2.5. Multiplier types

Multipliers are categorized relative to their applications, architecture and the way the partial products are produced and summed up. Based on all these, a designer might find following types of multipliers.

2.5.1. Sequential multipliers

The sequential multiplier is shown in the figure 2.6. The generations of the partial products require NXM two bit AND gates. Most of the area of the multiplier is devoted to the adding of the N partial products, which require (N-1) M-bit adders. The shifting of the partial products for their proper alignment is performed by simple routing and does not require any logic. The over all structure can easily be compacted into a rectangle, resulting in a very efficient layout[15].

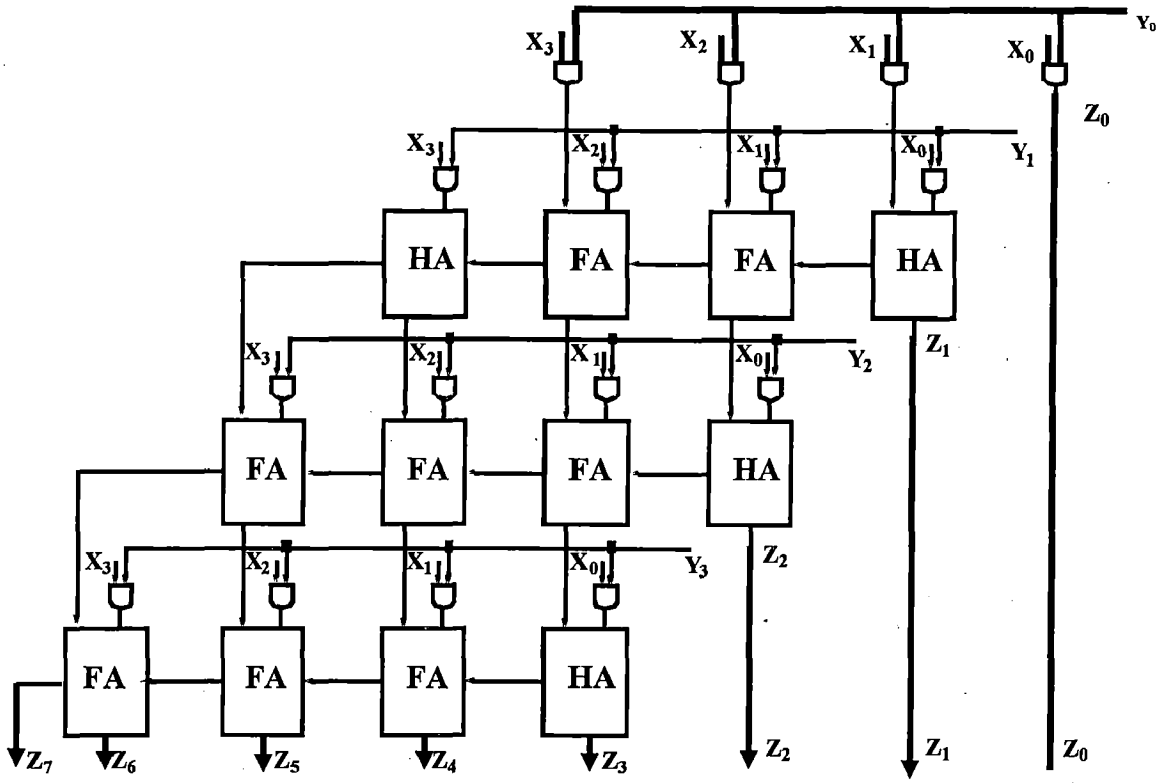


Figure 2.6 Sequential multiplier

2.5.2. Array multipliers

In array multipliers, the counters and compressors are connected in a serial fashion for all it slices of the Partial Product parallelogram. As can be seen in Figure 2.7, the array topology is a two-dimensional structure that fits nicely on the VLSI planar process. Array multipliers can be pipelined to decrease the clock period at the expense of latency.

In this type of array, the output of each row of counters (3:2 compressors) is the input to the next row of counters. In the simple array, each row of [3:2] compressors adds a partial product to the partial sum, generating a new partial sum and a sequence of carries. The delay of the array depends on the depth of the array. Therefore, the summing time for the simple array is N-2 [3:2] compressor delays, where N is the number of partial products. The drawback of this type of array is the hardware is underutilized. The counters are used only once in the calculation of the result, for the remaining time, they are idle. This drawback can be diminished by pipelining the array so that several multiplications can occur simultaneously. Pipelining would increase the throughput of the multiplier, but would also increase the latency and area of the multiplier. A fully pipelined array is normally avoided, since the array would be faster than the clock of processor. Figure 2.7 depicts the layout of a simple array topology. The dots represent the partial products.

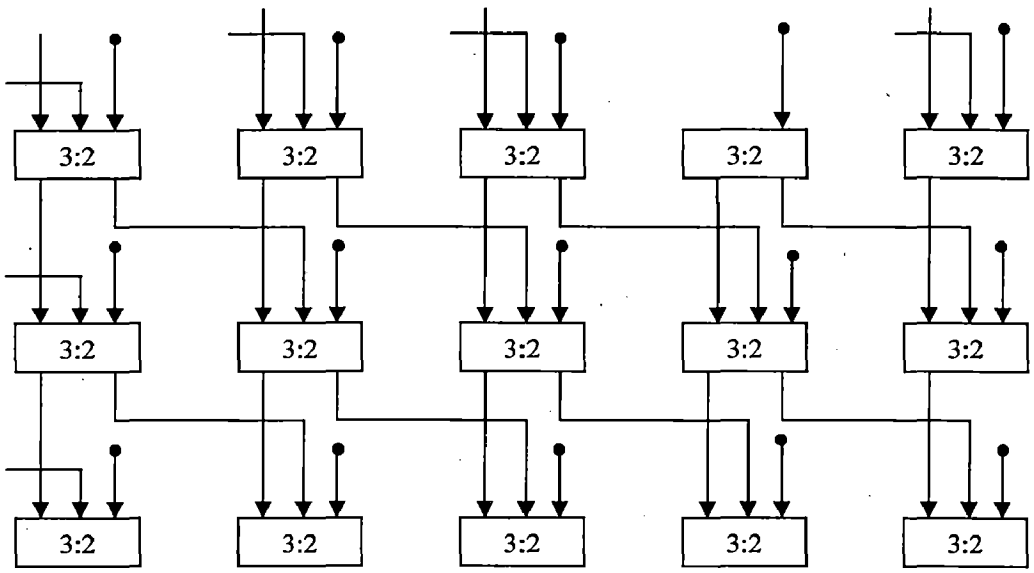


Figure 2.7 Array topology

2.5.3. Tree multipliers

In order to speed up the process of addition of partial products, tree based structure is used. In tree architecture, the compressors are connected for each bit slice in the PP parallelogram. Normally, they are used in parallel. Although the trees are faster than arrays, they both use the same number of compressors to reduce the partial products. The first tree structure was introduced by Wallace. Wallace showed that PPs can be reduced by connecting [3:2] compressors in parallel in a tree topology.

Wallace trees are irregular in the sense that the informal description does not specify a systematic method for the compressor interconnections. However, it is an efficient implementation of adding partial products in parallel. The Wallace tree operates in three steps[16]:

1. *Multiply*: Each bit of multiplicand is ANDed with each bit of multiplier yielding n^2 results. Depending on the position of the multiplied bits, the wires carry different weights.

2. *Addition:* As long as there are more than 3 wires with the same weights add a following layer. Take 3 wires of same weight and input them into a full adder. The result will be an output wire of half-adder and if only one is left, connect it to the next layer.

3. Group the wires in two numbers and add in a conventional adder. A typical Wallace tree architecture is shown in Figure 2.8.

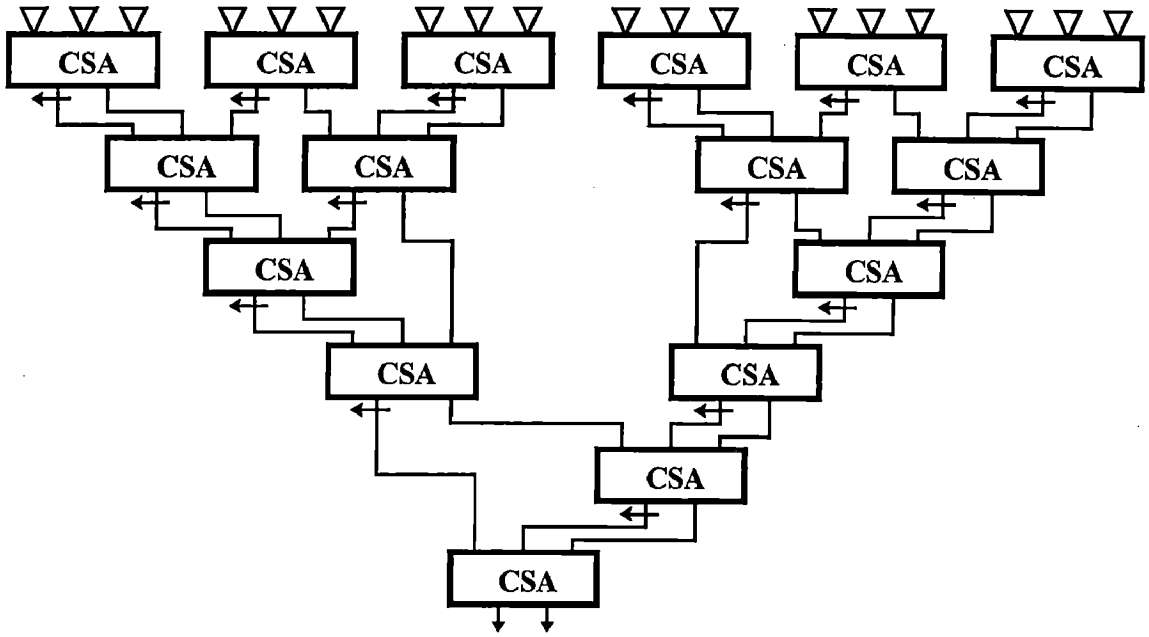


Figure 2.8 Wallace Tree

Chapter 3
POWER CONSUMPTION

Power consumption is one of the basic parameters of any kind of Integrated Circuit (IC). Power and performance are always traded off to meet the system requirements. Power has a direct impact on system cost.

There are two types of power dissipation. One is the maximum power dissipation which is related to the peak of the instantaneous current and the other is the average power dissipation. The peak current has an effect on the supply voltage noise due to the power line resistance. It can cause heating of the device, thus resulting in performance degradation. From the battery lifetime point of view, the average power dissipation is more important.

The power dissipated in a circuit falls into two broad categories:

1. Static power
2. Dynamic power

3.1 Defining Static Power

Static power is the power dissipated by a gate when it is not switching, that is, when it is inactive or static. Static power is dissipated in several ways. The largest percentage of static power results from source-to-drain subthreshold leakage, which is caused by reduced threshold voltages that prevent the gate from completely turning off. Static power is also dissipated when current leaks between the diffusion layers and the substrate. For this reason, static power is often called leakage power.

The static or steady state power dissipation of a circuit is expressed by the following relation[15]

$$P_{stat} = I_{stat} V_{DD} \tag{3.1}$$

Where, I_{stat} is the current that flows through the circuit in the absence of switching activity.

Ideally the portion of static current should be zero as the PMOS and NMOS transistors are never on simultaneously in steady-state operation. But unfortunately, there exits a leakage current flowing through the reverse-biased diode junctions of the transistors located between the source/drain and the substrate as shown in Figure 3.1.

There are two types of leakage currents: *reverse-bias diode leakage* on the transistor drains; and *sub-threshold leakage* through the channel of an "off" device. The magnitude of these currents is set predominantly by the processing technology.

The diode leakage occur when a transistor is turned off and another active transistor charges up/down the drain with respect to the bulk potential of the former. In the case of the inverter with a high input voltage, the output voltage will be low because the NMOS transistor is on. The PMOS transistor will be turned off, but its drain-to-bulk voltage will be equal to the supply voltage V_{DD}. The leakage current density is temperature sensitive, so current density can increase dramatically at higher temperatures.

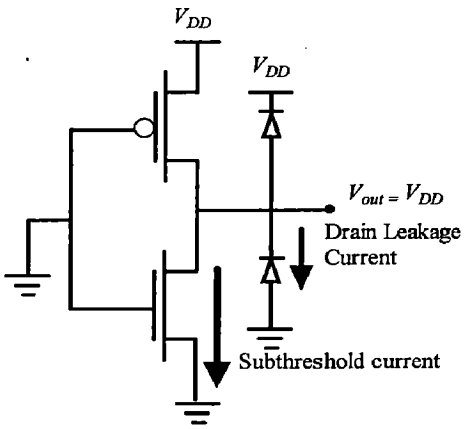


Figure.3.1 Sources of leakage currents in CMOS inverter (for $Vin = 0$ V).

The other source of leakage current is the sub-threshold current of the transistors. MOS transistor can experience a drain-source current, even when V_{GS} is smaller than the threshold voltage. The closer the threshold voltage is to zero volts, the larger the leakage

current at V_{GS} = 0 V and the larger the static power consumption. To offset this effect, the threshold voltage of the device has generally been kept high enough.

3.2 Defining Dynamic Power

Dynamic power is the power dissipated when the circuit is active. A circuit is active anytime the voltage on a net change due to some stimulus applied to the circuit. Because voltage on an input net can change without necessarily resulting in a logic transition on the output, dynamic power can be dissipated even when an output net doesn't change its logic state.

The dynamic power of a circuit is composed of two kinds of power:

1. Switching power
2. Internal power

3.2.1 Switching Power

The switching power of a driving cell is the power dissipated by the charging and discharging of the load capacitance at the output of the cell. The total load capacitance at the output of a driving cell is the sum of the net and gate capacitances on the driving output. Because such charging and discharging are the result of the logic transitions at the output of the cell, switching power increases as logic transitions increase. Therefore, the switching power of a cell is a function of both the total load capacitance at the cell output and the rate of logic transitions.

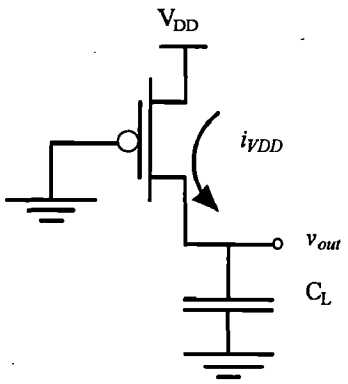


Figure 3.2 Equivalent circuit during the low-to-high transition.

Each time the capacitor C_L gets charged through the PMOS transistor, its voltage rises from 0 to V_{DD}, and a certain amount of energy is drawn from the power supply. Part of this energy is dissipated in the PMOS device, while the remainder is stored on the load capacitor. During the high-to-low transition, this capacitor is discharged, and the stored energy is dissipated in the NMOS transistor.

The values of the energy E_{VDD}, taken from the supply during the transition, as well as the energy E_C, stored on the capacitor at the end of the transition, can be derived by integrating the instantaneous power over the period of interest. The corresponding waveforms of $v_{out}(t)$ and $i_{VDD}(t)$ are pictured in figure 3.3.

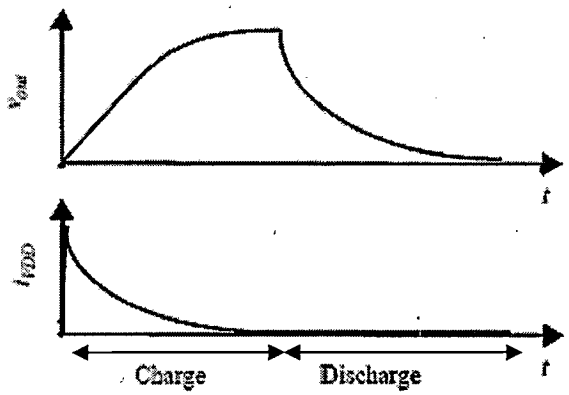


Figure 3.3 Output voltages and supply current during (dis)charge of C_L.

$$E_{VDD} = \int_0^\infty i_{VDD}(t) v_{out} dt = V_{DD} \int_0^\infty C_L \frac{dv_{out}}{dt} dt = V_{DD} C_L \int_0^{V_{DD}} dv_{out} = V_{DD}^2 C_L \tag{3.2}$$

$$E_C = \int_0^\infty i_{VDD}(t) v_{out} dt = \int_0^\infty C_L \frac{dv_{out}}{dt} v_{out} dt = C_L \int_0^{V_{DD}} v_{out} dv_{out} = \frac{C_L V_{DD}^2}{2} \tag{3.3}$$

during the low-to-high transition, C_L is loaded with a charge $C_L V_{DD}$. Providing this charge requires an energy from the supply equal to $C_L V_{DD}^2$ $(= Q \times V_{DD})$. The energy stored on the capacitor equals $C_L V_{DD}/2$. This means that only half of the energy supplied by the power source is stored on C_L. The other half has been dissipated by the PMOS transistor. Notice

that this energy dissipation is independent of the size (and hence the resistance) of the PMOS device! During the discharge phase, the charge is removed from the capacitor, and its energy is dissipated in the NMOS device. Once again, there is no dependence on the size of the device. Each switching cycle (consisting of an L→H and an H→L transition) takes a fixed amount of energy, equal to $C_L V_{DD}$. In order to compute the power consumption, we have to take into account how often the device is switched. If the gate is switched on and off $f_{0\to1}$ times per second, the power consumption equals

$$P_{dyn} = C_L V_{DD}^2 f_{0\to1} \tag{3.4}$$

$f_{0\to1}$ represents the frequency of energy-consuming transitions, this is 0→1 transitions for static CMOS.

3.2.2 Internal Power

The finite slope of the input signal causes a direct current path between V_{DD} and *GND* for a short period of time during switching, while the NMOS and the PMOS transistors are conducting simultaneously. Internal power includes power dissipated by a momentary short circuit between the P and N transistors of a gate, called short-circuit power.

Short circuit currents occur when the rise/fall time at the input of a gate is larger than the output rise/fall time. For the ideal case of a step input, the transistors change state immediately, one turning on and other turning off. There is not a conductive path from the supply to ground. For real circuits, however, the input signal will have some finite rise/fall time. While the condition $V_{tn} \le V_{in} \le V_{dd} - V_{tp}$ holds for the input voltage, there will be a conductive path open because both devices are on. The longer the input rise/fall time, the longer the short-circuit current will continue to flow, and the average short-circuit current will increase. To minimize the total average short-circuit current power, it is desirable to have equal input and output edge times. Short-circuit current power is either linearly or quadratic dependent on the supply voltage, depending on the size of the channel length. While reducing the supply increases the duration of the current linearly due to increased rise/fall times, the peak magnitude of the current is reduced linearly (velocity saturation) such that the average current is approximately constant and the average power is just a linear function of supply voltage (P=IV). For larger devices that

are not velocity saturated, the average current is approximately linear with supply voltage so that the average power is a quadratic function of supply voltage

We can compute the average power consumption as follows.

$$P_{sc} = t_{sc} V_{dd} I_{peak} f = C_{sc} V_{dd} f \tag{3.5}$$

t_{sc} represents the time both devices are conducting.

For most ICs, the short-circuit power dissipated is approximately 5-10% of the total dynamic power, if the supply is lowered to below the sum of the thresholds of the transistors i.e. $V_{dd} < V_{tn} + V_{tp}$. However, short-circuit currents will be eliminated because both devices cannot be on at the same time for all values of input voltage.

3.3 Low power or energy design techniques

Power consumption becomes an issue in complex electronic systems where cost is extremely important. Reducing power consumption is an important design task for IC engineers. Power is important for portable equipment like mobile, laptop, PDA, GPS, hearing aids and wrist watch etc. The requirement is a long life battery and a light system which is only possible if the equipment consumes bare minimum power. Lowering power also reduces the cost for cooling system and makes the chip package smart reducing the size of the device. There are a number of low power techniques available for CMOS circuits. Some of them are discussed in subsequent sections.

3.3.1 Pipelining

Pipelining is a popular design technique to reduce power consumption by increasing the throughput of logic blocks and processors to reduce frequency and supply voltage[2]. Pipelining is used to reduce power consumption, as illustrated in Figure 3.4. The idea is to insert registers after some appropriate distance in the circuit. The system response becomes faster than before. In order to maintain the previous delay, the supply voltage is reduced which reduces the power consumption.

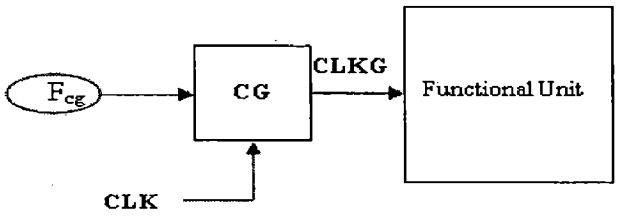


Figure 3.5 Clock-gating principle[2].

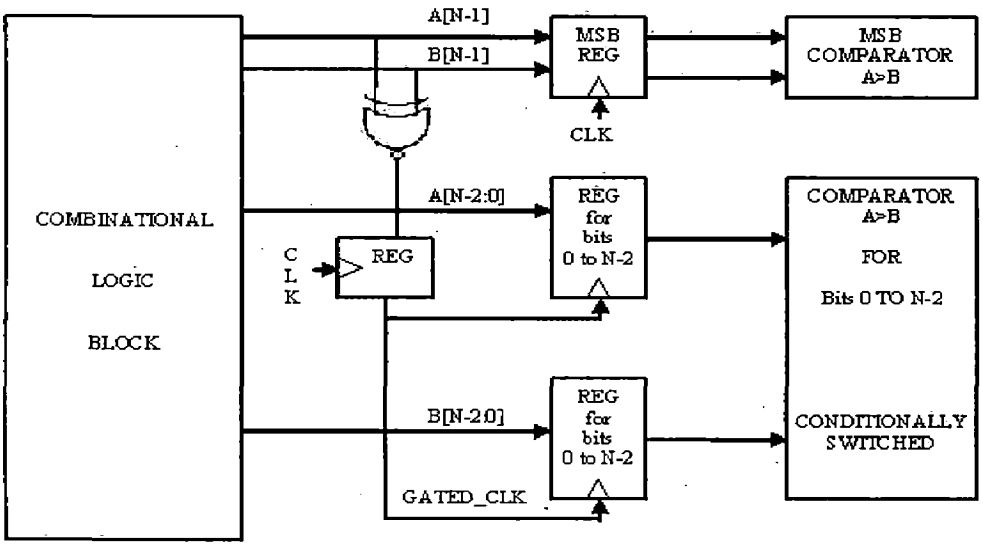
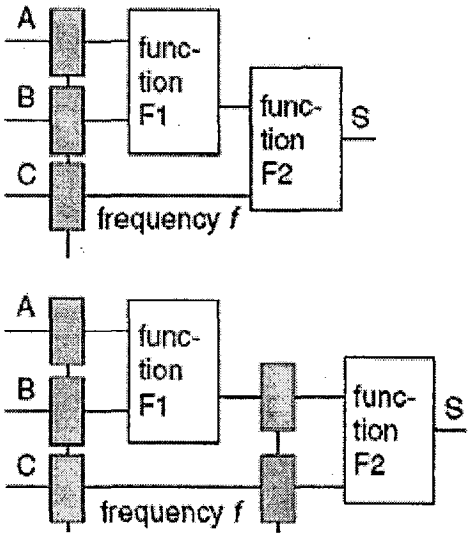


Figure 3.6 Using gated clocks to reduce power[18].

Assume a pipelined system for comparing the output of two numbers from a block of combinational logic as shown in Fig. 3.6, the first pipeline stage is a combinational block and the next pipeline stage is a comparator which performs the function $A > B$, where A and B are generated in the first stage (i.e., from the combinational block). If the most significant bits, *A [N -1]* and *B[N-1]*, are different then the computation of $A > B$ can be performed strictly from the MSB's and therefore the comparator logic for bits *A[N-2:0]* and *B[N-2 : 0]* is not required (and hence the logic can be powered down). If the data is assumed to be random (i.e., there is a 50% chance that *A[N - 1]* and *B [N - 1]* are different), the power savings can be quite significant. One approach to accomplish this is to gate the clocks as shown in Fig. 3.6. The XNOR output of the *A [N -1]* and *B[N - 1]* is latched by a special register to generate a gated clock. This gated clock is then used to clock the lower order registers.



Baseline:
Power:
$P_1 = f * C * V_{DD}^2$

Pipeline:
Power:
$P_2 = f * C * 1.2 * V_{DD_{low}}^2$
As F1 or F2 are faster than
F1 + F2, one can reduce V_{dd}.

Figure 3.4 Pipeline for low power[2].

3.3.2 Clock gating

Clock gating is very effective in reducing the power consumption in digital circuits. The goal of this technique is to disable or suppress transitions from propagating to parts of the clock path (i.e., flip-flops, clock network, and logic) under a certain condition computed by clock-gating circuits. The savings are mainly due to the switching capacitance reduction in the clock network and the switching activity in the logic fed by the storage elements because unnecessary transitions are not loaded when the clock is not active.

Clock gating (CG) is illustrated in Figure 3.5. A block CG, which inhibits the clock signal when the idle condition is true, is associated with each sequential functional unit.. The clock signal is computed by function Fcg. CLK is the system clock and CLKG the gated clock of the functional unit. Clock-gating techniques have been successfully implemented in many microprocessors.

different), the power savings can be quite significant. One approach to accomplish this is to gate the clocks as shown in Fig. 3.6. The XNOR output of the *A [N -1]* and *B[N - 1]* is latched by a special register to generate a gated clock. This gated clock is then used to clock the lower order registers.

3.3.3 Voltage scaling

In CMOS circuits, the dominant component of power consumption is proportional to $V_{DD}^2 f$, where V_{DD} is voltage and f is frequency. Energy is product of power and time. The time to run a certain number of cycles is inversely proportional to frequency, so energy per cycle is proportional to V_{DD}^2 . At a given voltage, the maximum frequency at which the circuit can run safely decreases with decreasing voltage. Thus, the system can reduce energy consumption by reducing supply voltage, but this necessitates running at a slower speed.

3.3.4 Delay balancing

Glitches in the circuit consume a considerable amount of power. They are produced by a delay in the arrival of input signals at a certain gate in the circuit. Delay balancing is then used to minimize the glitches which in return save power. Suppose we want to add four inputs A, B, C and D. One way to add them is to feed the inputs sequentially as shown in Figure 3.7 (a). Suppose the delay of one adder is T_{add}. The inputs A and B arrive at the same time at adder 1 but there exits a delay of 1 T_{add} and 2 T_{add} between the inputs arriving at adder 2 and 3 respectively. This would produce glitches at the output node. In order to remove glitches, we have to balance the delays at the inputs of each adder. Figure 3.7 (b) shows a modified architecture in which the delays are balanced at the input of each adder.

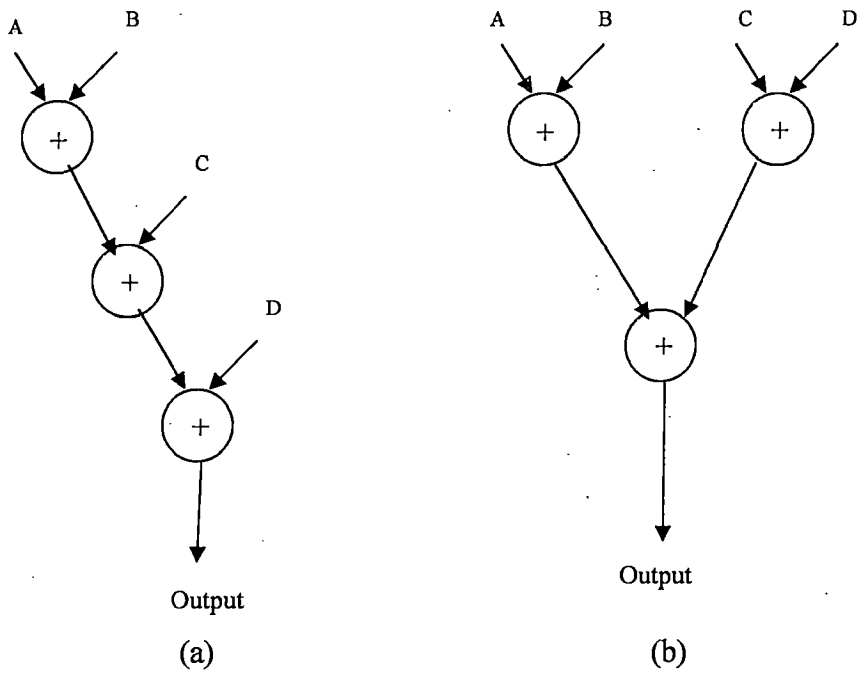


Figure 3.7 Delay balancing example

3.3.5 Transition activity reduction

The power consumption is directly proportional to the transition or switching activity of the circuit. So reducing transition activity will also reduce the power consumption of the circuit. To reduce the transition activity in complex electronic systems, some special encodings are used for data and address busses. These encodings are very effective in reducing the power consumption. These are Limited Weight Codes (LWC), Zero-transition encoding (T0), Bus-invert encoding (BI), T0_BI encoding, Dual_T0 encoding etc. an example of Bus invert encoding is shown in Table 1. In *bus-invert encoding*, an extra line is used. If the extra bit is zero, the original bits are kept intact and they are inverted if the extra bit is 1. The overhead is one extra bit line but the number of transitions is reduced significantly which in turn reduces the power consumption.

Input sequence	Number of Transitions Number of Transitions	Bus_invert Encoded sequence	New number of transitions
0000	1	0 0000	1
0001	1	0 0001	1
0010	2	0 0010	2
0011	1	0 0011	1
0100	3	1 1011	2
0101	1	1 1010	1
0110	2	1 1001	2
0111	1	1 1000	1
1000	4	0 1000	1
Total transitions	**16**		**12**

Table 3.1 Bus invert encoding[19]

Chapter 4

MULTIPLIER ARCHITECTURES

This chapter describes architectures of all the three implemented multipliers in detail.

4.1 The architecture of the 2-dimensional pipelined gated Booth multiplier

In 2-D gating technique, clock is gated to registers in both vertical direction (data flow direction in pipeline) and horizontal direction (within each pipeline stage). The proposed power-aware scalable 2-D pipeline gated Booth multiplier consists of a shared radix-4 Booth encoder, a shared and configurable partial product generation unit, shared and pipelined partial product reduction unit ,a multiplexer and a shared final ripple carry adder shown in Figure 4.1. Based on the gated input signals the gated clock signals are generated that appropriately selects required parts of the multiplier and multiplicand operands, the booth encoder and partial product generator unit, the partial product reduction units, and ripple carry adder unit for given data precision.

Depending on the number of multiplier bits, the Booth encoder and partial product generator adjust the number of partial products generated while maintaining the unused partial product generator sections in static condition. For shorter precisions the unused parts of the partial product reduction and ripple carry adder units are deactivated using gated clock signals. The final product is generated from the active parts of the booth encoder, partial product generator, partial product reduction, and ripple carry adder units.

The functional units of the proposed 16-bit multiplier are described below

4.1.1 Pipelined Gating Technique

Latched-based clock gating technique used in each of the five pipeline stages enables the multiplier circuit to deactivate the unused part of the logic and avoid excessive power-consumption in each multiplication operation. Synopsys Power Compiler was utilized for generating a latch-based clock gating circuit. Each pipeline stage in the proposed

multiplier is optimized to minimize the amount of switching in the logic between the pipeline stages and also within the pipeline registers.

In the first stage pipeline registers is partitioned into three states, the most significant 8-bits of the input operand are gated only if gated signal 3 high, and the middle 4-bits are gated if both gated signal 3 and gated signal 2 are high, and finally the least 4-bits are gated when all the gated signals are high.

The second pipeline stage is after the Booth encoder and the Partial product generator unit. Depending on the gated clock signals it allows required partial product bits to the next stage. Gated clock1 allows only the first 5bits of the first and second partial products. Gated clock2 allows first 9 bits of the second and third partial products and next higher order 4 bits of the first and second partial product bits. And Gated clock3 allows 17 bits of rest of the four partial products and next higher order 8 bits of first four partial products.

The third pipeline stage, after first partial product reduction block, is gated by either Gated clock2 or Gated clock3. First 15 bits of first and second rows and first 8 bits of the third row at the output of first partial product reduction unit are gated by the Gated clock2. The rest of the 16 bits in first and second rows, 8 bits in third row, 10 bits in fourth row and 3 bits in the fifth row are gated by gated clock3.

The fourth pipeline stage, after second partial product reduction block, is gated only by Gated clock3.

The fifth pipeline stage, after MUX and final partial product reduction block, is gated by either of Gated clock1, Gated clock2 or Gated clock3. First 8 bits of first and second rows output of final partial product reduction unit are gated by the Gated clock1. Next higher order 8 bits of first and second rows output of final partial product reduction unit are gated by the Gated clock2. The rest of the 16 bits in first and second rows are gated by gated clock3.

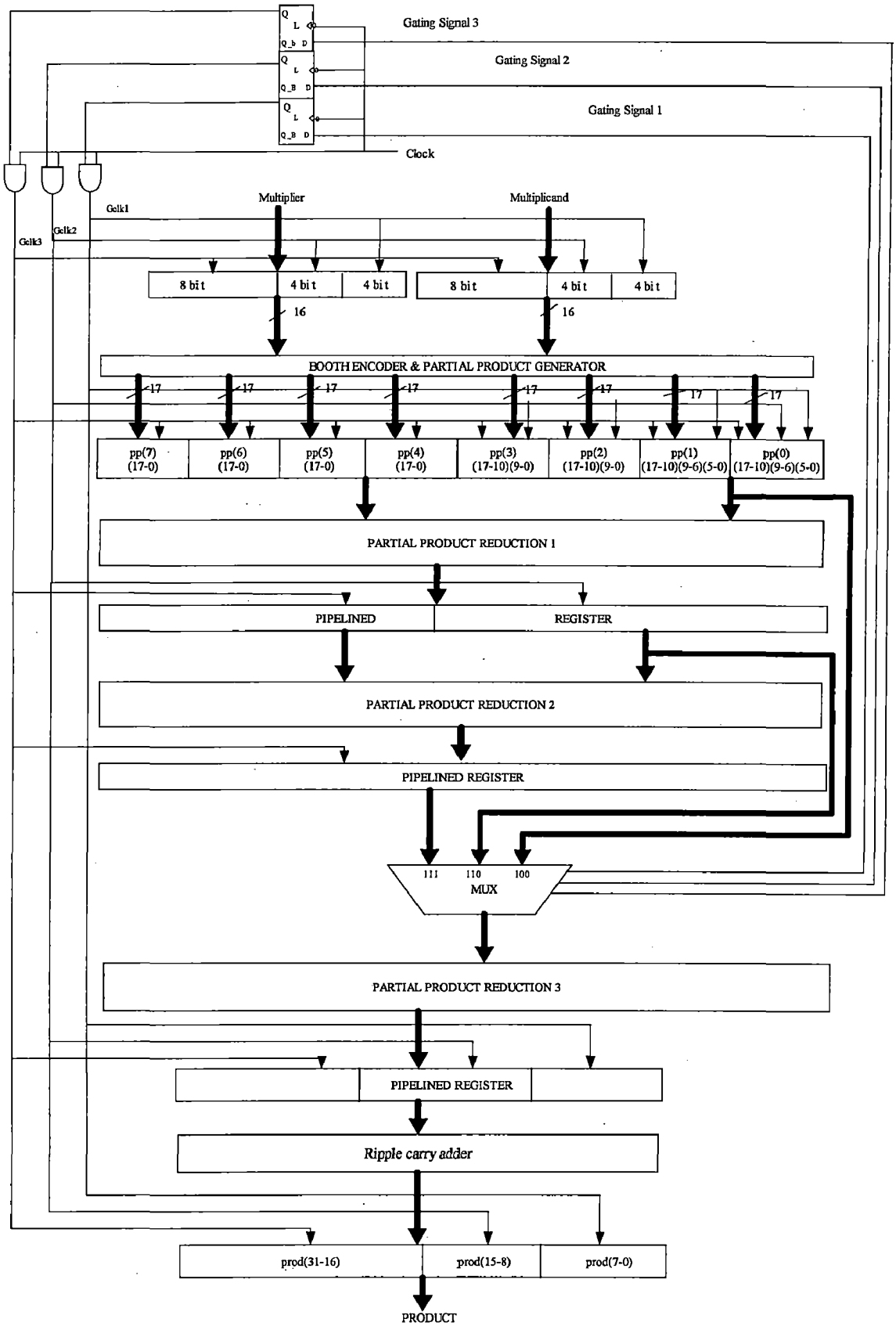


Figure 4.1 2D-Power-aware reconfigurable pipelined Booth multiplier.

The final pipeline stage, after ripple carry adder, is gated by either of Gated clock1, Gated clock2 or Gated clock3. First 8 bits output of ripple carry adder are gated by the Gated clock1. Next higher order 8 bits output of ripple carry adder are gated by the Gated clock2. The rest of the 16 bits are gated by gated clock3.

4.1.2 Booth Encoder and Partial Product Generator Unit

Booth encoder and partial product generator unit is configurable and can be shared between the 16-bit, 8-bit and 4-bit multiplication modes. The total numbers of partial products (PP) generated are N/2 (N = max. number of multiplier bits), where PP_i is can be zero, complement, twice, twice the complement of the multiplicand or multiplicand depending on the multiplier bits. The 4-bit multiplication mode only requires first two of the partial products and 8-bit multiplication requires only first four partial products. There are three types of configuration modes for the partial product, which are consistent with the operation modes of the power-aware multiplier. In the 4-bit multiplication, partial products are of 5 bit length, in the 8-bit multiplication partial products are of 9-bit length and in the 16-bit multiplication they are of 17-bit length. Depending on the multiplier and multiplicand bits that are gated in the pipeline stage before this block, only the required partial products are generated and the rest of the bits are held in the static state.

4.1.3 Partial product reduction units

Partial product reduction units are shared between all of 16-bit, 8-bit and 4-bit multiplication modes. All partial product reduction units employ either [3:2] compressors or [2:2] compressors for reduction of partial products.

The partial product summation is done in a Wallace-tree structure. Wallace-tree is divided into three blocks. After the first block, the first three rows contain bits in case of the 8-bit multiplication and after the second reduction the first three rows are the rows after reduction in case of the 16-bit multiplication. Outputs of both of the reduction units along with the two rows after the Booth-encoder and partial product generation unit are applied to the multiplexer unit, which are then passed to the final reduction unit for final reduction depending on the mode of multiplication along with their correction vector.

The final reduction unit reduces the three rows into two 32-bit rows which are then added to the ripple carry adder unit.

The compressor configuration for each of the blocks is shown in the figures 4.2 through 4.4 (in final reduction the bits shown are in case of 16-bit multiplication).

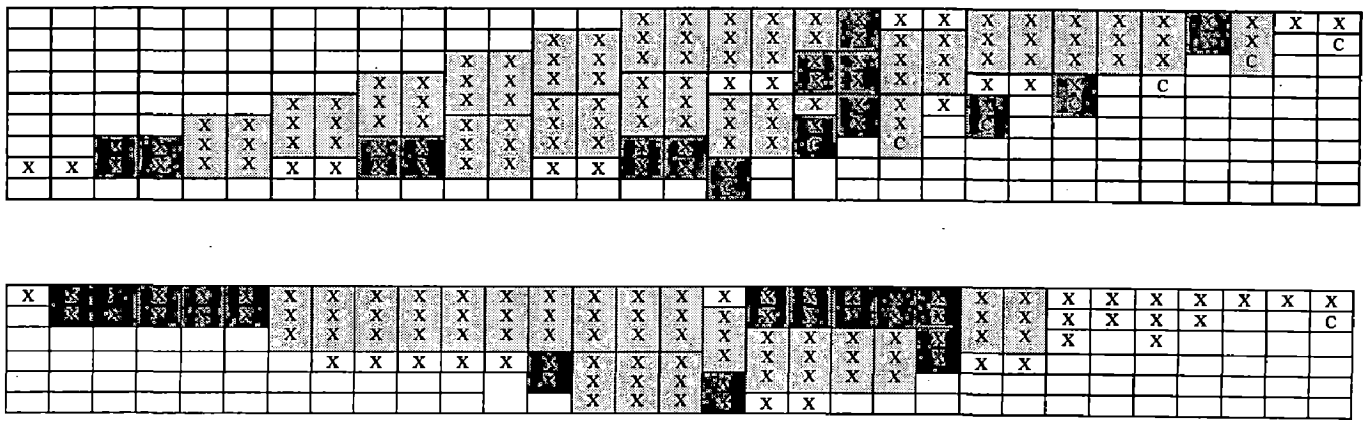


Figure 4.2 First Reduction Unit.

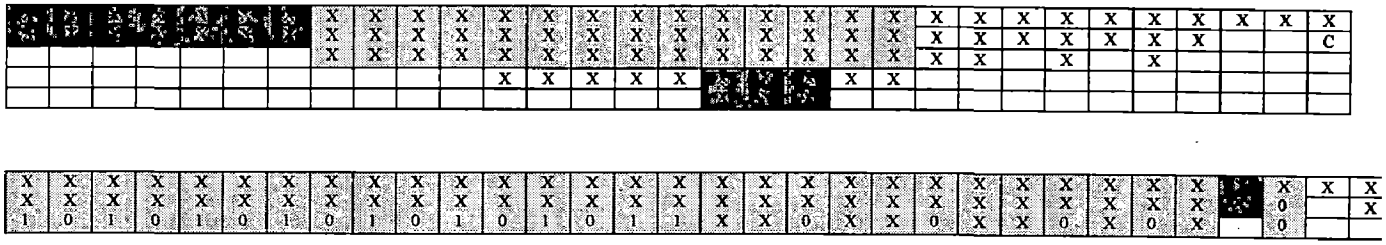


Figure 4.3 Second Reduction Unit.

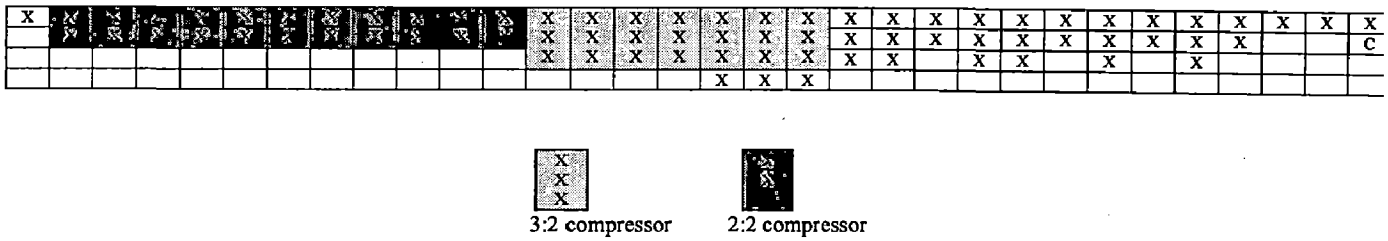


Figure 4.4 Final Reduction Unit (last row is in case of 16 bit multiplication)

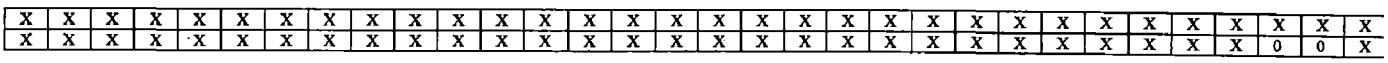


Figure 4.5 Input to the ripple carry adder

4.1.4 Multiplexer Unit

This block selects the outputs from booth encoder and partial product generation unit, first second partial product reduction unit or second partial product reduction unit. In addition, it also adds the correction vectors required for sign extension prevention based on the type of multiplication (16- , 8- or 4- bit). The output of this stage is given to final partial product reduction unit.

4.1.5 Ripple carry adder

The ripple carry adder is also shared unit between all of the 16-bit, 8-bit and 4-bit multiplications. The bits that are not used for the particular mode of multiplication are held at static condition.

4.2 The architecture of the 1-dimensional pipeline gated Booth multiplier

In 1-D gating technique, clock is gated to registers in vertical direction (data flow direction in pipeline) or in horizontal direction (within each pipeline stage). Gating in vertical direction is used in the thesis.

The reconfigurable 1-D pipeline gated Booth multiplier similar to 2-D pipeline gated multiplier consists of a shared radix-4 Booth encoder, a shared partial product generation unit, shared and pipelined partial product reduction unit , a multiplexer and a shared final ripple carry adder. Only difference is that the gating is applied only in vertical direction.

The functional units of the 1D pipeline gated 16-bit multiplier are described below

4.2.1 Pipelined Gating Technique

In the first stage pipeline registers is partitioned into three states for multiplier, the most significant 8-bits of multiplier are gated only if gated signal 3 high, and the middle 4-bits are gated if both gated signal 3 and gated signal 2 are high, and finally the least 4-bits are gated when all the gated signals are high.

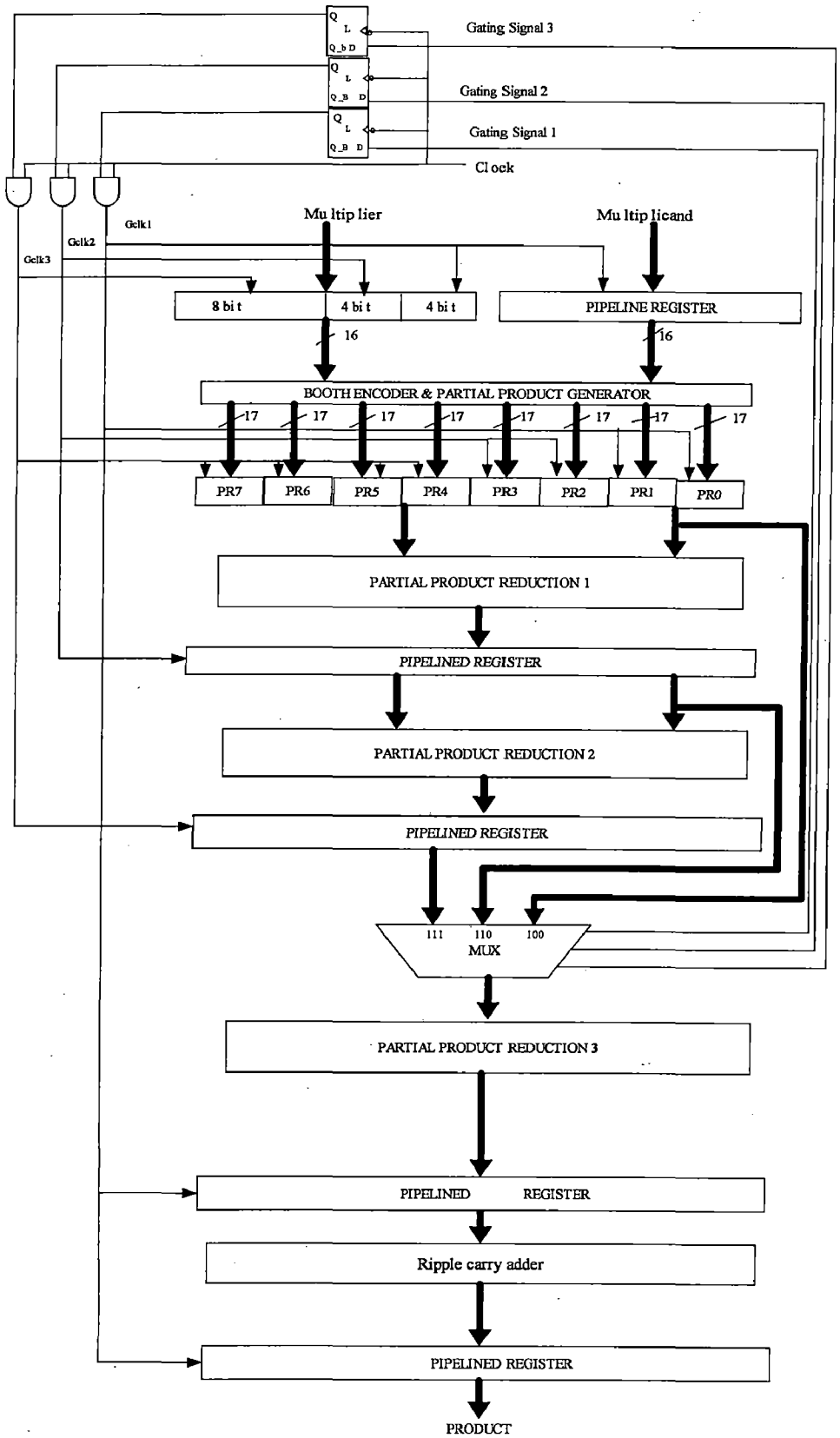


Figure 4.2. 1D-Power-aware scalable pipelined Booth multiplier.

All of the multiplicand bits are passed.

The second pipeline stage is after the Booth encoder and the Partial product generator unit. Depending on the gated clock signals it allows required partial product bits to the next stage. Gated clock1 allows first and second partial products. Gated clock2 allows second and third partial products. And Gated clock3 rest of the four partial products.

The third pipeline stage, after first partial product reduction block, is gated by r Gated clock2.

The fourth pipeline stage, after second partial product reduction block, is gated by Gated clock3.

The fifth pipeline stage, after MUX and final partial product reduction block, is gated by either of Gated clock1.

The final pipeline stage, after ripple carry adder, is gated by Gated clock1.

4.2.2 Booth Encoder and Partial Product Generator Unit

This unit is not configurable, unlike the 2-D case. All the partial products generated are of 17-bits. But, depending on the multiplier bits passed some or all of the partial products are valid and remaining partial products are held in a static condition.

4.2.3 Partial product reduction units

Similar to 2-D case, partial product summation is done in a Wallace-tree structure. Wallace-tree is divided into three blocks. After the first block, the first three rows contain bits in case of the 8-bit multiplication and after the second reduction the first three rows are the rows after reduction in case of the 16-bit multiplication. Outputs of both of the reduction units along with the two rows after the Booth-encoder and partial product generation unit are applied to the multiplexer unit, which are then passed to the final reduction unit for final reduction depending on the mode of multiplication along with their correction vector. The final reduction unit reduces the three rows into two 32-bit rows which are then added to the ripple carry adder unit.

The compressor configuration for each of the blocks is shown in the figures 4.7 through 4.9 (in final reduction the bits shown are in case of 16-bit multiplication).

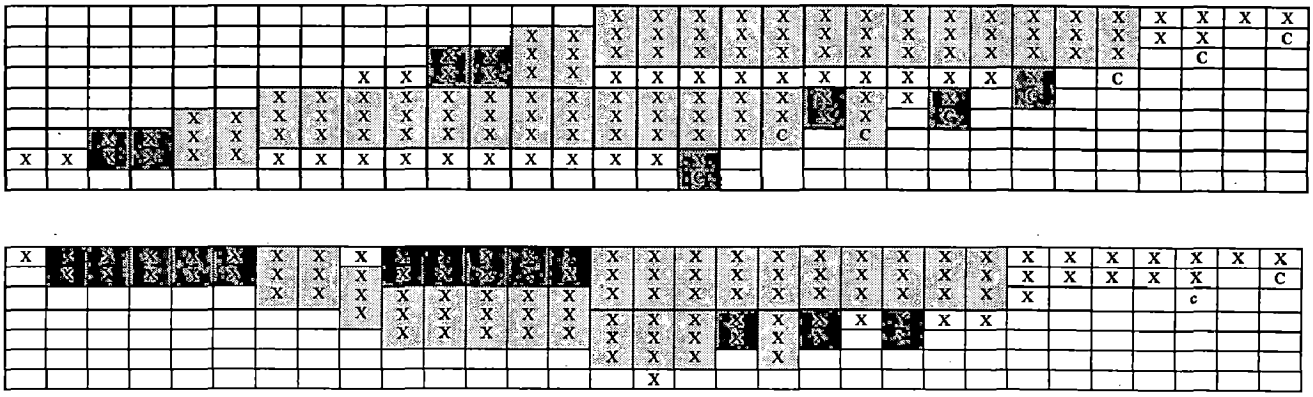


Figure 4.7 First Reduction Unit.

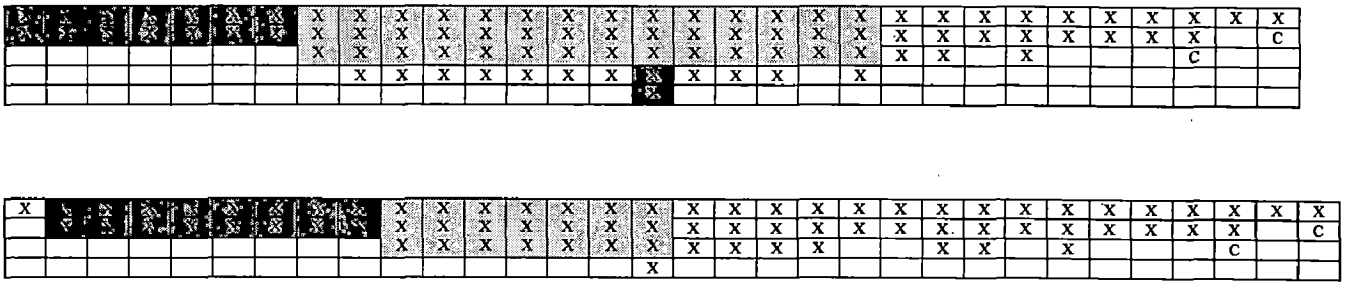


Figure 4.8 Second Reduction Unit.

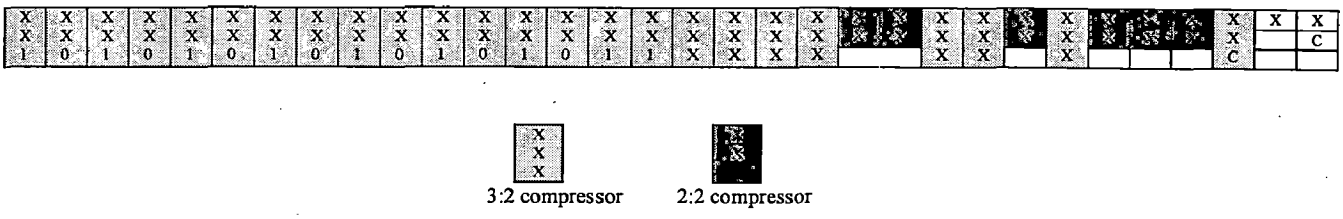


Figure 4.9 Final Reduction Unit.

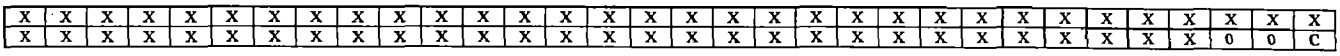


Figure 4.10 Input to the ripple carry adder

4.2.4 Multiplexer Unit

This block selects the outputs from booth encoder and partial product generation unit, first second partial product reduction unit or second partial product reduction unit. In addition, it also adds the correction vector required for sign extension prevention only in case of 16-bit multiplication, in 8-bit and 4-bit multiplication partial products are sign extended to 16-bit and 8-bit respectively. The output of this stage is given to final partial product reduction unit.

4.2.5 Ripple carry adder

The ripple carry adder is also shared unit between all of the 16-bit, 8-bit and 4-bit multiplications.

4.3 The architecture of the non-pipeline gated booth multiplier

The non-pipeline gated Booth multiplier consists of a radix-4 Booth encoder, a partial product generation unit, partial product reduction unit and a final ripple carry adder, is shown in figure 4.12. Booth encoder, the partial product generation unit and ripple carry adder are same as the 1D case. The partial product reduction unit is an optimized Wallace-tree, shown in figure 4.11(the last column is the 3rd column in the reduction tree).

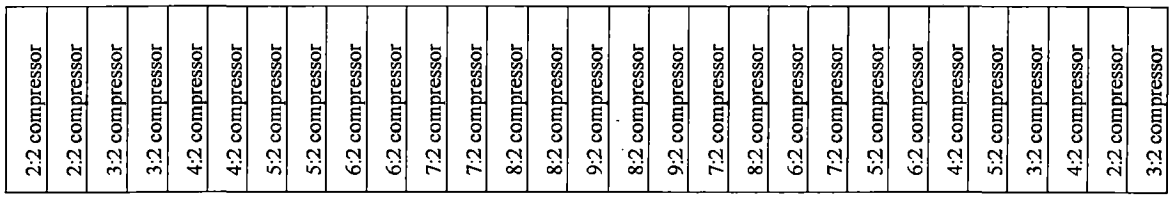


Figure 4.11 Wallace tree reduction

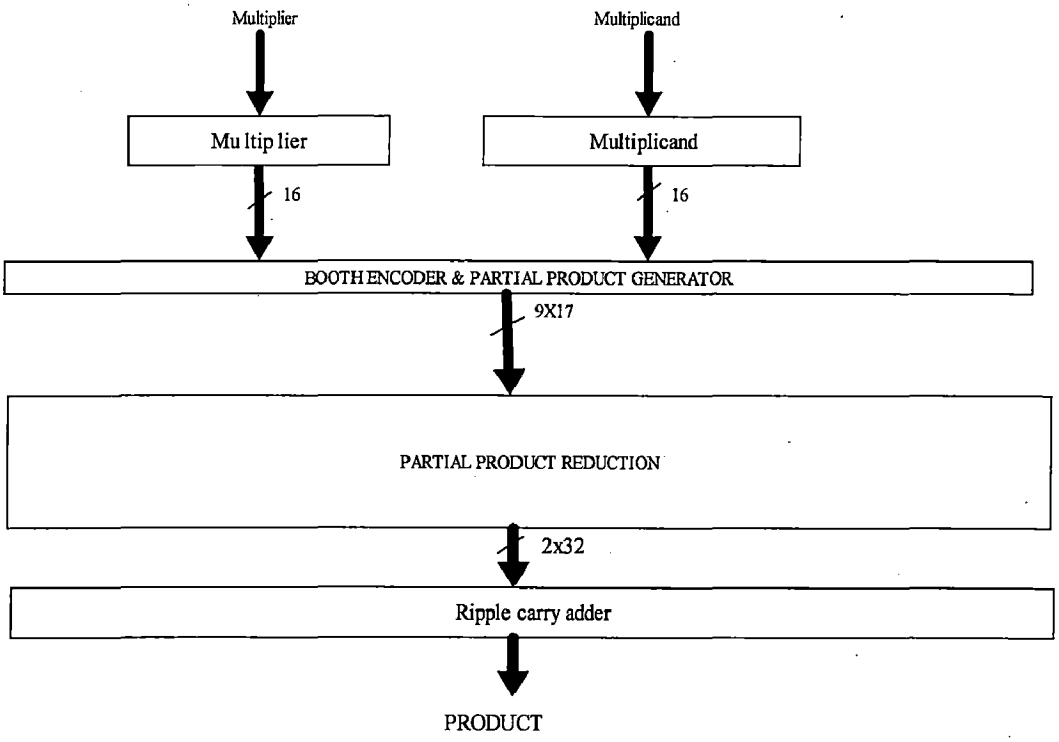


Figure 4.12 Booth multiplier

Chapter 5
RESULTS AND ANALYSIS

The proposed reconfigurable pipelined Booth multiplier was first modeled in VHDL and functionally verified using Modelsim simulator. VHDL simulations were conducted using uniformly distributed random input test vectors with a supply voltage of 1V under typical conditions. After functional validation, the architecture was synthesized for appropriate time and area constraints using SYNOPSYS Design Compiler [20]. TSMC 90nm CMOS technology and standard cell library were used.

Figure 5.1 through 5.9 shows the outputs of the synthesized non-clock gated, 1-dimensional pipeline gated and 2-dimensional pipeline gated circuits at different multiplication modes (16-bit, 8-bit and 4-bit).

The power analysis of the gate level structure has been conducted using Synopsys VCS and Primepower tools [21]. Figures 5.10 and 5.11 shows comparison of peak and average power consumption for different Booth multipliers in different input precisions (16-bit, 8-bit, 4-bit).

From the figures 5.10 and 5.11 and tables 5.1, several observations are made:

1. Among the three multipliers in each figure, the designs using 1-D and 2-D pipeline gating techniques have lower power dissipations compared to the non-pipelined gating designs under different input precision.

2. Among all three multipliers, the designs using 2-D pipeline gating techniques show significant power savings over the corresponding designs using 1-D pipeline gating technique. This advantage is not large in 8-bit multiplication (19.8% under equal input precision probability), but becomes much greater in 4-bit multiplication (44.22% under equal input precision probability), and 16-bit

multiplication have almost same power consumption. The reason for this difference is that as the length of multiplication goes up, the number of registers and the components that are active increases. 1-D pipeline gating technique only deals with the vertical pipeline stage increment, while 2-D pipeline gating technique controls the registers in both directions.

3. The area overhead of implementing 2-D over 1-D techniques is very small (0.18% in 6-stage 16-bit multiplier).

4. Peak power dissipation affects the system reliability in operating under power constraints. 1-D and 2-D pipeline gating techniques both have the ability to reduce system peak power dissipation. But the same as average power dissipation, 2-D technique has great advantage over 1-D technique under different input precisions.

5. Area overhead for 6-stage 1-D and 2-D pipeline gating techniques over non-pipeline technique are 17.8% and 18% respectively for 6-pipeline stages. Area is more because the number registers required are more.

The pipeline latency reduction of the designs using non-pipeline gating, 1-D and 2-D pipeline gating techniques is the same.

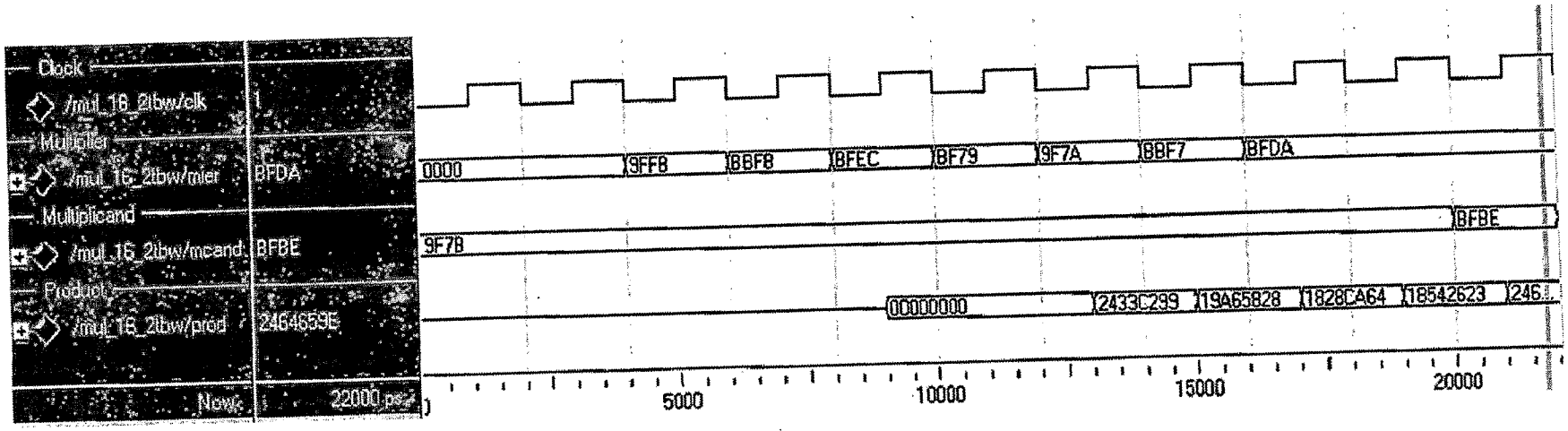


Figure 5.1 Output of multiplier without clock gating after synthesis for 16-bit multiplication.

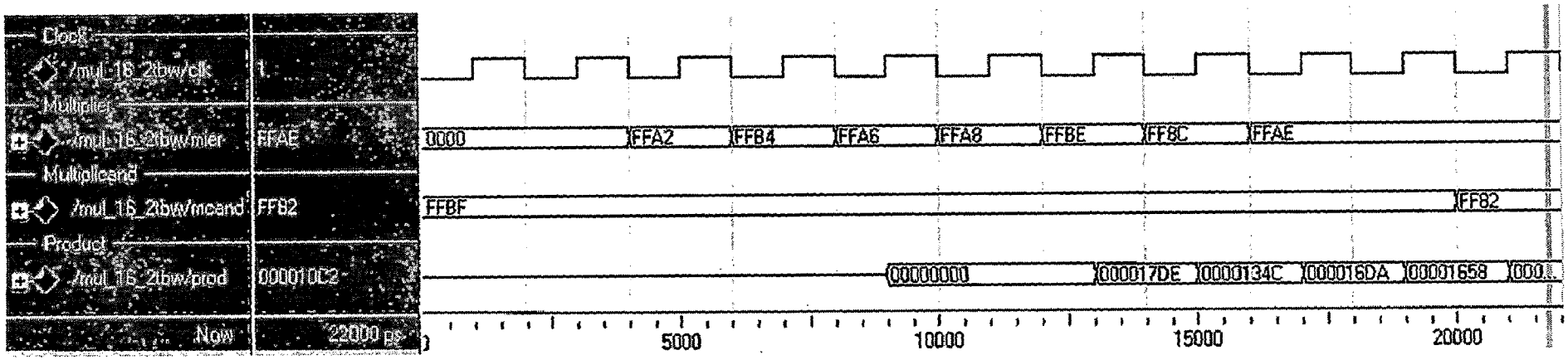


Figure 5.2 Output of multiplier without clock gating after synthesis for 8-bit multiplication.

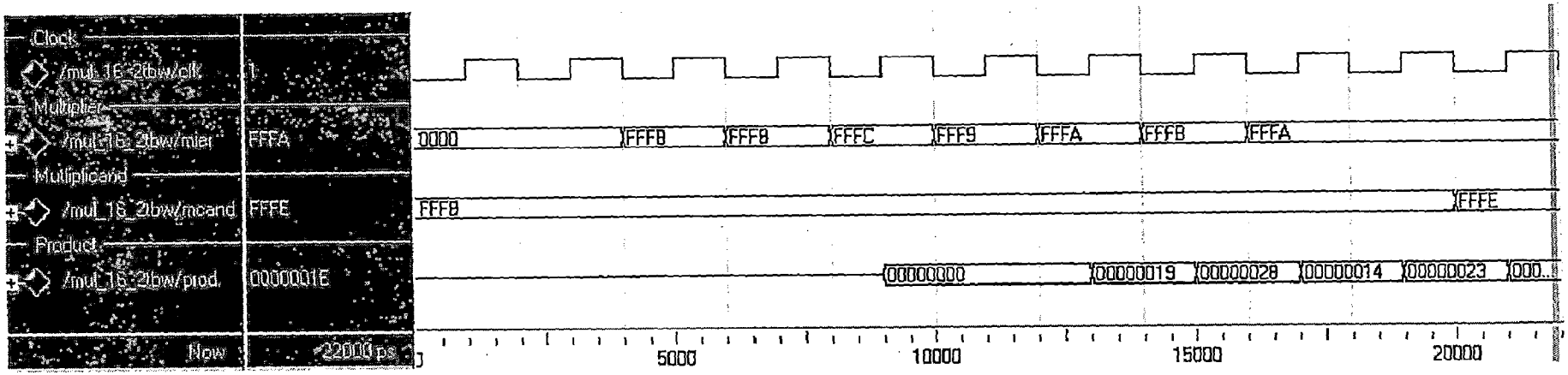


Figure 5.3 Output of multiplier without clock gating after synthesis for 4-bit multiplication.

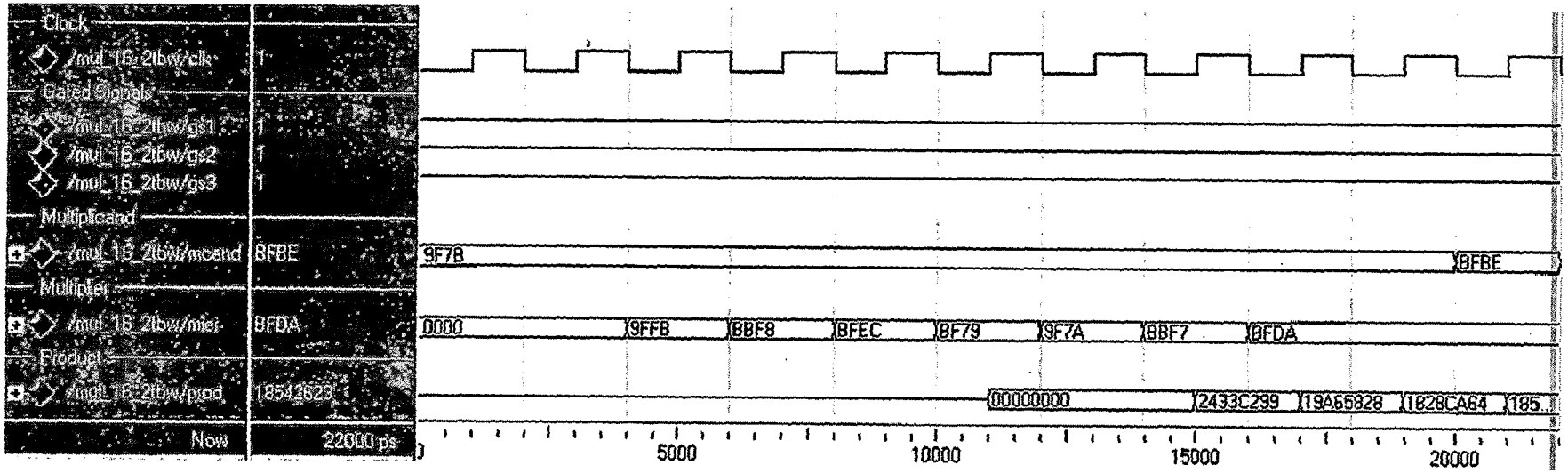


Figure 5.4 Output of multiplier with 1-dimensional clock gating after synthesis for 16-bit multiplication.

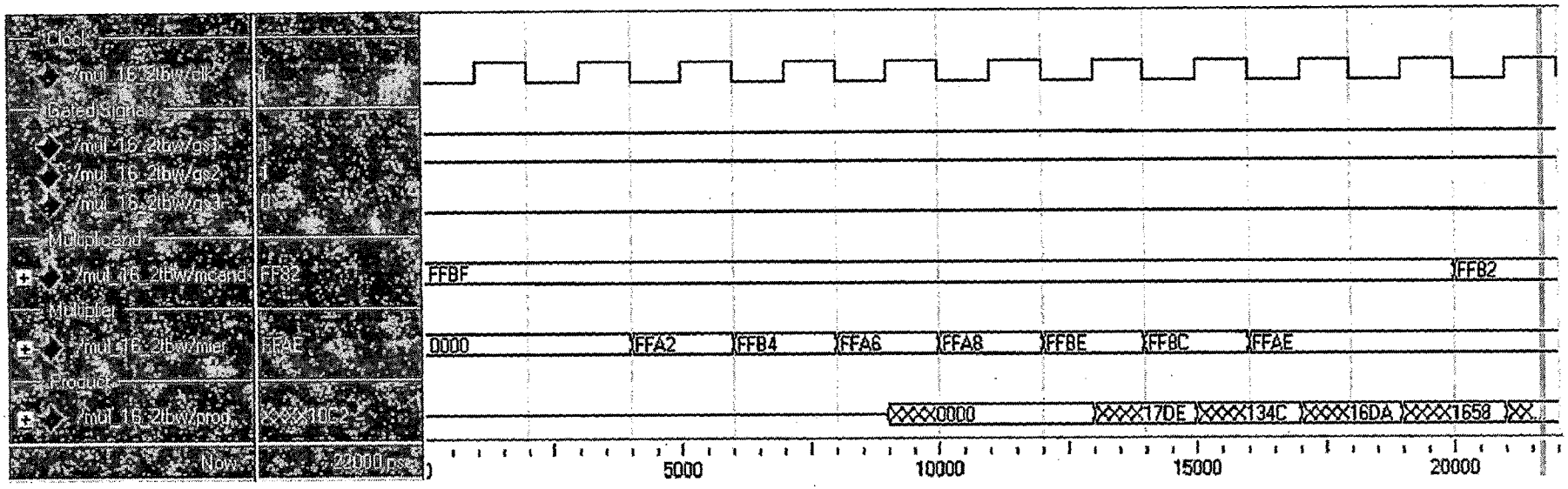


Figure 5.5 Output of multiplier with 1-dimensional clock gating after synthesis for 8-bit multiplication.

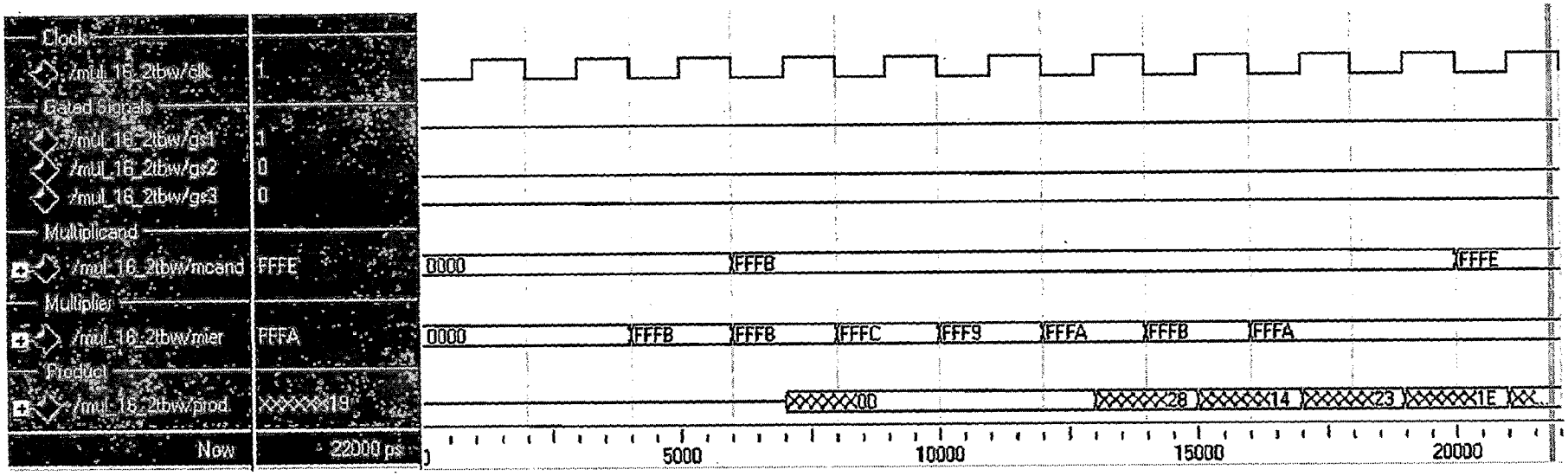


Figure 5.6 Output of multiplier with 1-dimensional clock gating after synthesis for 4-bit multiplication.

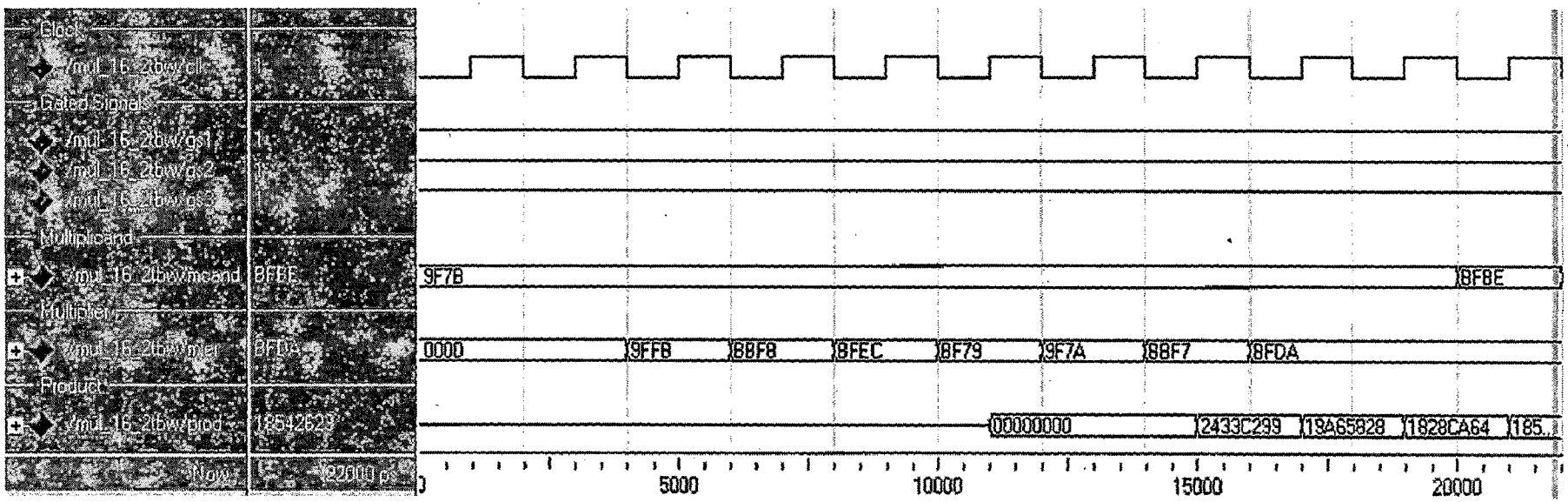


Figure 5.7 Output of multiplier with 2-dimensional clock gating after synthesis for 16-bit multiplication.

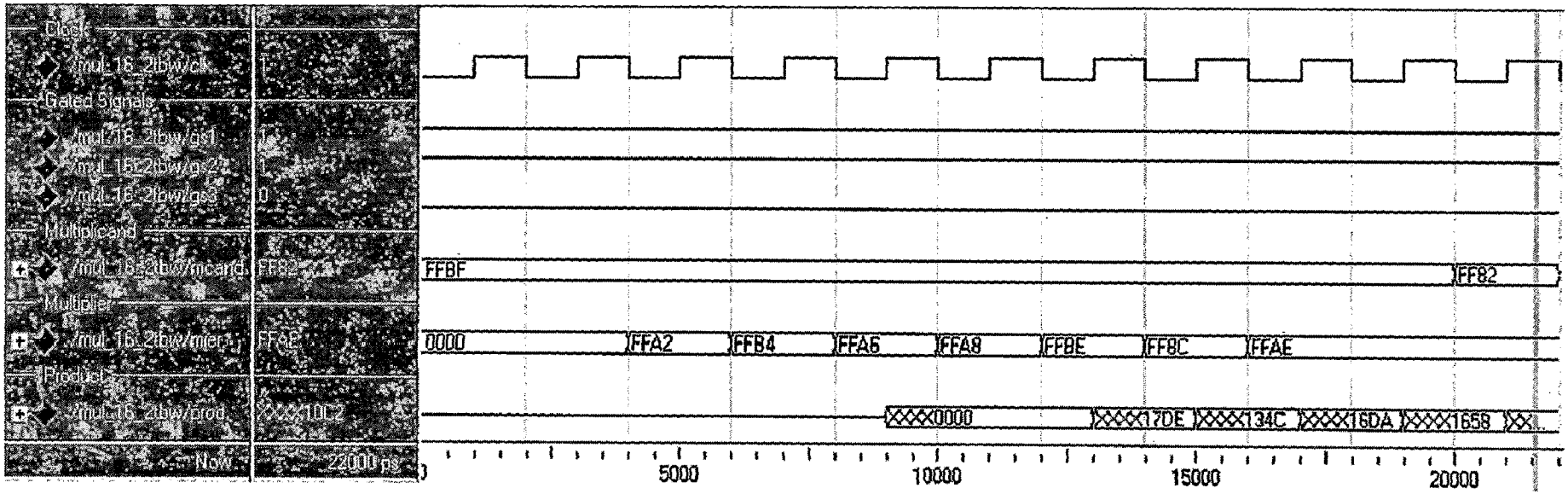


Figure 5.8 Output of multiplier with 2-dimensional clock gating after synthesis for 8-bit multiplication.

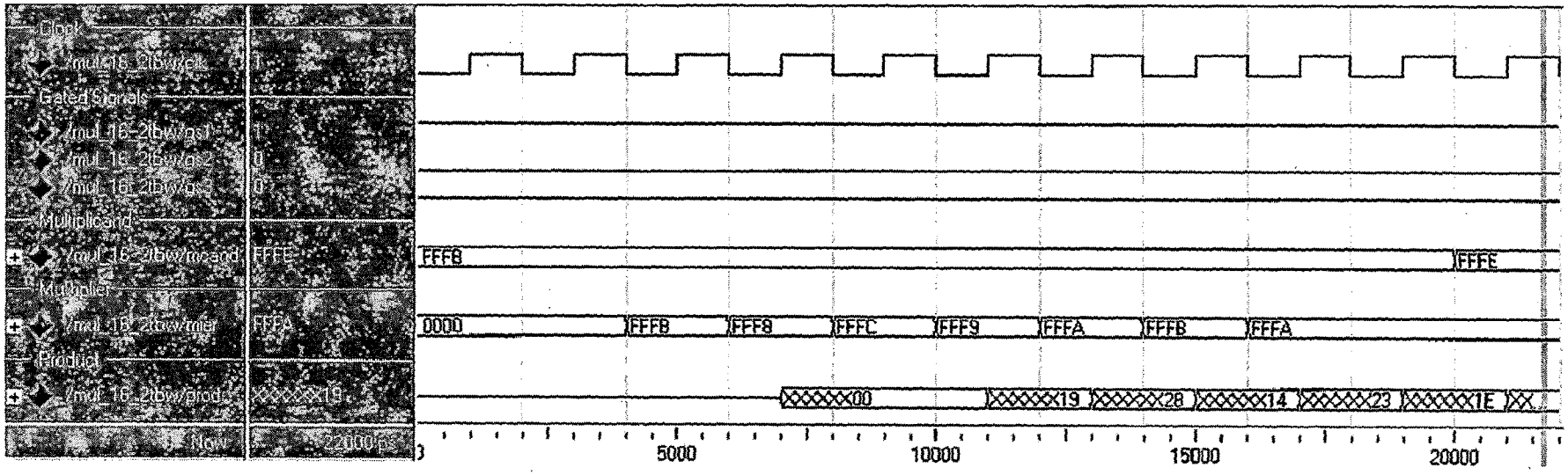


Figure 5.8 Output of multiplier with 2-dimensional clock gating after synthesis for 4-bit multiplication.

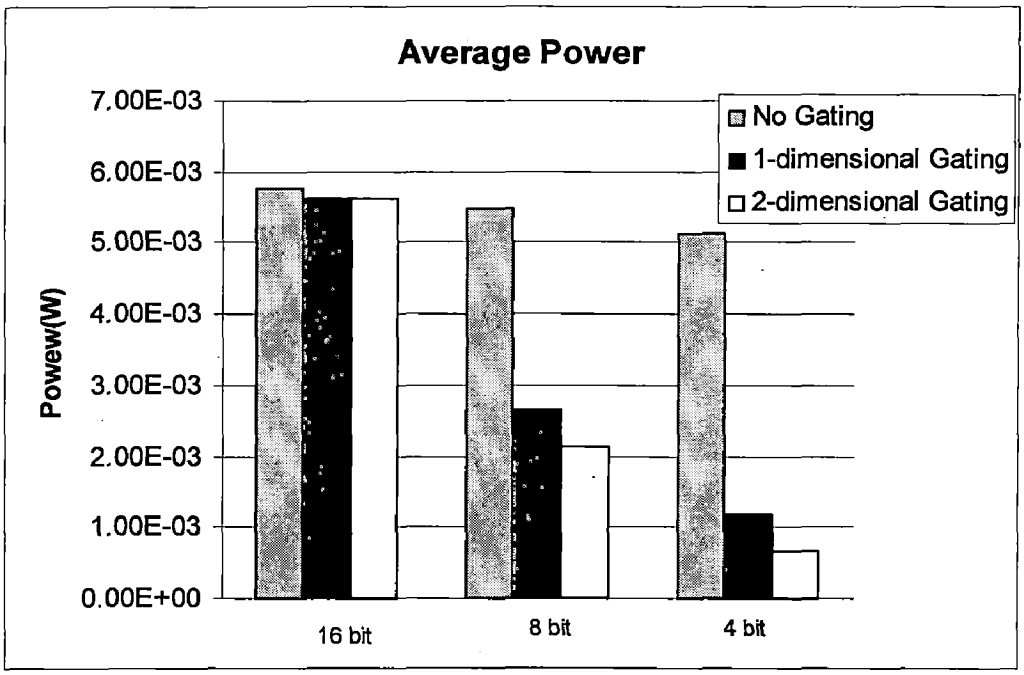


Figure 5.10. Average power dissipation of various multipliers under different input precisions (for 500MHz 6-stage pipeline)

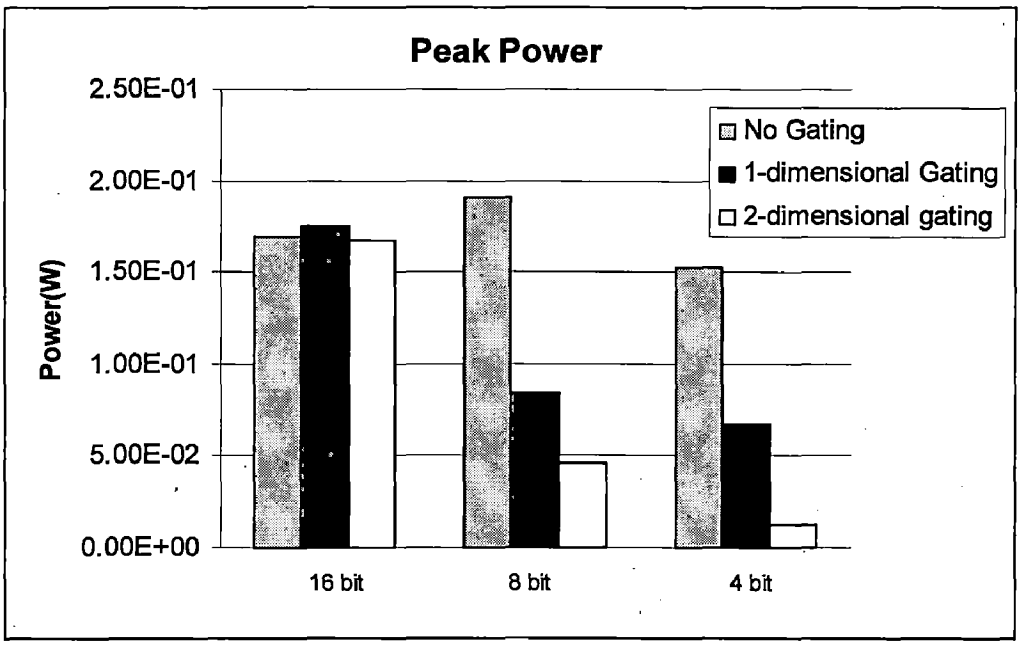


Figure 5.11. Peak power dissipation of various multipliers under different input precisions (for 500MHz 6-stage pipeline)

Area Overhead (in percentage)

1D vs 2D	0.18%
1D vs non gated	17.84%
2D vs non gated	18.06%

Chapter 6
CONCLUSION AND FUTURE SCOPE

A novel reconfigurable pipelined Booth multiplier using 2-dimensional pipeline gating scheme is proposed. This technique is to gate the clock to registers in both vertical direction (data flow direction in pipeline) and horizontal direction (within each pipeline stage). For signed multipliers using 2's complement representation, sign extension, which wastes power and causes longer delay, could be avoided by implementing this technique. Our multiplier based on the gated input signals implements a 16-bit, 8bit or 4-bit multiplication operation. The relation and difference between this 2-D technique and existing 1-D technique are discussed. A set of Booth multipliers is designed using both techniques. Simulation results show that 2-D pipeline gating technique has great advantage over 1-D technique in terms of average and peak power savings while maintaining the same latency reduction rate. 2-D and 1-D pipeline technique can be applied with some additional area.

For the 8-bit and 4-bit computations, the proposed Booth multiplier leads to a 61% and 87% power consumption reduction over a non-scalable Booth multiplier, respectively. The proposed scalable pipelined Booth multiplier proves to be globally 48% more power efficient than a non-scalable pipelined Booth multiplier, and also it has fast speed due to pipelining.

6.1 Scope of future work

Following are the suggestions for future work

1) Further increase scalability of the multiplier for lesser number of bits. This can reduce the power consumption further.

2) Use this multiplier in any of the DSP applications and verify the power savings.

3) Increase the number pipeline stages to further reduce the power consumption and increase the clock frequency. However, excessive increase in number of pipeline stages can clock frequency to a large value that a processor cannot support. Thus, it is not advised to increase pipeline stages excessively.

REFERENCES

[1] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen "Low-power CMOS digital design,"***IEEE Journal of Solid-State Circuits*** , Volume:27, Issue:4, Apr. 1992, pages 473-484.

[2] Christian Piguet, "Low-Power CMOS Circuits Technology, Logic Design and CAD Tools," ***CRC Press***, 2005.

[3] Manish Bhardwaj, R. Min, and A. P. Chandrakasan, "Quantifying and Enhancing Power Awareness of VLSI Systems," ***IEEE Transactions on VLSI Systems, 2001***, Volume 9, Issue 6, pages 757-772.

[4] S. H. Nawab, J. M. Winograd, "Approximate signal processing," ***1995 International Conference on Acoustics, Speech, and Signal Processing***, May 9-12, Pages 2857 -2860 vol.5.

[5] Z. Huang. M. D. Ercegovac, "Two-dimensional signal gating for low-power array multiplier design," ***IEEE International Symposium on Circuits and Systems,*** 2002, Volume 1, pages 489-492

[6] P. C. H. Meier, R. A. Rutenber, L. R. Carley, "Inverse polarity techniques for high-speed/low-power multipliers," ***International Symposium on Low Power Electronics and Design***, pages 264-266, 1999

[7] K. H. Lee, C. S. Rim, "A hardware reduced multiplier for low power design," ***Proceedings of the second IEEE Asia-Pacific Conference on ASICs.*** Pages 331-334, 2000

[8] S. Kim, M. C. Papaefthymiou, "Reconfigurable low energy multiplier for multimedia system design," ***Proceedings of IEEE Computer Society Workshop on VLSI,*** 2000 pages 129-134.

[9] Jia Di, J. S. Yuan and R. Demara, "High Throughput Power-aware FIR Filter Design Based On Fine-grain Pipelining Multipliers and Adders," ***Proceedings of the IEEE Computer Society Annual Symposium on VLSI,*** pages 260- 261, Feb. 2003.

[10] Jia Di and J. S. Yuan, "Run-time reconfigurable power-aware pipelined signed array multiplier design for DSP applications". ***IEEE International Symposium on Circuits and Systems,*** 2003, pages 405-408.

[11] Hanho Lee, "A Power-Aware Scalable Pipelined Booth Multiplier," ***IEEE International SOC Conference Proceedings***, sept. 2004 pages 123- 126

[12] B.Parhami, "Computer arithmetic – algorithms and hardware designs," ***Oxford University Press,*** 1999.

[13] A. D. Booth, "A signed binary multiplication technique", ***Quart. J.Math.***, vol. IV, pt. 2, 1951.

[14] 0. L MacSorley, "High-speed Arithmetic in Binary Computers," ***IRE Proc.,*** vol. 49, pages 67-91, 1961.

[15] Jan M. Rabaey,Anantha Chandrakasan and Borivoje Nikolic, "Digital Integrated Circuits –A Design Perspective," 2nd Edition, ***Prentice Hall Electronics and VLSI series***, 2005.

[16] S.Shah, A. J. Al-Khalili, D. AI- Khalili "Comparison of 32-bit Multipliers for Various Performance Measure," ***The 12th International Conference on Microelectronics***, Tehran, Oct. 31- Nov. 2, 2000.

[17] Gensuke Goto, Tomio Sato, Masao Nakajima, and Takao Sukemura, " A 54x54-b Regularly Structured Tree Multiplier", ***JSSC***, col. 27, no. 9, September 1992.

[18] A. P. Chandrakasan, and R. W. Brodersen "Minimizing Power Consumption in Digital CMOS Circuits" ***Proceedings of the IEEE***. Apr 1995, Issue: 4 pages 498-523 Vol. 83

[19] U. Narayanan, K.S. Chung, T. Kim, "Enhanced Bus Invert Encodings for Low-Power," ***ISCAS,*** pages V-25 - V-28, vol. 5,2002.

[20] Himanshu Bhatnagar, "Advanced ASIC chip synthesis using Synopsys Design Compiler, Physical Compiler and PrimeTime," ***Kluwer Academic Publishers***, 2002, Second edition.

[21] https://solvnet.synopsys.com/

[22] Niel H.E. Weste and Kamaran Eshraghian, "Principles of CMOS VLSI Design : A Systems Perspective," ***TMH,***2005.

[23] Oklobdzija, V.G., Villeger, D., Liu, S.S., "A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach," ***IEEE Transactions on Computers***, Vol. 45, No, 3, pages 294-305, March 1996.

[24] Jalil Fadavi-Ardekani, "M x N Booth Encoded Multiplier Generator Using Optimized Wallace Trees", ***IEEE Transactians on VLSI systems***,Vol.1 , No.2, pages 120-125, June 1993.

APPENDIX-A

A.1. 2-Dimensional pipeline gated Booth multiplier

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity multiplier is
   Generic (m: integer: =16; n: integer: =16);
   Port (clk: in STD_LOGIC;
         gs1, gs2, gs3: in std_logic;
         mcand: in  STD_LOGIC_VECTOR (n-1 downto 0);
         mier: in  STD_LOGIC_VECTOR (m-1 downto 0);
         prod : out  STD_LOGIC_VECTOR (m+n-1 downto 0));
end multiplier;
-------------------------ARHITECTURE OF 2D BOOTH MULTIPLIER----------------------
architecture Behavioral of multiplier is
type ary is array (0 to m/2-1) of STD_LOGIC_VECTOR ( n+1 downto 0);
type ary1 is array (0 to 5) of STD_LOGIC_VECTOR( m+n downto 1);
type ary2 is array (0 to 2) of STD_LOGIC_VECTOR( m+n downto 1);
type ary3 is array (0 to 1) of STD_LOGIC_VECTOR( m+n downto 1);
subtype word is std_logic_vector (n-1 downto 0);
signal load1, load2, load3, gclk1, gclk2, gclk3:std_logic;
signal pps, pps0: ary;
signal pps1, pps10, pps20: ary1;
signal pps2: ary2;
signal pps3: ary2;
signal pps4, pps40: ary3;
signal pps5: std_logic_vector (m+n downto 1);
-------------------------------BOOTH ENCODER AND PP GENERATOR------------------
PROCEDURE Booth_PP_gen (a: in std_logic_vector (2 downto 0); md: in
std_logic_vector; PP: out std_logic_vector; topbit: out std_logic)  is
 variable bb       : std_logic_vector (md'range);
 variable psum     : std_logic_vector (md'range);
 variable b_bar    : std_logic_vector (md'range);
 variable two_b    : std_logic_vector (md'range);
 variable two_b_bar: std_logic_vector (md'range);
 variable cin      : std_logic;
begin
                two_b:=md (md'left-1 downto 0) & '0';
                b_bar:=not md;
                two_b_bar:=not two_b;
                case a is
```

```
                when "001" | "010" =>
                                bb: = md;
                                cin : = '0';
                when "011" =>
                                bb: = two_b;
                                cin: ='0';
                when  "100" =>
                                bb:= two_b_bar;
                                cin:='1';

                when   "101" | "110" =>
                                bb:= b_bar;
                                cin:='1';
                when others =>
                                bb:=(others=>'0');
                                cin:='0';
                end case;

              case a is
                                when "001"|"010"|"011" => topbit:=not md(md'left);
                                when "100"|"101"|"110" => topbit:=md(md'left);
                                when "000"|"111"=>topbit:='1';
                                when others =>topbit:='0';
            end case;
        PP:=bb&cin;
end Booth_PP_gen;

Procedure Booth_PP_gen_after(a: in std_logic_vector(2 downto 0);md:in
std_logic_vector;PP:out std_logic_vector;fr:out  std_logic;topbit: out std_logic)is
  variable bb       : std_logic_vector (md'left-1 downto md'right);
 variable psum      : std_logic_vector (md'left-1 downto md'right);
 variable b_bar     : std_logic_vector (md'left-1 downto md'right);
 variable two_b     : std_logic_vector (md'left-1 downto md'right);
 variable two_b_bar : std_logic_vector (md'left-1 downto md'right);
begin
              two_b:=md(md'left-1 downto md'right) ;
              b_bar:=not md(md'left downto md'right+1);
              two_b_bar:=not two_b;
              case a is
                        when "001" | "010" =>
                          bb:= md(md'left downto md'right+1);
                        when "011" =>
                          bb:= two_b;
                        when  "100"  =>
                          bb:= two_b_bar;
                        when   "101" | "110" =>
```

```
                    bb:= b_bar;
                  when others =>
                    bb:=(others=>'0');
                end case;
        case a is
                  when "001"|"010"|"011" => topbit:=not md(md'left);
                  when "100"|"101"|"110" => topbit:=md(md'left);
                  when "000"|"111"=>topbit:='1';
                  when others =>topbit:='0';
        end case;
        PP:=bb(bb'left downto bb'right+1);
        fr:=bb(bb'right);
end Booth_PP_gen_after;
-------------------------------------------------------------------------------------------------
---------------------------------------- [2:2] compressor-----------------------------------
PROCEDURE CSA_ha( a:in std_logic ;b: in std_logic ;s: out std_logic; c: out std_logic)
is
begin
        s:= a xor b;
        c:= a and b;
end PROCEDURE CSA_ha;
-------------------------------------------------------------------------------------------------
---------------------------------------- [3:2] compressor-----------------------------------
PROCEDURE CSA_fa( a:in std_logic ;b: in std_logic ;d : in std_logic ;s: out std_logic;
c: out std_logic) is
begin
        s:= a xor b xor d;
        c:= (a and b) or ( a and d) or (b and d);
end PROCEDURE CSA_fa;
-------------------------------------------------------------------------------------------------
-----------------------------------------------------------RCA-----------------------------------
PROCEDURE rca ( a : in std_logic_vector; b: in std_logic_vector;sum:out
std_logic_vector) is
  variable c: std_logic_vector(a'range);
  variable nc:std_logic;
begin

  for i in a'right to a'left loop
        if i=a'right then
          CSA_fa(a(i),b(i),'0',sum(i),c(i));
        elsif i/=a'left and i/=a'right then
          CSA_fa(a(i),b(i),c(i-1),sum(i),c(i));
        else
          CSA_fa(a(i),b(i),c(i-1),sum(i),nc);
        end if;
  end loop;
```

```
end PROCEDURE rca;

-------------------BEGINNING ARHITECTURE OF 2D BOOTH MULTIPLIER ----
Begin
---------------------------------Clock Gating Logic ----------------------------------------
GATED_CLKs: process (clk,gs1, gs2, gs3)
begin
        if clk='0' then
                load1<=gs1;
                load2<=gs2;
                load3<=gs3;
        end if ;
                gclk1<= gs1 and clk;
                gclk2<= gs2 and clk;
                gclk3<= gs3 and clk;
end process GATED_CLKs;
-------------------------------------------------------------------------------------------------
------------------------------------- PIPELINE REGISTER 1 ---------------------------------
GATING_MD_MR4: process(gclk1,mcand1,mier1)
begin
if gclk1'event and gclk1='1' then
  mier(3 downto 0)<=mier1(3 downto 0);
  mcand(3 downto 0)<=mcand1(3 downto 0);
end if;
end process GATING_MD_MR4;

GATING_MD_MR8: process(gclk2,mcand1,mier1)
begin
if gclk2'event and gclk2='1' then
  mier(7 downto 4) <=mier1(7 downto 4) ;
  mcand(7 downto 4) <=mcand1(7 downto 4) ;
end if;
end process GATING_MD_MR8;

GATING_MD_MR16: process(gclk3,mcand1,mier1)
begin
if gclk3'event and gclk3='1' then
  mier(15 downto 8)<=mier1(15 downto 8);
  mcand(15 downto 8)<=mcand1(15 downto 8);
end if;
end process GATING_MD_MR16;
-------------------------------------------------------------------------------------------------
-----------------------------------------PPS generation----------------------------------------
PP_GEN: process (mier,mcand,gs1,gs2,gs3)
  variable t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12:std_logic;
  variable ppv : ary;
```

```
  variable a: std_logic_vector(2 downto 0);
  variable gck1,gck2,gck3:std_logic;
  variable mux:std_logic_vector(2 downto 0);
begin
                gck1:=gs1;
                gck2:=gs2;
                gck3:=gs3;
                ---------------------------- 4-bit -------------------------------
                a:=mier(1 downto 0) &'0';
                Booth_PP_gen(a,mcand(3 downto 0),ppv(0)(4 downto 0),t1);
                a:=mier(3 downto 1);
                Booth_PP_gen(a,mcand(3 downto 0),ppv(1)(4 downto 0),t3);
                ---------------------------- 8-bit -------------------------------
                a:=mier(1 downto 0) &'0';
                Booth_PP_gen_after(a,mcand(7 downto 3),ppv(0)(8 downto 6),t2,t5);
                a:=mier(3 downto 1);
                Booth_PP_gen_after(a,mcand(7 downto 3),ppv(1)(8 downto 6),t4,t6);
                a:=mier(5 downto 3) ;
                Booth_PP_gen(a,mcand(7 downto 0),ppv(2)(8 downto 0),t7);
                a:=mier(7 downto 5);
                Booth_PP_gen(a,mcand (7 downto 0),ppv(3)(8 downto 0),t8);
                ---------------------------- 16-bit -------------------------------
                a:=mier(1 downto 0) &'0';
                Booth_PP_gen_after(a,mcand(15 downto 7),ppv(0)(16 downto
10),t9,ppv(0)(17));
                a:=mier(3 downto 1);
                Booth_PP_gen_after(a,mcand(15 downto 7),ppv(1)(16 downto
10),t10,ppv(1)(17));
                a:=mier(5 downto 3) ;
                Booth_PP_gen_after(a,mcand(15 downto 7),ppv(2)(16 downto
10),t11,ppv(2)(17));
                a:=mier(7 downto 5);
                Booth_PP_gen_after(a,mcand (15 downto 7),ppv(3)(16 downto
10),t12,ppv(3)(17));
                a:=mier(9 downto 7) ;
                Booth_PP_gen(a,mcand,ppv(4)(16 downto 0),ppv(4)(17));
                a:=mier(11 downto 9);
                Booth_PP_gen(a,mcand,ppv(5)(16 downto 0),ppv(5)(17));
                a:=mier(13 downto 11) ;
                Booth_PP_gen(a,mcand,ppv(6)(16 downto 0),ppv(6)(17));
                a:=mier(15 downto 13);
                Booth_PP_gen(a,mcand,ppv(7)(16 downto 0),ppv(7)(17));
                ----------TOPS-----------------------------------------
        mux:=gck1&gck2&gck3;
        tops:case mux is
```

```
        when "100" =>           ppv(0)(5):=t1;
                                ppv(1)(5):=t3;
                                ppv(0)(9):='0';
                                ppv(1)(9):='0';
                                ppv(2)(9):='0';
                                ppv(3)(9):='0';

        when "110"=>            ppv(0)(5):=t2;
                                ppv(1)(5):=t4;
                                ppv(0)(9):=t5;
                                ppv(1)(9):=t6;
                                ppv(2)(9):=t7;
                                ppv(3)(9):=t8;

        when "111" =>
                                ppv(0)(5):=t2;
                                ppv(1)(5):=t4;
                                ppv(0)(9):=t9;
                                ppv(1)(9):=t10;
                                ppv(2)(9):=t11;
                                ppv(3)(9):=t12;

        when others =>          ppv(0)(5):='0';
                                ppv(1)(5):='0';
                                ppv(0)(9):='0';
                                ppv(1)(9):='0';
                                ppv(2)(9):='0';
                                ppv(3)(9):='0';


        end case tops;
        pps0<=ppv;
 end process PP_GEN;
-------------------------------------------------------------------------------------------------
------------------------------------ PIPELINE REGISTER 2 ----------------------------------
PPS0_GCLK1: process(gclk1)
begin
        if gclk1'event and gclk1='1' then
                pps(0)(5 downto 0)<=pps0(0)(5 downto 0);
                pps(1)(5 downto 0)<=pps0(1)(5 downto 0);
        end if ;
end process PPS0_GCLK1;

PPS0_GCLK2: process(gclk2)
begin
        if gclk2'event and gclk2='1' then
```

```
            pps(0)(9 downto 6)<=pps0(0)(9 downto 6);
                pps(1)(9 downto 6)<=pps0(1)(9 downto 6);
                pps(2)(9 downto 0)<=pps0(2)(9 downto 0);
                pps(3)(9 downto 0)<=pps0(3)(9 downto 0);
        end if;
end process PPS0_GCLK2;
PPS0_GCLK3: process(gclk3)
begin
        if gclk3'event and gclk3='1' then
                pps(0)(17 downto 10)<=pps0(0)(17 downto 10);
                pps(1)(17 downto 10)<=pps0(1)(17 downto 10);
                pps(2)(17 downto 10)<=pps0(2)(17 downto 10);
                pps(3)(17 downto 10)<=pps0(3)(17 downto 10);
                pps(4)<=pps0(4);
                pps(5)<=pps0(5);
                pps(6)<=pps0(6);
                pps(7)<=pps0(7);
        end if;
end process PPS0_GCLK3;
-------------------------------------------------------------------------------------------------
---------------------------------------- PP Reduction 1 -----------------------------------------
PPS_REDN1: process (pps)
variable ppv1 : ary;
variable ppv2,ppv3: ary1;
begin
         ppv1:=pps;
        ----------------------------STAGE 1-------------------------
         ppv2(1)(1):=ppv1(0)(0);
         ppv2(0)(1):=ppv1(0)(1);
         ppv2(0)(2):=ppv1(0)(2);
         u3:csa_fa(ppv1(0)(3),ppv1(1)(1),ppv1(1)(0),ppv2(0)(3),ppv2(1)(4));
         u4:csa_ha(ppv1(0)(4),ppv1(1)(2),ppv2(0)(4),ppv2(1)(5));
         u5:csa_fa(ppv1(0)(5),ppv1(1)(3),ppv1(2)(1),ppv2(0)(5),ppv2(1)(6));

         ppv2(2)(5):=ppv1(2)(0);

         u6:csa_fa(ppv1(0)(6),ppv1(1)(4),ppv1(2)(2),ppv2(0)(6),ppv2(1)(7));

         u7:csa_fa(ppv1(0)(7),ppv1(1)(5),ppv1(2)(3),ppv2(0)(7),ppv2(1)(8));
         u1_7: csa_ha(ppv1(3)(1),ppv1(3)(0),ppv2(2)(7),ppv2(3)(8));

         u8:csa_fa(ppv1(0)(8),ppv1(1)(6),ppv1(2)(4),ppv2(0)(8),ppv2(1)(9));
         ppv2(2)(8):=ppv1(3)(2);

         u9:csa_fa(ppv1(0)(9),ppv1(1)(7),ppv1(2)(5),ppv2(0)(9),ppv2(1)(10));
         ppv2(2)(9):=ppv1(3)(3);
```

```
u1_9:csa_ha(ppv1(4)(1),ppv1(4)(0),ppv2(3)(9),ppv2(3)(10));

u10:csa_fa(ppv1(3)(4),ppv1(1)(8),ppv1(2)(6),ppv2(0)(10),ppv2(1)(11));
u1_10:csa_ha(ppv1(0)(10),ppv1(4)(2),ppv2(2)(10),ppv2(3)(11));

u11:csa_fa(ppv1(3)(5),ppv1(1)(9),ppv1(2)(7),ppv2(0)(11),ppv2(1)(12));
u1_11:csa_fa(ppv1(5)(0),ppv1(4)(3),ppv1(5)(1),ppv2(2)(11),ppv2(3)(12));
ppv2(4)(11):=ppv1(0)(11);

u12:csa_ha(ppv1(2)(8),ppv1(3)(6),ppv2(0)(12),ppv2(1)(13));
u1_12:csa_ha(ppv1(0)(12),ppv1(1)(10),ppv2(2)(12),ppv2(3)(13));
u2_12:csa_ha(ppv1(4)(4),ppv1(5)(2),ppv2(4)(12),ppv2(5)(13));

u13:csa_ha(ppv1(2)(9),ppv1(3)(7),ppv2(0)(13),ppv2(1)(14));
u1_13:csa_fa(ppv1(6)(0),ppv1(6)(1),ppv1(5)(3),ppv2(2)(13),ppv2(3)(14));
u2_13:csa_fa(ppv1(0)(13),ppv1(1)(11),ppv1(4)(9),ppv2(4)(13),ppv2(5)(14));

u14:csa_fa(ppv1(0)(14),ppv1(1)(12),ppv1(2)(10),ppv2(2)(14),ppv2(3)(15));
u1_14:csa_fa(ppv1(6)(2),ppv1(4)(6),ppv1(5)(4),ppv2(4)(14),ppv2(5)(15));
ppv2(0)(14):=ppv1(3)(8);

u15:csa_fa(ppv1(0)(15),ppv1(1)(13),ppv1(2)(11),ppv2(1)(15),ppv2(2)(16));
u1_15:csa_fa(ppv1(6)(3),ppv1(4)(7),ppv1(5)(5),ppv2(2)(15),ppv2(3)(16));
u2_15:csa_ha(ppv1(7)(0),ppv1(7)(1),ppv2(4)(15),ppv2(5)(16));
ppv2(0)(15):=ppv1(3)(9);

u16:csa_fa(ppv1(0)(16),ppv1(1)(14),ppv1(2)(12),ppv2(0)(16),ppv2(1)(17));
u1_16:csa_fa(ppv1(3)(10),ppv1(4)(8),ppv1(5)(6),ppv2(1)(16),ppv2(2)(17));
u2_16:csa_ha(ppv1(6)(4),ppv1(7)(2),ppv2(4)(16),ppv2(5)(17));
u17:csa_fa(ppv1(0)(17),ppv1(1)(15),ppv1(2)(13),ppv2(0)(17),ppv2(1)(18));
u1_17:csa_fa(ppv1(3)(11),ppv1(4)(9),ppv1(5)(7),ppv2(3)(17),ppv2(4)(18));
u2_17:csa_ha(ppv1(6)(5),ppv1(7)(3),ppv2(4)(17),ppv2(5)(18));

u18:csa_fa(ppv1(1)(16),ppv1(2)(14),ppv1(3)(12),ppv2(2)(18),ppv2(3)(19));
u1_18:csa_fa(ppv1(4)(10),ppv1(5)(8),ppv1(6)(6),ppv2(3)(18),ppv2(4)(19));
ppv2(0)(18):=ppv1(7)(4);

u19:csa_fa(ppv1(1)(17),ppv1(2)(15),ppv1(3)(13),ppv2(0)(19),ppv2(1)(20));
u1_19:csa_fa(ppv1(4)(11),ppv1(5)(9),ppv1(6)(7),ppv2(1)(19),ppv2(2)(20));
ppv2(2)(19):=ppv1(7)(5);

u20:csa_fa(ppv1(2)(16),ppv1(3)(14),ppv1(4)(12),ppv2(0)(20),ppv2(1)(21));
u1_20:csa_fa(ppv1(5)(10),ppv1(6)(8),ppv1(7)(6),ppv2(3)(20),ppv2(3)(21));

u21:csa_fa(ppv1(2)(17),ppv1(3)(15),ppv1(4)(13),ppv2(0)(21),ppv2(1)(22));
u1_21:csa_fa(ppv1(5)(11),ppv1(6)(9),ppv1(7)(7),ppv2(2)(21),ppv2(3)(22));
```

```
u22:csa_fa(ppv1(3)(16),ppv1(4)(14),ppv1(5)(12),ppv2(0)(22),ppv2(1)(23));
u1_22:csa_ha(ppv1(6)(10),ppv1(7)(8),ppv2(2)(22),ppv2(3)(23));

u23:csa_fa(ppv1(3)(17),ppv1(4)(15),ppv1(5)(13),ppv2(0)(23),ppv2(1)(24));
u1_23:csa_ha(ppv1(6)(11),ppv1(7)(9),ppv2(2)(23),ppv2(2)(24));

u24:csa_fa(ppv1(4)(16),ppv1(5)(14),ppv1(6)(12),ppv2(0)(24),ppv2(1)(25));
ppv2(3)(24):=ppv1(7)(10);

u25:csa_fa(ppv1(4)(17),ppv1(5)(15),ppv1(6)(13),ppv2(0)(25),ppv2(1)(26));
ppv2(2)(25):=ppv1(7)(11);
u26:csa_fa(ppv1(5)(16),ppv1(6)(14),ppv1(7)(12),ppv2(0)(26),ppv2(1)(27));

u27:csa_fa(ppv1(5)(17),ppv1(6)(15),ppv1(7)(13),ppv2(0)(27),ppv2(1)(28));

u28:csa_ha(ppv1(6)(16),ppv1(7)(14),ppv2(0)(28),ppv2(1)(29));

u29:csa_ha(ppv1(6)(17),ppv1(7)(15),ppv2(0)(29),ppv2(1)(30));

ppv2(0)(30):= ppv1(7)(16);
ppv2(0)(31):= ppv1(7)(17);
----------STAGE 2----------
ppv3(0)(1):=ppv2(0)(1);
ppv3(0)(2):=ppv2(0)(2);
ppv3(0)(3):=ppv2(0)(3);
ppv3(1)(1):=ppv2(1)(1);
ppv3(0)(4):=ppv2(0)(4);
ppv3(1)(4):=ppv2(1)(4);
ppv3(0)(5):=ppv2(0)(5);
ppv3(1)(5):=ppv2(1)(5);
ppv3(2)(5):=ppv2(2)(5);
ppv3(0)(6):=ppv2(0)(6);
ppv3(1)(6):=ppv2(1)(6);

ppv3(0)(7):=ppv2(0)(7);
ppv3(1)(7):=ppv2(1)(7);
ppv3(2)(7):=ppv2(2)(7);

u1_0_8:csa_fa(ppv2(0)(8),ppv2(1)(8),ppv2(2)(8),ppv3(0)(8),ppv3(1)(9));
ppv3(1)(8):=ppv2(3)(8);

u1_0_9:csa_fa(ppv2(0)(9),ppv2(1)(9),ppv2(2)(9),ppv3(0)(9),ppv3(1)(10));
ppv3(2)(9):=ppv2(3)(9);

u1_0_10:csa_ha(ppv2(0)(10),ppv2(1)(10),ppv3(0)(10),ppv3(1)(11));
```

```
u1_1_10:csa_ha(ppv2(2)(10),ppv2(3)(10),ppv3(2)(10),ppv3(3)(11));

u1_0_11:csa_ha(ppv2(0)(11),ppv2(1)(11),ppv3(0)(11),ppv3(1)(12));
u1_1_11:csa_fa(ppv2(2)(11),ppv2(3)(11),ppv2(4)(11),ppv3(2)(11),ppv3(3)(12));

u1_0_12:csa_ha(ppv2(0)(12),ppv2(1)(12),ppv3(0)(12),ppv3(1)(13));
u1_1_12:csa_fa(ppv2(2)(12),ppv2(3)(12),ppv2(4)(12),ppv3(2)(12),ppv3(3)(13));

u1_0_13:csa_fa(ppv2(2)(13),ppv2(3)(13),ppv2(4)(13),ppv3(2)(13),ppv3(3)(14));
u1_1_13:csa_ha(ppv2(0)(13),ppv2(1)(13),ppv3(0)(13),ppv3(1)(14));
ppv3(4)(13):=ppv2(5)(13);

u1_0_14:csa_ha(ppv2(0)(14),ppv2(1)(14),ppv3(0)(14),ppv3(1)(15));
u1_1_14:csa_fa(ppv2(2)(14),ppv2(3)(14),ppv2(4)(14),ppv3(2)(14),ppv3(3)(15));
ppv3(4)(14):=ppv2(5)(14);

ppv3(0)(15):=ppv2(0)(15);
u1_0_15:csa_fa(ppv2(3)(15),ppv2(1)(15),ppv2(2)(15),ppv3(2)(15),ppv3(2)(16));
u1_1_15:csa_ha(ppv2(5)(15),ppv2(4)(15),ppv3(4)(15),ppv3(3)(16));

u1_0_16:csa_fa(ppv2(0)(16),ppv2(1)(16),ppv2(2)(16),ppv3(0)(16),ppv3(1)(17));
u1_1_16:csa_fa(ppv2(3)(16),ppv2(4)(16),ppv2(5)(16),ppv3(1)(16),ppv3(2)(17));

u1_0_17:csa_fa(ppv2(0)(17),ppv2(1)(17),ppv2(2)(17),ppv3(0)(17),ppv3(1)(18));
u1_1_17:csa_fa(ppv2(3)(17),ppv2(4)(17),ppv2(5)(17),ppv3(3)(17),ppv3(3)(18));

u1_0_18:csa_fa(ppv2(0)(18),ppv2(1)(18),ppv2(2)(18),ppv3(0)(18),ppv3(1)(19));
u1_1_18:csa_fa(ppv2(3)(18),ppv2(4)(18),ppv2(5)(18),ppv3(2)(18),ppv3(3)(19));

u1_0_19:csa_fa(ppv2(0)(19),ppv2(1)(19),ppv2(2)(19),ppv3(0)(19),ppv3(1)(20));
u1_1_19:csa_ha(ppv2(3)(19),ppv2(4)(19),ppv3(2)(19),ppv3(3)(20));

u1_0_20:csa_fa(ppv2(0)(20),ppv2(1)(20),ppv2(2)(20),ppv3(0)(20),ppv3(1)(21));
ppv3(2)(20):=ppv2(3)(20);

u1_0_21:csa_fa(ppv2(0)(21),ppv2(1)(21),ppv2(2)(21),ppv3(0)(21),ppv3(1)(22));
ppv3(2)(21):=ppv2(3)(21);

u1_0_22:csa_fa(ppv2(0)(22),ppv2(1)(22),ppv2(2)(22),ppv3(0)(22),ppv3(1)(23));
ppv3(2)(22):=ppv2(3)(22);

u1_0_23:csa_fa(ppv2(0)(23),ppv2(1)(23),ppv2(2)(23),ppv3(0)(23),ppv3(1)(24));
ppv3(2)(23):=ppv2(3)(23);

u1_0_24:csa_fa(ppv2(0)(24),ppv2(1)(24),ppv2(2)(24),ppv3(0)(24),ppv3(1)(25));
ppv3(2)(24):=ppv2(3)(24);
```

```
        u1_0_25:csa_fa(ppv2(0)(25),ppv2(1)(25),ppv2(2)(25),ppv3(0)(25),ppv3(1)(26));

        u1_0_26:csa_ha(ppv2(0)(26),ppv2(1)(26),ppv3(0)(26),ppv3(1)(27));
        u1_0_27:csa_ha(ppv2(0)(27),ppv2(1)(27),ppv3(0)(27),ppv3(1)(28));
        u1_0_28:csa_ha(ppv2(0)(28),ppv2(1)(28),ppv3(0)(28),ppv3(1)(29));
        u1_0_29:csa_ha(ppv2(0)(29),ppv2(1)(29),ppv3(0)(29),ppv3(1)(30));
        u1_0_30:csa_ha(ppv2(0)(30),ppv2(1)(30),ppv3(0)(30),ppv3(1)(31));

        ppv3(0)(31):=ppv2(0)(31);
        ppv3(1)(2):='0';
        ppv3(1)(3):='0';
        ppv3(2)(4 downto 1):=(others=>'0');
        ppv3(2)(6):='0';
        ppv3(2)(8):='0';
        pps10<=ppv3;
end process PPS_REDN1;
-------------------------------------------------------------------------------------------------
---------------------------------------- PIPELINE REGISTER 3 ------------------------------------
PPS1_GCLK2:process (gclk2)
begin
        if gclk2'event and gclk2='1' then
                pps1(0)(15 downto 1)<=pps10(0)(15 downto 1);
                pps1(1)(15 downto 1)<=pps10(1)(15 downto 1);
                pps1(2)(8 downto 1)<=pps10(2)(8 downto 1);
        end if;
end process PPS1_GCLK2;

PPS1_GCLK3:process (gclk3)
begin
        if gclk3'event and gclk3='1' then
                pps1(0)(31 downto 16)<=pps10(0)(31 downto 16);
                pps1(1)(31 downto 16)<=pps10(1)(31 downto 16);
                pps1(2)(24 downto 9)<=pps10(2)(24 downto 9);
                pps1(3)(20 downto 11)<=pps10(3)(20 downto 11);
                pps1(4)(15 downto 13)<=pps10(4)(15 downto 13);
        end if;
end process PPS1_GCLK3;
-------------------------------------------------------------------------------------------------
---------------------------------------- PP Reduction 2 -----------------------------------------
PPS_REDN2 : process (pps1)
 variable ppv4,ppv5,ppv6: ary1;
begin
 ppv4:=pps1;
    ----------------------------STAGE 1-----------------------
    ppv5(0)(10 downto 1):= ppv4(0)(10 downto 1);
```

```
ppv5(1)(10 downto 1):= ppv4(1)(10 downto 1);
ppv5(2)(10 downto 5):= ppv4(2)(10 downto 5);

u2_11:csa_fa(ppv4(0)(11),ppv4(1)(11),ppv4(2)(11),ppv5(0)(11),ppv5(1)(12));
ppv5(1)(11):=ppv4(3)(11);

u2_12:csa_fa(ppv4(0)(12),ppv4(1)(12),ppv4(2)(12),ppv5(0)(12),ppv5(1)(13));
ppv5(2)(12):=ppv4(3)(12);

u2_13:csa_fa(ppv4(0)(13),ppv4(1)(13),ppv4(2)(13),ppv5(0)(13),ppv5(1)(14));
u21_13:csa_ha(ppv4(3)(13),ppv4(4)(13),ppv5(2)(13),ppv5(3)(14));


u2_14:csa_fa(ppv4(0)(14),ppv4(1)(14),ppv4(2)(14),ppv5(0)(14),ppv5(1)(15));
u21_14:csa_ha(ppv4(3)(14),ppv4(4)(14),ppv5(2)(14),ppv5(3)(15));


u2_15:csa_fa(ppv4(0)(15),ppv4(1)(15),ppv4(2)(15),ppv5(0)(15),ppv5(1)(16));
u22_15:csa_ha(ppv4(3)(15),ppv4(4)(15),ppv5(2)(15),ppv5(3)(16));

u2_16:csa_fa(ppv4(0)(16),ppv4(1)(16),ppv4(2)(16),ppv5(0)(16),ppv5(1)(17));
ppv5(2)(16):=ppv4(3)(16);

u2_17:csa_fa(ppv4(0)(17),ppv4(1)(17),ppv4(2)(17),ppv5(0)(17),ppv5(1)(18));
ppv5(2)(17):=ppv4(3)(17);

u2_18:csa_fa(ppv4(0)(18),ppv4(1)(18),ppv4(2)(18),ppv5(0)(18),ppv5(1)(19));
ppv5(2)(18):=ppv4(3)(18);

u2_19:csa_fa(ppv4(0)(19),ppv4(1)(19),ppv4(2)(19),ppv5(0)(19),ppv5(1)(20));
ppv5(2)(19):=ppv4(3)(19);

u2_20:csa_fa(ppv4(0)(20),ppv4(1)(20),ppv4(2)(20),ppv5(0)(20),ppv5(1)(21));
ppv5(2)(20):=ppv4(3)(20);

u2_21:csa_fa(ppv4(0)(21),ppv4(1)(21),ppv4(2)(21),ppv5(0)(21),ppv5(1)(22));
u2_22:csa_fa(ppv4(0)(22),ppv4(1)(22),ppv4(2)(22),ppv5(0)(22),ppv5(1)(23));
u2_23:csa_fa(ppv4(0)(23),ppv4(1)(23),ppv4(2)(23),ppv5(0)(23),ppv5(1)(24));
u2_24:csa_fa(ppv4(0)(24),ppv4(1)(24),ppv4(2)(24),ppv5(0)(24),ppv5(1)(25));
u2_25:csa_ha(ppv4(0)(25),ppv4(1)(25),ppv5(0)(25),ppv5(1)(26));
u2_26:csa_ha(ppv4(0)(26),ppv4(1)(26),ppv5(0)(26),ppv5(1)(27));
u2_27:csa_ha(ppv4(0)(27),ppv4(1)(27),ppv5(0)(27),ppv5(1)(28));
u2_28:csa_ha(ppv4(0)(28),ppv4(1)(28),ppv5(0)(28),ppv5(1)(29));
u2_29:csa_ha(ppv4(0)(29),ppv4(1)(29),ppv5(0)(29),ppv5(1)(30));
u2_30:csa_ha(ppv4(0)(30),ppv4(1)(30),ppv5(0)(30),ppv5(1)(31));
u2_31:csa_ha(ppv4(0)(31),ppv4(1)(31),ppv5(0)(31),ppv5(0)(32));
```

```
    ppv5(2)(11):='0';
    ----------------------------------STAGE 2-------------------------

    ppv6(0)(13 downto 1):= ppv5(0)(13 downto 1);
    ppv6(1)(13 downto 1):= ppv5(1)(13 downto 1);
    ppv6(2)(13 downto 5):= ppv5(2)(13 downto 5);

   u2_1_14:csa_fa(ppv5(0)(14),ppv5(1)(14),ppv5(2)(14),ppv6(0)(14),ppv6(1)(15));
   ppv6(1)(14):=ppv5(3)(14);
   u2_1_15:csa_fa(ppv5(0)(15),ppv5(1)(15),ppv5(2)(15),ppv6(0)(15),ppv6(1)(16));
   ppv6(2)(15):=ppv5(3)(15);

   u2_1_16:csa_fa(ppv5(0)(16),ppv5(1)(16),ppv5(2)(16),ppv6(0)(16),ppv6(1)(17));
   ppv6(2)(16):=ppv5(3)(16);

  u2_1_17:csa_fa(ppv5(0)(17),ppv5(1)(17),ppv5(2)(17),ppv6(0)(17),ppv6(1)(18));
  u2_1_18:csa_fa(ppv5(0)(18),ppv5(1)(18),ppv5(2)(18),ppv6(0)(18),ppv6(1)(19));
  u2_1_19:csa_fa(ppv5(0)(19),ppv5(1)(19),ppv5(2)(19),ppv6(0)(19),ppv6(1)(20));
  u2_1_20:csa_fa(ppv5(0)(20),ppv5(1)(20),ppv5(2)(20),ppv6(0)(20),ppv6(1)(21));
  u2_1_21:csa_ha(ppv5(0)(21),ppv5(1)(21),ppv6(0)(21),ppv6(1)(22));
  u2_1_22:csa_ha(ppv5(0)(22),ppv5(1)(22),ppv6(0)(22),ppv6(1)(23));
  u2_1_23:csa_ha(ppv5(0)(23),ppv5(1)(23),ppv6(0)(23),ppv6(1)(24));
  u2_1_24:csa_ha(ppv5(0)(24),ppv5(1)(24),ppv6(0)(24),ppv6(1)(25));
  u2_1_25:csa_ha(ppv5(0)(25),ppv5(1)(25),ppv6(0)(25),ppv6(1)(26));
  u2_1_26:csa_ha(ppv5(0)(26),ppv5(1)(26),ppv6(0)(26),ppv6(1)(27));
  u2_1_27:csa_ha(ppv5(0)(27),ppv5(1)(27),ppv6(0)(27),ppv6(1)(28));
  u2_1_28:csa_ha(ppv5(0)(28),ppv5(1)(28),ppv6(0)(28),ppv6(1)(29));
  u2_1_29:csa_ha(ppv5(0)(29),ppv5(1)(29),ppv6(0)(29),ppv6(1)(30));
  u2_1_30:csa_ha(ppv5(0)(30),ppv5(1)(30),ppv6(0)(30),ppv6(1)(31));
  u2_1_31:csa_ha(ppv5(0)(31),ppv5(1)(31),ppv6(0)(31),ppv6(1)(32));

  ppv6(0)(32):=ppv5(0)(32);
  ppv6(2)(14):='0';
  pps20<=ppv6;
end process PPS_REDN2;
-------------------------------------------------------------------------------------------------
---------------------------------------- PIPELINE REGISTER 4 ------------------------------------
PPS2_GCLK3: process(gclk3)
begin
        if gclk3'event and gclk3='1' then
                pps2(0)(32 downto 1)<=pps20(0)(32 downto 1);
                pps2(1)(32 downto 1)<=pps20(1)(32 downto 1);
                pps2(2)(16 downto 5)<= pps20(2)(16 downto 5);
        end if;
end process PPS2_GCLK3;
```

```
---------------------------------------- MUX Process ----------------------------------------
 MUX_process: process(pps1,pps2,pps,gs1,gs2,gs3)
    variable m:std_logic_vector(2 downto 0);
    variable cv1: std_logic_vector (3 downto 0);
    variable cv2: std_logic_vector (7 downto 0);
    variable cv3: std_logic_vector (15 downto 0);
   variable ci1: integer range -2**(3) to ( 2**(3))-1;
   variable ci2: integer range -2**(7) to ( 2**(7))-1;
   variable ci3: integer range -2**(15) to ( 2**(15))-1;
begin
m:=gs1&gs2&gs3;
ci1:= -(((2**4)-1)/3);
cv1 := std_logic_vector(conv_unsigned(ci1,4));
ci2:= -(((2**8)-1)/3);
cv2 := std_logic_vector(conv_unsigned(ci2,8));
ci3:= -(((2**16)-1)/3);
cv3 := std_logic_vector(conv_unsigned(ci3,16));

        gating:case m is
                when "111"=>
                                        pps3(0)(32 downto 1)<=pps2(0)(32 downto 1);
                                        pps3(1)(32 downto 1)<=pps2(1)(32 downto 1);
                                        pps3(2)(4 downto 1)<=(others=>'0');
                                        pps3(2)(16 downto 5)<= pps2(2)(16 downto 5);
                                        pps3(2)(32 downto 17)<= cv3;
                when "110"=>
                                        pps3(0)(15 downto 1)<=pps1(0)(15 downto 1);
                                        pps3(0)(32 downto 16)<=(others=>'0');
                                        pps3(1)(15 downto 1)<=pps1(1)(15 downto 1);
                                        pps3(1)(32 downto 16)<=(others=>'0');
                                        pps3(2)(4 downto 1)<=(others=>'0');
                                        pps3(2)(8 downto 5)<=pps1(2)(8 downto 5);
                                        pps3(2)(16 downto 9)<=cv2;
                                        pps3(2)( 32 downto 17)<=(others=>'0');
                when "100"  =>
                                        pps3(0)(5 downto 1)<=pps(0)(5 downto 1);
                                        pps3(0)(32 downto 6)<=(others=>'0');
                                        pps3(1)(1)<=pps(0)(0);
                                        pps3(1)(2)<='0';
                                        pps3(1)(7 downto 3)<=pps(1)(5 downto 1);
                                        pps3(1)(32 downto 8)<=(others=>'0');
                                        pps3(2)( 2 downto 1)<=(others=>'0');
                                        pps3(2)(3)<=pps(1)(0);
                                        pps3(2)(4)<='0';
                                        pps3(2)(8 downto 5)<=cv1;
                                        pps3(2)(32 downto 9)<=(others=>'0');
```

```
            when others =>
                                        for i in 0 to 2 loop
                                            pps3(i)<=(others=>'0');
                                        end loop;
        end case gating;

end process MUX_process;
-------------------------------------------------------------------------------------------------
---------------------------------------- PP Reduction 3 -----------------------------------------
PPS_REDN_final : process(pps3)

  variable ppv7:ary2;
  variable ppv8:ary3;
  variable nc1:std_logic;
begin

        ppv7:=pps3;
        ppv8(0)(2 downto 1):= ppv7(0)(2 downto 1);
        ppv8(1)(2 downto 1):= ppv7(1)(2 downto 1);
        ppv8(1)(3):='0';
        u_F_3:csa_fa(ppv7(0)(3),ppv7(1)(3),ppv7(2)(3),ppv8(0)(3),ppv8(1)(4));
        u_F_4:csa_ha(ppv7(0)(4),ppv7(1)(4),ppv8(0)(4),ppv8(1)(5));
        u_F_5:csa_fa(ppv7(0)(5),ppv7(1)(5),ppv7(2)(5),ppv8(0)(5),ppv8(1)(6));
        u_F_6:csa_fa(ppv7(0)(6),ppv7(1)(6),ppv7(2)(6),ppv8(0)(6),ppv8(1)(7));
        u_F_7:csa_fa(ppv7(0)(7),ppv7(1)(7),ppv7(2)(7),ppv8(0)(7),ppv8(1)(8));
        u_F_8:csa_fa(ppv7(0)(8),ppv7(1)(8),ppv7(2)(8),ppv8(0)(8),ppv8(1)(9));
        u_F_9:csa_fa(ppv7(0)(9),ppv7(1)(9),ppv7(2)(9),ppv8(0)(9),ppv8(1)(10));
        u_F_10:csa_fa(ppv7(0)(10),ppv7(1)(10),ppv7(2)(10),ppv8(0)(10),ppv8(1)(11));
        u_F_11:csa_fa(ppv7(0)(11),ppv7(1)(11),ppv7(2)(11),ppv8(0)(11),ppv8(1)(12));
         u_F_12:csa_fa(ppv7(0)(12),ppv7(1)(12),ppv7(2)(12),ppv8(0)(12),ppv8(1)(13));
        u_F_13:csa_fa(ppv7(0)(13),ppv7(1)(13),ppv7(2)(13),ppv8(0)(13),ppv8(1)(14));
        u_F_14:csa_fa(ppv7(0)(14),ppv7(1)(14),ppv7(2)(14),ppv8(0)(14),ppv8(1)(15));
        u_F_15:csa_fa(ppv7(0)(15),ppv7(1)(15),ppv7(2)(15),ppv8(0)(15),ppv8(1)(16));
        u_F_16:csa_fa(ppv7(0)(16),ppv7(1)(16),ppv7(2)(16),ppv8(0)(16),ppv8(1)(17));
        u_F_17:csa_fa(ppv7(0)(17),ppv7(1)(17),ppv7(2)(17),ppv8(0)(17),ppv8(1)(18));
        u_F_18:csa_fa(ppv7(0)(18),ppv7(1)(18),ppv7(2)(18),ppv8(0)(18),ppv8(1)(19));
        u_F_19:csa_fa(ppv7(0)(19),ppv7(1)(19),ppv7(2)(19),ppv8(0)(19),ppv8(1)(20));
        u_F_20:csa_fa(ppv7(0)(20),ppv7(1)(20),ppv7(2)(20),ppv8(0)(20),ppv8(1)(21));
        u_F_21:csa_fa(ppv7(0)(21),ppv7(1)(21),ppv7(2)(21),ppv8(0)(21),ppv8(1)(22));
        u_F_22:csa_fa(ppv7(0)(22),ppv7(1)(22),ppv7(2)(22),ppv8(0)(22),ppv8(1)(23));
        u_F_23:csa_fa(ppv7(0)(23),ppv7(1)(23),ppv7(2)(23),ppv8(0)(23),ppv8(1)(24));
        u_F_24:csa_fa(ppv7(0)(24),ppv7(1)(24),ppv7(2)(24),ppv8(0)(24),ppv8(1)(25));
        u_F_25:csa_fa(ppv7(0)(25),ppv7(1)(25),ppv7(2)(25),ppv8(0)(25),ppv8(1)(26));
        u_F_26:csa_fa(ppv7(0)(26),ppv7(1)(26),ppv7(2)(26),ppv8(0)(26),ppv8(1)(27));
        u_F_27:csa_fa(ppv7(0)(27),ppv7(1)(27),ppv7(2)(27),ppv8(0)(27),ppv8(1)(28));
        u_F_28:csa_fa(ppv7(0)(28),ppv7(1)(28),ppv7(2)(28),ppv8(0)(28),ppv8(1)(29));
```

```
        u_F_29:csa_fa(ppv7(0)(29),ppv7(1)(29),ppv7(2)(29),ppv8(0)(29),ppv8(1)(30));
        u_F_30:csa_fa(ppv7(0)(30),ppv7(1)(30),ppv7(2)(30),ppv8(0)(30),ppv8(1)(31));
        u_F_31:csa_fa(ppv7(0)(31),ppv7(1)(31),ppv7(2)(31),ppv8(0)(31),ppv8(1)(32));
        u_F_32:csa_fa(ppv7(0)(32),ppv7(1)(32),ppv7(2)(32),ppv8(0)(32),nc1);
  pps40<=ppv8;
end process PPS_REDN_final;
-------------------------------------------------------------------------------------------------
---------------------------------------- PIPELINE REGISTER 5 ------------------------------------
PPSf_GCLK1: process(gclk1)
begin
        if gclk1'event and gclk1='1' then
         pps4(0)(8 downto 1)<=pps40(0)(8 downto 1);
         pps4(1)(8 downto 1)<=pps40(1)(8 downto 1);
        end if;
end process PPSf_GCLK1;

PPSf_GCLK2: process(gclk2)
begin
        if gclk2'event and gclk2='1' then
          pps4(0)(16 downto 9)<=pps40(0)(16 downto 9);
          pps4(1)(16 downto 9)<=pps40(1)(16 downto 9);
        end if;
end process PPSf_GCLK2;

PPSf_GCLK3: process(gclk3)
begin
        if gclk3'event and gclk3='1' then
           pps4(0)(32 downto 17)<=pps40(0)(32 downto 17);
           pps4(1)(32 downto 17)<=pps40(1)(32 downto 17);
        end if;
end process PPSf_GCLK3;
-------------------------------------------------------------------------------------------------
----------------------------------------------RCA------------------------------------------------
RIPPLE:process (pps4)
variable ppv9:ary3;
variable ppv10: std_logic_vector(m+n downto 1);
begin
        ppv9:=pps4;
        rca(ppv9(0),ppv9(1),ppv10);
        pps5<=ppv10;
end process RIPPLE;
-------------------------------------------------------------------------------------------------
---------------------------------------- PIPELINE REGISTER 6 ------------------------------------
PROD_GCLK1:process(gclk1)
begin
        if gclk1'event and gclk1='1' then
```

```
        prod(7 downto 0)<=pps5(8 downto 1);
        end if;
end process PROD_GCLK1;

PROD_GCLK2:process(gclk2)
begin
        if gclk2'event and gclk2='1' then
         prod(16 downto 8)<=pps5(17 downto 9);
        end if;

end process PROD_GCLK2;

PROD_GCLK3:process(gclk3)
begin
        if gclk3'event and gclk3='1' then
                prod(31 downto 17)<=pps5(32 downto 18);
        end if;
end process PROD_GCLK3;
-------------------------------------------------------------------------------------------------
end Behavioral;
--------------------------- END OF 2D-MULTIPLIER ARCHITECTURE----------------------
```

APPENDIX-B

B. 1-Dimensional pipeline gated Booth multiplier

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity multiplier_1d is
   generic (m:integer :=16; n: integer:=16);
   Port ( mier1 : in  STD_LOGIC_VECTOR (m-1 downto 0);
          mcand1 : in  STD_LOGIC_VECTOR (n-1 downto 0);
          clk : in  STD_LOGIC;
          gs1 : in  STD_LOGIC;
          gs2 : in  STD_LOGIC;
          gs3 : in  STD_LOGIC;
          prod : out  STD_LOGIC_VECTOR (m+n-1 downto 0));
end multiplier_1d;
-------------------------ARHITECTURE OF 1D BOOTH MULTIPLIER----------------------
architecture Behavioral of multiplier_1d is

type ary is array(0 to m/2-1) of STD_LOGIC_VECTOR( n+1 downto 0);
type ary1 is array(0 to 6) of STD_LOGIC_VECTOR( m+n downto 1);
type ary2 is array(0 to 2) of STD_LOGIC_VECTOR( m+n downto 1);
type ary3 is array(0 to 1) of STD_LOGIC_VECTOR( m+n downto 1);
subtype word is std_logic_vector(n-1 downto 0);

signal load1,load2,load3,gclk1,gclk2,gclk3:std_logic;
signal mier:STD_LOGIC_VECTOR (m-1 downto 0);
signal mcand:STD_LOGIC_VECTOR (n-1 downto 0);
signal pps,pps0 : ary;
signal pps1,pps10,pps20: ary1;
signal pps2: ary2;
signal pps3: ary2;
signal pps4,pps40:ary3;
signal pps5: std_logic_vector(m+n downto 1);
------------------------------BOOTH ENCODER AND PP GENERATOR------------------
PROCEDURE Booth_PP_gen(a:in  std_logic_vector(2 downto 0);md: in
std_logic_vector;PP:out std_logic_vector)  is
  variable bb       : std_logic_vector (md'range);
  variable psum     : std_logic_vector (md'range);
  variable b_bar    : std_logic_vector (md'range);
  variable two_b    : std_logic_vector (md'range);
  variable two_b_bar : std_logic_vector (md'range);
  variable cin      : std_logic;
```

```
 variable topbit    : std_logic;
begin
                two_b:=md(md'left-1 downto 0) & '0';
                b_bar:=not md;
                two_b_bar:=not two_b;

                case a is
                 when "001" | "010" =>      bb:= md;
                                            cin:='0';
                 when "011" =>              bb:= two_b;
                                            cin:='0';
                 when  "100"  =>            bb:= two_b_bar;
                                            cin:='1';
                 when   "101" | "110" =>    bb:= b_bar;
                                            cin:='1';
                  when others =>            bb:=(others=>'0');
                                            cin:='0';
                end case;

                top: case a is
                      when "001"|"010"|"011" => topbit:=not md(md'left);
                      when "100"|"101"|"110" => topbit:=md(md'left);
                      when "000"|"111"=>topbit:='1';
                      when others =>topbit:='0';
                end case top;

                PP:=topbit&bb&cin;
end Booth_PP_gen;
-------------------------------------------------------------------------------------------------
---------------------------------------- [2:2] compressor-----------------------------------
PROCEDURE CSA_ha( a:in std_logic ;b: in std_logic;s: out std_logic;c: out std_logic) is
begin
s:= a xor b;
c:= a and b;
end PROCEDURE CSA_ha;
-------------------------------------------------------------------------------------------------
---------------------------------------- [3:2] compressor-----------------------------------
PROCEDURE CSA_fa( a:in std_logic ;b: in std_logic ;d : in std_logic ;s: out std_logic;
c: out std_logic) is
begin
  s:= a xor b xor d;
  c:= (a and b) or ( a and d) or (b and d);
end PROCEDURE CSA_fa;
-------------------------------------------------------------------------------------------------
-----------------------------------------RCA-----------------------------------------
```

```
PROCEDURE rca ( a : in std_logic_vector; b: in std_logic_vector;sum:out
std_logic_vector) is
variable c: std_logic_vector(a'range);
variable nc:std_logic;
begin
for i in a'right to a'left loop
 if i=a'right then
   CSA_fa(a(i),b(i),'0',sum(i),c(i));
 elsif i/=a'left then
   CSA_fa(a(i),b(i),c(i-1),sum(i),c(i));
 else
 CSA_fa(a(i),b(i),c(i-1),sum(i),nc);
 end if;
end loop;
end PROCEDURE rca;
----------BEGINNING ARHITECTURE OF 1D BOOTH MULTIPLIER ----------
Begin
----------Clock Gating Logic ----------
GATED_CLKs: process (clk,gs1, gs2, gs3,load1,load2,load3)
begin
        if clk='0' then
                load1<=gs1;
                load2<=gs2;
                load3<=gs3;
        end if ;
                gclk1<= load1 and clk;
                gclk2<= load2 and clk;
                gclk3<= load3 and clk;
end process GATED_CLKs;
----------
---------- PIPELINE REGISTER 1 ----------
PIPELINING_MD: process(clk,mcand1)
begin
if clk'event and clk='1' then
    mcand<=mcand1;
end if;
end process PIPELINING_MD;

PIPELINING_MR1: process(gclk1,mier1)
begin
if gclk1'event and gclk1='1' then
    mier(3 downto 0)<=mier1(3 downto 0);
end if;
end process PIPELINING_MR1;

PIPELINING_MR2: process(gclk2,mier1)
```

```
begin
if gclk2'event and gclk2='1' then
    mier(7 downto 4)<=mier1(7 downto 4);
end if;
end process PIPELINING_MR2;

PIPELINING_MR3: process(gclk3,mier1)
begin
if gclk3'event and gclk3='1' then
    mier(15 downto 8)<=mier1(15 downto 8);
end if;
end process PIPELINING_MR3;
-------------------------------------------------------------------------------------------------
------------------------------------- PP Generation ---------------------------------------
PP_GEN:process (mier,mcand)
  variable ppv : ary;
  variable a: std_logic_vector(2 downto 0);
  variable mux:std_logic_vector(2 downto 0);
 begin
                ---------------        4- bits ------------------------------
                a:=mier(1 downto 0) &'0';
                Booth_PP_gen(a,mcand,ppv(0));
                a:=mier(3 downto 1);
                Booth_PP_gen(a,mcand,ppv(1));
                ---------------        8- bits ------------------------------
                a:=mier(5 downto 3);
                Booth_PP_gen(a,mcand,ppv(2));
                a:=mier(7 downto 5);
                Booth_PP_gen(a,mcand ,ppv(3));
                ------------------     16-bits ------------------------------
                a:=mier(9 downto 7) ;
                Booth_PP_gen(a,mcand,ppv(4));
                a:=mier(11 downto 9);
                Booth_PP_gen(a,mcand,ppv(5));
                a:=mier(13 downto 11) ;
                Booth_PP_gen(a,mcand,ppv(6));
                a:=mier(15 downto 13);
                Booth_PP_gen(a,mcand,ppv(7));
                -------------TOPS-----------------------------------------
        pps0<=ppv;
 end process PP_GEN;
-------------------------------------------------------------------------------------------------
------------------------------------- PIPELINE REGISTER 2 ---------------------------------
PPS0_GCLK1: process(gclk1,pps0)
begin
        if gclk1'event and gclk1='1' then
```

```
                pps(0)<=pps0(0);
                pps(1)<=pps0(1);
        end if;
end process PPS0_GCLK1;

PPS0_GCLK2: process(gclk2,pps0)
begin
        if gclk2'event and gclk2='1' then
                pps(2)<=pps0(2);
                pps(3)<=pps0(3);
        end if;
end process PPS0_GCLK2;

PPS0_GCLK3: process(gclk3,pps0)
begin
        if gclk3'event and gclk3='1' then
                pps(4)<=pps0(4);
                pps(5)<=pps0(5);
                pps(6)<=pps0(6);
                pps(7)<=pps0(7);
        end if;
end process PPS0_GCLK3;
-------------------------------------------------------------------------------------------------
---------------------------------------- PP Reduction 1 -----------------------------------------
PPS_REDN1: process (pps)
variable ppv1 : ary;
variable ppv2,ppv3: ary1;
begin
        ppv1:=pps;
        ----------------------------STAGE 1-------------------------
        ppv2(1)(1):=ppv1(0)(0);
        ppv2(0)(4 downto 1):=ppv1(0)(4 downto 1);
        ppv2(1)(4 downto 3) := ppv1(1)(2 downto 1);
        ppv2(2)(3):=ppv1(1)(0);

        u5:csa_fa(ppv1(0)(5),ppv1(1)(3),ppv1(2)(1),ppv2(0)(5),ppv2(1)(6));
        ppv2(1)(5):=ppv1(2)(0);

        u6:csa_fa(ppv1(0)(6),ppv1(1)(4),ppv1(2)(2),ppv2(0)(6),ppv2(1)(7));
        u7:csa_fa(ppv1(0)(7),ppv1(1)(5),ppv1(2)(3),ppv2(0)(7),ppv2(1)(8));
        u1_7: csa_ha(ppv1(3)(1),ppv1(3)(0),ppv2(2)(7),ppv2(3)(8));

        u8:csa_fa(ppv1(0)(8),ppv1(1)(6),ppv1(2)(4),ppv2(0)(8),ppv2(1)(9));
        ppv2(2)(8):=ppv1(3)(2);

        u9:csa_fa(ppv1(0)(9),ppv1(1)(7),ppv1(2)(5),ppv2(0)(9),ppv2(1)(10));
```

```
ppv2(2)(9):=ppv1(3)(3);
u1_9:csa_ha(ppv1(4)(1),ppv1(4)(0),ppv2(3)(9),ppv2(4)(10));

u10:csa_fa(ppv1(0)(10),ppv1(1)(8),ppv1(2)(6),ppv2(0)(10),ppv2(1)(11));
ppv2(2)(10):=ppv1(3)(4);
ppv2(3)(10):=ppv1(4)(2);

u11:csa_fa(ppv1(0)(11),ppv1(1)(9),ppv1(2)(7),ppv2(0)(11),ppv2(1)(12));
u1_11:csa_fa(ppv1(5)(0),ppv1(4)(3),ppv1(5)(1),ppv2(3)(11),ppv2(4)(12));
ppv2(2)(11):=ppv1(3)(5);

u12:csa_ha(ppv1(5)(2),ppv1(4)(4),ppv2(3)(12),ppv2(4)(13));
u1_12:csa_fa(ppv1(0)(12),ppv1(1)(10),ppv1(2)(8),ppv2(0)(12),ppv2(1)(13));
ppv2(2)(12):=ppv1(3)(6);

u13:csa_fa(ppv1(2)(9),ppv1(1)(11),ppv1(0)(13),ppv2(0)(13),ppv2(1)(14));
u1_13:csa_fa(ppv1(4)(5),ppv1(6)(1),ppv1(5)(3),ppv2(3)(13),ppv2(4)(14));
ppv2(2)(13):=ppv1(3)(7);
ppv2(5)(13):=ppv1(6)(0);
u14:csa_fa(ppv1(0)(14),ppv1(1)(12),ppv1(2)(10),ppv2(0)(14),ppv2(1)(15));
u1_14:csa_fa(ppv1(6)(2),ppv1(4)(6),ppv1(5)(4),ppv2(3)(14),ppv2(4)(15));
ppv2(2)(14):=ppv1(3)(8);

u15:csa_fa(ppv1(0)(15),ppv1(1)(13),ppv1(2)(11),ppv2(0)(15),ppv2(1)(16));
u1_15:csa_fa(ppv1(6)(3),ppv1(4)(7),ppv1(5)(5),ppv2(3)(15),ppv2(4)(16));
u2_15:csa_ha(ppv1(7)(0),ppv1(7)(1),ppv2(5)(15),ppv2(6)(16));
ppv2(2)(15):=ppv1(3)(9);

u16:csa_fa(ppv1(0)(16),ppv1(1)(14),ppv1(2)(12),ppv2(0)(16),ppv2(1)(17));
u1_16:csa_fa(ppv1(6)(4),ppv1(4)(8),ppv1(5)(6),ppv2(3)(16),ppv2(4)(17));
ppv2(2)(16):=ppv1(3)(10);
ppv2(5)(16):=ppv1(7)(2);

u17:csa_fa(ppv1(0)(17),ppv1(1)(15),ppv1(2)(13),ppv2(0)(17),ppv2(1)(18));
u1_17:csa_fa(ppv1(6)(5),ppv1(4)(9),ppv1(5)(7),ppv2(3)(17),ppv2(3)(18));
ppv2(2)(17):=ppv1(3)(11);
ppv2(5)(17):=ppv1(7)(3);

u18:csa_fa(ppv1(1)(16),ppv1(2)(14),ppv1(3)(12),ppv2(0)(18),ppv2(1)(19));
u1_18:csa_fa(ppv1(4)(10),ppv1(5)(8),ppv1(6)(6),ppv2(2)(18),ppv2(3)(19));
ppv2(4)(18):=ppv1(7)(4);

u19:csa_fa(ppv1(1)(17),ppv1(2)(15),ppv1(3)(13),ppv2(0)(19),ppv2(1)(20));
u1_19:csa_fa(ppv1(4)(11),ppv1(5)(9),ppv1(6)(7),ppv2(2)(19),ppv2(3)(20));
ppv2(4)(19):=ppv1(7)(5);
```

```
u20:csa_ha(ppv1(2)(16),ppv1(3)(14),ppv2(0)(20),ppv2(1)(21));
u1_20:csa_fa(ppv1(5)(10),ppv1(6)(8),ppv1(4)(12),ppv2(2)(20),ppv2(3)(21));
ppv2(4)(20):=ppv1(7)(6);

u21:csa_ha(ppv1(2)(17),ppv1(3)(15),ppv2(0)(21),ppv2(1)(22));
u1_21:csa_fa(ppv1(5)(11),ppv1(6)(9),ppv1(4)(13),ppv2(2)(21),ppv2(3)(22));
ppv2(4)(21):=ppv1(7)(7);

ppv2(0)(22):=ppv1(3)(16);
u22:csa_fa(ppv1(6)(10),ppv1(4)(14),ppv1(5)(12),ppv2(2)(22),ppv2(3)(23));
ppv2(4)(22):=ppv1(7)(8);

ppv2(0)(23):=ppv1(3)(17);
u23:csa_fa(ppv1(6)(11),ppv1(4)(15),ppv1(5)(13),ppv2(1)(23),ppv2(2)(24));
ppv2(2)(23):=ppv1(7)(9);

u24:csa_fa(ppv1(4)(16),ppv1(5)(14),ppv1(6)(12),ppv2(0)(24),ppv2(1)(25));
ppv2(1)(24):=ppv1(7)(10);

u25:csa_fa(ppv1(4)(17),ppv1(5)(15),ppv1(6)(13),ppv2(0)(25),ppv2(1)(26));
ppv2(2)(25):=ppv1(7)(11);

u26:csa_fa(ppv1(5)(16),ppv1(6)(14),ppv1(7)(12),ppv2(0)(26),ppv2(1)(27));

u27:csa_fa(ppv1(5)(17),ppv1(6)(15),ppv1(7)(13),ppv2(0)(27),ppv2(1)(28));

u28:csa_ha(ppv1(6)(16),ppv1(7)(14),ppv2(0)(28),ppv2(1)(29));

u29:csa_ha(ppv1(6)(17),ppv1(7)(15),ppv2(0)(29),ppv2(1)(30));

ppv2(0)(30):= ppv1(7)(16);
ppv2(0)(31):= ppv1(7)(17);
ppv2(1)(2):='0';
----------STAGE 2----------
ppv3(0)(7 downto 1):=ppv2(0)(7 downto 1);
ppv3(1)(7 downto 1):=ppv2(1)(7 downto 1);
ppv3(2)(3):=ppv2(2)(3);
ppv3(2)(7):=ppv2(2)(7);

u1_0_8:csa_fa(ppv2(0)(8),ppv2(1)(8),ppv2(2)(8),ppv3(0)(8),ppv3(1)(9));
ppv3(1)(8):=ppv2(3)(8);

u1_0_9:csa_fa(ppv2(0)(9),ppv2(1)(9),ppv2(2)(9),ppv3(0)(9),ppv3(1)(10));
ppv3(2)(9):=ppv2(3)(9);

u1_0_10:csa_fa(ppv2(0)(10),ppv2(1)(10),ppv2(2)(10),ppv3(0)(10),ppv3(1)(11));
```

```
u1_1_10:csa_ha(ppv2(3)(10),ppv2(4)(10),ppv3(2)(10),ppv3(3)(11));

u1_0_11:csa_fa(ppv2(0)(11),ppv2(1)(11),ppv2(2)(11),ppv3(0)(11),ppv3(1)(12));
ppv3(2)(11):=ppv2(3)(11);

u1_0_12:csa_fa(ppv2(0)(12),ppv2(1)(12),ppv2(2)(12),ppv3(0)(12),ppv3(1)(13));
u1_1_12:csa_ha(ppv2(3)(12),ppv2(4)(12),ppv3(2)(12),ppv3(3)(13));

u1_0_13:csa_fa(ppv2(0)(13),ppv2(1)(13),ppv2(2)(13),ppv3(0)(13),ppv3(1)(14));
u1_1_13:csa_fa(ppv2(3)(13),ppv2(4)(13),ppv2(5)(13),ppv3(2)(13),ppv3(3)(14));

u1_0_14:csa_fa(ppv2(0)(14),ppv2(1)(14),ppv2(2)(14),ppv3(0)(14),ppv3(1)(15));
u1_1_14:csa_ha(ppv2(3)(14),ppv2(4)(14),ppv3(2)(14),ppv3(3)(15));

u1_0_15:csa_fa(ppv2(0)(15),ppv2(1)(15),ppv2(2)(15),ppv3(0)(15),ppv3(1)(16));
u1_1_15:csa_fa(ppv2(3)(15),ppv2(4)(15),ppv2(5)(15),ppv3(2)(15),ppv3(3)(16));

u1_0_16:csa_fa(ppv2(0)(16),ppv2(1)(16),ppv2(2)(16),ppv3(0)(16),ppv3(1)(17));
u1_1_16:csa_fa(ppv2(3)(16),ppv2(4)(16),ppv2(5)(16),ppv3(2)(16),ppv3(3)(17));
ppv3(4)(16):=ppv2(6)(16);

u1_0_17:csa_fa(ppv2(0)(17),ppv2(1)(17),ppv2(2)(17),ppv3(0)(17),ppv3(1)(18));
u1_1_17:csa_fa(ppv2(3)(17),ppv2(4)(17),ppv2(5)(17),ppv3(2)(17),ppv3(3)(18));

u1_0_18:csa_fa(ppv2(0)(18),ppv2(1)(18),ppv2(2)(18),ppv3(0)(18),ppv3(1)(19));
u1_1_18:csa_ha(ppv2(3)(18),ppv2(4)(18),ppv3(2)(18),ppv3(3)(19));

u1_0_19:csa_ha(ppv2(0)(19),ppv2(1)(19),ppv3(0)(19),ppv3(1)(20));
u1_1_19:csa_fa(ppv2(2)(19),ppv2(3)(19),ppv2(4)(19),ppv3(2)(19),ppv3(3)(20));

u1_0_20:csa_ha(ppv2(0)(20),ppv2(1)(20),ppv3(0)(20),ppv3(1)(21));
u1_1_20:csa_fa(ppv2(2)(20),ppv2(3)(20),ppv2(4)(20),ppv3(2)(20),ppv3(3)(21));

u1_0_21:csa_ha(ppv2(0)(21),ppv2(1)(21),ppv3(0)(21),ppv3(1)(22));
u1_1_21:csa_fa(ppv2(2)(21),ppv2(3)(21),ppv2(4)(21),ppv3(2)(21),ppv3(3)(22));

u1_0_22:csa_ha(ppv2(0)(22),ppv2(1)(22),ppv3(0)(22),ppv3(1)(23));
u1_1_22:csa_fa(ppv2(2)(22),ppv2(3)(22),ppv2(4)(22),ppv3(2)(22),ppv3(3)(23));

u1_0_23:csa_fa(ppv2(3)(23),ppv2(1)(23),ppv2(2)(23),ppv3(2)(23),ppv3(1)(24));
ppv3(0)(23):=ppv2(0)(23);

u1_0_24:csa_fa(ppv2(0)(24),ppv2(1)(24),ppv2(2)(24),ppv3(0)(24),ppv3(1)(25));

u1_0_25:csa_fa(ppv2(0)(25),ppv2(1)(25),ppv2(2)(25),ppv3(0)(25),ppv3(1)(26));
```

```
        u1_0_26:csa_ha(ppv2(0)(26),ppv2(1)(26),ppv3(0)(26),ppv3(1)(27));
        u1_0_27:csa_ha(ppv2(0)(27),ppv2(1)(27),ppv3(0)(27),ppv3(1)(28));
        u1_0_28:csa_ha(ppv2(0)(28),ppv2(1)(28),ppv3(0)(28),ppv3(1)(29));
        u1_0_29:csa_ha(ppv2(0)(29),ppv2(1)(29),ppv3(0)(29),ppv3(1)(30));
        u1_0_30:csa_ha(ppv2(0)(30),ppv2(1)(30),ppv3(0)(30),ppv3(1)(31));

        ppv3(0)(31):=ppv2(0)(31);
        ppv3(2)(6 downto 4):=(others=>'0');
        ppv3(2)(2 downto 1):=(others=>'0');
        ppv3(2)(8):='0';
        ppv3(3)(12):='0';
        pps10<=ppv3;
end process PPS_REDN1;
-------------------------------------------------------------------------------------------------
---------------------------------------- PIPELINE REGISTER 3 ------------------------------------
PPS1_GCLK2:process (gclk2)
begin
       if gclk2'event and gclk2='1' then
        pps1(0)(31 downto 1)<=pps10(0)(23 downto 1);
        pps1(1)(31 downto 1)<=pps10(1)(23 downto 1);
        pps1(2)(24 downto 1)<=pps10(2)(7 downto 1);
        pps1(3)(23 downto 11)<=pps10(3)(23 downto 11);
        pps1(4)(16)<=pps10(4)(16);
       end if;
end process PPS1_GCLK2;
-------------------------------------------------------------------------------------------------
---------------------------------------- PP Reduction 2 -----------------------------------------
PPS_REDN2 : process (pps1)
  variable ppv4,ppv5,ppv6: ary1;
begin
 ppv4:=pps1;
        ----------------------------STAGE 1-------------------------
         ppv5(0)(10 downto 1):= ppv4(0)(10 downto 1);
         ppv5(1)(10 downto 1):= ppv4(1)(10 downto 1);
         ppv5(2)(10 downto 3):= ppv4(2)(10 downto 3);

         u2_11:csa_fa(ppv4(0)(11),ppv4(1)(11),ppv4(2)(11),ppv5(0)(11),ppv5(1)(12));
         ppv5(1)(11):=ppv4(3)(11);

         u2_12:csa_fa(ppv4(0)(12),ppv4(1)(12),ppv4(2)(12),ppv5(0)(12),ppv5(1)(13));

        u2_13:csa_fa(ppv4(0)(13),ppv4(1)(13),ppv4(2)(13),ppv5(0)(13),ppv5(1)(14));
        ppv5(2)(13):=ppv4(3)(13);

        u2_14:csa_fa(ppv4(0)(14),ppv4(1)(14),ppv4(2)(14),ppv5(0)(14),ppv5(1)(15));
        ppv5(2)(14):=ppv4(3)(14);
```

```
u2_15:csa_fa(ppv4(0)(15),ppv4(1)(15),ppv4(2)(15),ppv5(0)(15),ppv5(1)(16));
ppv5(2)(15):=ppv4(3)(15);

u2_16:csa_fa(ppv4(0)(16),ppv4(1)(16),ppv4(2)(16),ppv5(0)(16),ppv5(1)(17));
u2_1_16:csa_ha(ppv4(3)(16),ppv4(4)(16),ppv5(2)(16),ppv5(3)(17));

u2_17:csa_fa(ppv4(0)(17),ppv4(1)(17),ppv4(2)(17),ppv5(0)(17),ppv5(1)(18));
ppv5(2)(17):=ppv4(3)(17);

u2_18:csa_fa(ppv4(0)(18),ppv4(1)(18),ppv4(2)(18),ppv5(0)(18),ppv5(1)(19));
ppv5(2)(18):=ppv4(3)(18);

u2_19:csa_fa(ppv4(0)(19),ppv4(1)(19),ppv4(2)(19),ppv5(0)(19),ppv5(1)(20));
ppv5(2)(19):=ppv4(3)(19);

u2_20:csa_fa(ppv4(0)(20),ppv4(1)(20),ppv4(2)(20),ppv5(0)(20),ppv5(1)(21));
ppv5(2)(20):=ppv4(3)(20);

u2_21:csa_fa(ppv4(0)(21),ppv4(1)(21),ppv4(2)(21),ppv5(0)(21),ppv5(1)(22));
ppv5(2)(21):=ppv4(3)(21);
u2_22:csa_fa(ppv4(0)(22),ppv4(1)(22),ppv4(2)(22),ppv5(0)(22),ppv5(1)(23));
ppv5(2)(22):=ppv4(3)(22);
u2_23:csa_fa(ppv4(0)(23),ppv4(1)(23),ppv4(2)(23),ppv5(0)(23),ppv5(1)(24));
ppv5(2)(23):=ppv4(3)(23);

u2_24:csa_ha(ppv4(0)(24),ppv4(1)(24),ppv5(0)(24),ppv5(1)(25));
u2_25:csa_ha(ppv4(0)(25),ppv4(1)(25),ppv5(0)(25),ppv5(1)(26));
u2_26:csa_ha(ppv4(0)(26),ppv4(1)(26),ppv5(0)(26),ppv5(1)(27));
u2_27:csa_ha(ppv4(0)(27),ppv4(1)(27),ppv5(0)(27),ppv5(1)(28));
u2_28:csa_ha(ppv4(0)(28),ppv4(1)(28),ppv5(0)(28),ppv5(1)(29));
u2_29:csa_ha(ppv4(0)(29),ppv4(1)(29),ppv5(0)(29),ppv5(1)(30));
u2_30:csa_ha(ppv4(0)(30),ppv4(1)(30),ppv5(0)(30),ppv5(1)(31));
u2_31:csa_ha(ppv4(0)(31),ppv4(1)(31),ppv5(0)(31),ppv5(0)(32));
ppv5(2)(12 downto 11):=(others=>'0');
---------------------------------STAGE 2-------------------------
ppv6(0)(16 downto 1):= ppv5(0)(16 downto 1);
ppv6(1)(16 downto 1):= ppv5(1)(16 downto 1);
ppv6(2)(16 downto 3):= ppv5(2)(16 downto 3);

u2_1_17:csa_fa(ppv5(0)(17),ppv5(1)(17),ppv5(2)(17),ppv6(0)(17),ppv6(1)(18))
ppv6(1)(17):=ppv5(3)(17);

u2_1_18:csa_fa(ppv5(0)(18),ppv5(1)(18),ppv5(2)(18),ppv6(0)(18),ppv6(1)(19));
u2_1_19:csa_fa(ppv5(0)(19),ppv5(1)(19),ppv5(2)(19),ppv6(0)(19),ppv6(1)(20));
u2_1_20:csa_fa(ppv5(0)(20),ppv5(1)(20),ppv5(2)(20),ppv6(0)(20),ppv6(1)(21));
```

```
        u2_1_21:csa_fa(ppv5(0)(21),ppv5(1)(21),ppv5(2)(21),ppv6(0)(21),ppv6(1)(22));
        u2_1_22:csa_fa(ppv5(0)(22),ppv5(1)(22),ppv5(2)(22),ppv6(0)(22),ppv6(1)(23));
        u2_1_23:csa_fa(ppv5(0)(23),ppv5(1)(23),ppv5(2)(23),ppv6(0)(23),ppv6(1)(24));
        u2_1_24:csa_ha(ppv5(0)(24),ppv5(1)(24),ppv6(0)(24),ppv6(1)(25));
        u2_1_25:csa_ha(ppv5(0)(25),ppv5(1)(25),ppv6(0)(25),ppv6(1)(26));
        u2_1_26:csa_ha(ppv5(0)(26),ppv5(1)(26),ppv6(0)(26),ppv6(1)(27));
        u2_1_27:csa_ha(ppv5(0)(27),ppv5(1)(27),ppv6(0)(27),ppv6(1)(28));
        u2_1_28:csa_ha(ppv5(0)(28),ppv5(1)(28),ppv6(0)(28),ppv6(1)(29));
        u2_1_29:csa_ha(ppv5(0)(29),ppv5(1)(29),ppv6(0)(29),ppv6(1)(30));
        u2_1_30:csa_ha(ppv5(0)(30),ppv5(1)(30),ppv6(0)(30),ppv6(1)(31));
        u2_1_31:csa_ha(ppv5(0)(31),ppv5(1)(31),ppv6(0)(31),ppv6(1)(32));
    ppv6(0)(32):=ppv5(0)(32);
    pps20<=ppv6;
end process PPS_REDN2;
-------------------------------------------------------------------------------------------------
---------------------------------------- PIPELINE REGISTER 3 ------------------------------------
PPS2_GCLK3: process(gclk3)
begin
        if gclk3'event and gclk3='1' then
          pps2(0)(32 downto 1)<=pps20(0)(32 downto 1);
          pps2(1)(32 downto 1)<=pps20(1)(32 downto 1);
          pps2(2)(16 downto 3)<= pps20(2)(16 downto 3);
        end if ;
end process PPS2_GCLK3;
-------------------------------------------------------------------------------------------------
---------------------------------------- MUX Process --------------------------------------------
MUX_process: process(pps1,pps2,pps,gs1,gs2,gs3)
variable m:std_logic_vector(2 downto 0);
variable cv3: std_logic_vector (15 downto 0);
variable ci3: integer range -2**(15) to ( 2**(15))-1;
begin
m:=gs1&gs2&gs3;
ci3:= -(((2**16)-1)/3);
cv3 := std_logic_vector(conv_unsigned(ci3,16));
gating:case m is
       when "111"=>
                                        pps3(0)(32 downto 1)<=pps2(0)(32 downto 1);
                                        pps3(1)(32 downto 1)<=pps2(1)(32 downto 1);
                                        pps3(2)(2 downto 1)<=(others=>'0');
                                        pps3(2)(16 downto 3)<= pps2(2)(16 downto 3);
                                        pps3(2)(32 downto 17)<= cv3;
       when "110"=>
                                        pps3(0)(16 downto 1)<=pps1(0)(16 downto 1);
                                        pps3(0)(32 downto 17)<=(others=>'0');
                                        pps3(1)(15 downto 1)<=pps1(1)(15 downto 1);
                                        pps3(1)(32 downto 16)<=(others=>'0');
```

```
                                    pps3(2)(8 downto 1)<=pps1(2)(8 downto 1);
                                    pps3(2)( 32 downto 9)<=(others=>'0');
        when "100"=>
                                    pps3(0)(8 downto 1)<=pps(0)(8 downto 1);
                                    pps3(0)(32 downto 9)<=(others=>'0');
                                    pps3(1)(1)<=pps(0)(0);
                                    pps3(1)(2)<='0';
                                    pps3(1)(8 downto 3)<=pps(1)(6 downto 1);
                                    pps3(1)(32 downto 9)<=(others=>'0');
                                    pps3(2)( 2 downto 1)<=(others=>'0');
                                    pps3(2)(3)<=pps(1)(0);
                                    pps3(2)(4)<='0';
                                    pps3(2)(32 downto 5)<=(others=>'0');

        when others=>
                                   for i in 0 to 2 loop
                                      pps3(i)<=(others=>'0');
                                   end loop;
end case gating;
end process MUX_process;
-------------------------------------------------------------------------------------------
---------------------------------------- PP Reduction 3 -----------------------------------
PPS_REDN_final : process(pps3)
variable ppv7:ary2;
variable ppv8:ary3;
variable nc1:std_logic;
begin
ppv7:=pps3;
   ppv8(0)(2 downto 1):= ppv7(0)(2 downto 1);
   ppv8(1)(2 downto 1):= ppv7(1)(2 downto 1);
   ppv8(1)(3):='0';
        u_F_3:csa_fa(ppv7(0)(3),ppv7(1)(3),ppv7(2)(3),ppv8(0)(3),ppv8(1)(4));
        u_F_4:csa_ha(ppv7(0)(4),ppv7(1)(4),ppv8(0)(4),ppv8(1)(5));
        u_F_5:csa_ha(ppv7(0)(5),ppv7(1)(5),ppv8(0)(5),ppv8(1)(6));
        u_F_6:csa_ha(ppv7(0)(6),ppv7(1)(6),ppv8(0)(6),ppv8(1)(7));
        u_F_7:csa_fa(ppv7(0)(7),ppv7(1)(7),ppv7(2)(7),ppv8(0)(7),ppv8(1)(8));
        u_F_8:csa_ha(ppv7(0)(8),ppv7(1)(8),ppv8(0)(8),ppv8(1)(9));
        u_F_9:csa_fa(ppv7(0)(9),ppv7(1)(9),ppv7(2)(9),ppv8(0)(9),ppv8(1)(10));
        u_F_10:csa_fa(ppv7(0)(10),ppv7(1)(10),ppv7(2)(10),ppv8(0)(10),ppv8(1)(11));
        u_F_11:csa_ha(ppv7(0)(11),ppv7(1)(11),ppv8(0)(11),ppv8(1)(12));
        u_F_12:csa_ha(ppv7(0)(12),ppv7(1)(12),ppv8(0)(12),ppv8(1)(13));
        u_F_13:csa_fa(ppv7(0)(13),ppv7(1)(13),ppv7(2)(13),ppv8(0)(13),ppv8(1)(14));
        u_F_14:csa_fa(ppv7(0)(14),ppv7(1)(14),ppv7(2)(14),ppv8(0)(14),ppv8(1)(15));
        u_F_15:csa_fa(ppv7(0)(15),ppv7(1)(15),ppv7(2)(15),ppv8(0)(15),ppv8(1)(16));
        u_F_16:csa_fa(ppv7(0)(16),ppv7(1)(16),ppv7(2)(16),ppv8(0)(16),ppv8(1)(17));
        u_F_17:csa_fa(ppv7(0)(17),ppv7(1)(17),ppv7(2)(17),ppv8(0)(17),ppv8(1)(18));
```

```
        u_F_18:csa_fa(ppv7(0)(18),ppv7(1)(18),ppv7(2)(18),ppv8(0)(18),ppv8(1)(19));
        u_F_19:csa_fa(ppv7(0)(19),ppv7(1)(19),ppv7(2)(19),ppv8(0)(19),ppv8(1)(20));
        u_F_20:csa_fa(ppv7(0)(20),ppv7(1)(20),ppv7(2)(20),ppv8(0)(20),ppv8(1)(21));
        u_F_21:csa_fa(ppv7(0)(21),ppv7(1)(21),ppv7(2)(21),ppv8(0)(21),ppv8(1)(22));
        u_F_22:csa_fa(ppv7(0)(22),ppv7(1)(22),ppv7(2)(22),ppv8(0)(22),ppv8(1)(23));
        u_F_23:csa_fa(ppv7(0)(23),ppv7(1)(23),ppv7(2)(23),ppv8(0)(23),ppv8(1)(24));
        u_F_24:csa_fa(ppv7(0)(24),ppv7(1)(24),ppv7(2)(24),ppv8(0)(24),ppv8(1)(25));
        u_F_25:csa_fa(ppv7(0)(25),ppv7(1)(25),ppv7(2)(25),ppv8(0)(25),ppv8(1)(26));
        u_F_26:csa_fa(ppv7(0)(26),ppv7(1)(26),ppv7(2)(26),ppv8(0)(26),ppv8(1)(27));
        u_F_27:csa_fa(ppv7(0)(27),ppv7(1)(27),ppv7(2)(27),ppv8(0)(27),ppv8(1)(28));
        u_F_28:csa_fa(ppv7(0)(28),ppv7(1)(28),ppv7(2)(28),ppv8(0)(28),ppv8(1)(29));
        u_F_29:csa_fa(ppv7(0)(29),ppv7(1)(29),ppv7(2)(29),ppv8(0)(29),ppv8(1)(30));
        u_F_30:csa_fa(ppv7(0)(30),ppv7(1)(30),ppv7(2)(30),ppv8(0)(30),ppv8(1)(31));
        u_F_31:csa_fa(ppv7(0)(31),ppv7(1)(31),ppv7(2)(31),ppv8(0)(31),ppv8(1)(32));
        u_F_32:csa_fa(ppv7(0)(32),ppv7(1)(32),ppv7(2)(32),ppv8(0)(32),nc1);
pps40<=ppv8;
end process PPS_REDN_final;
-------------------------------------------------------------------------------------------------
------------------------------------- PIPELINE REGISTER 5 -------------------------------
PPSf_GCLK1: process(gclk1)
begin
        if gclk1'event and gclk1='1' then
         pps4(0)(32 downto 1)<=pps40(0)(8 downto 1);
         pps4(1)(32 downto 1)<=pps40(1)(8 downto 1);
        end if;
end process PPSf_GCLK1;
-------------------------------------------------------------------------------------------------
-----------------------------------------------RCA-----------------------------------------------
RIPPLE:process (pps4)
variable ppv9:ary3;
variable ppv10: std_logic_vector(m+n downto 1);
begin
ppv9:=pps4;
rca(ppv9(0),ppv9(1),ppv10);
pps5<=ppv10;
end process RIPPLE;
-------------------------------------------------------------------------------------------------
------------------------------------- PIPELINE REGISTER 6 -------------------------------
PROD_GCLK1:process(gclk1)
begin
        if gclk1'event and gclk1='1' then
         prod( 31 downto 0)<=pps5(32 downto 1);
        end if;
end process PROD_GCLK1;
end Behavioral;
------------------------- END OF 1D-MULTIPLIER ARCHITECTURE-------------------
```

APPENDIX-C

C. Non-pipeline gated Booth multiplier

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity multiplier_p is
   Generic (m:integer :=16;n: integer :=16);

   Port ( mier : in  STD_LOGIC_VECTOR (m-1 downto 0);
          mcand : in  STD_LOGIC_VECTOR (n-1 downto 0);
          clk : in  STD_LOGIC;
          prod : out  STD_LOGIC_VECTOR (m+n-1 downto 0));
end multiplier_p;
-------------------------ARHITECTURE OF 1D BOOTH MULTIPLIER----------------------
architecture Behavioral of multiplier_p is
type ary is array(0 to m/2-1) of STD_LOGIC_VECTOR( n+1 downto 0);
type ary1 is array(0 to 5) of STD_LOGIC_VECTOR( m+n downto 1);
type ary2 is array(0 to 2) of STD_LOGIC_VECTOR( m+n downto 1);
type ary3 is array(0 to 1) of STD_LOGIC_VECTOR( m+n downto 1);

signal p_prod0,p_prod1,p_prod2,p_prod3,p_prod4,p_prod5 : STD_LOGIC_VECTOR(
m+n-1 downto 0);
signal pps: ary;
-------------------------------BOOTH ENCODER AND PP GENERATOR------------------
PROCEDURE Booth_PP_gen(a:in  std_logic_vector(2 downto 0);md: in
std_logic_vector;PP:out std_logic_vector)
 is
        variable bb       : std_logic_vector (md'range);
        variable psum     : std_logic_vector (md'range);
        variable b_bar    : std_logic_vector (md'range);
        variable two_b    : std_logic_vector (md'range);
        variable two_b_bar : std_logic_vector (md'range);
        variable cin      : std_logic;
        variable topbit   : std_logic;
        variable topout   : std_logic;
begin
        two_b:=md(md'left-1 downto 0) & '0';
        b_bar:=not md;
        two_b_bar:=not two_b;

        case a is
```

```
        when "001" | "010" =>
            bb:= md;
            cin:='0';
        when "011" =>
            bb:= two_b;
            cin:='0';
        when  "100"  =>
            bb:= two_b_bar;
            cin:='1';
        when   "101" | "110" =>
            bb:= b_bar;
            cin:='1';
        when others =>
            bb:=(others=>'0');
            cin:='0';
        end case;

        top: case a is
            when "001"|"010"|"011" => topbit:=not md(md'left);
            when "100"|"101"|"110" => topbit:=md(md'left);
            when "000"|"111"=>topbit:='1';
            when others =>topbit:='0';
        end case top;

        PP:=topbit&bb&cin;

end Booth_PP_gen;
-------------------------------------------------------------------------------------------
---------------------------------------- [2:2] compressor----------------------------------
PROCEDURE CSA_ha( a:in std_logic ;b: in std_logic;s: out std_logic;c: out std_logic) is
begin
s:= a xor b;
c:= a and b;
end PROCEDURE CSA_ha;
-------------------------------------------------------------------------------------------
---------------------------------------- [3:2] compressor----------------------------------
PROCEDURE CSA_fa( a:in std_logic ;b: in std_logic ;d : in std_logic ;s: out std_logic;
c: out std_logic) is
begin
  s:= a xor b xor d;
  c:= (a and b) or ( a and d) or (b and d);
end PROCEDURE CSA_fa;
-------------------------------------------------------------------------------------------
-----------------------------------------RCA-----------------------------------------------
PROCEDURE rca ( a : in std_logic_vector; b: in std_logic_vector;sum:out
std_logic_vector) is
```

```
variable c: std_logic_vector(a'range);
variable nc:std_logic;
begin
for i in a'right to a'left loop
 if i=a'right then
   CSA_fa(a(i),b(i),'0',sum(i),c(i));
 elsif i/=a'left then
   CSA_fa(a(i),b(i),c(i-1),sum(i),c(i));
 else
 CSA_fa(a(i),b(i),c(i-1),sum(i),nc);
 end if;
end loop;
end PROCEDURE rca;
-BEGINNING ARHITECTURE OF NON PIPELINE GATED BOOTH MULTIPLIER -
Begin
--------------------------------------------------------------------------------------------------
---------------------------------------- PP Generation ----------------------------------------
PP_GEN:process (mier,mcand)
  variable ppv : ary;
  variable a: std_logic_vector(2 downto 0);
begin
        a:=mier(1 downto 0) &'0';
        Booth_PP_gen(a,mcand,ppv(0));
        a:=mier(3 downto 1);
        Booth_PP_gen(a,mcand,ppv(1));
        a:=mier(5 downto 3);
        Booth_PP_gen(a,mcand,ppv(2));
        a:=mier(7 downto 5);
        Booth_PP_gen(a,mcand ,ppv(3));
        a:=mier(9 downto 7) ;
        Booth_PP_gen(a,mcand,ppv(4));
        a:=mier(11 downto 9);
        Booth_PP_gen(a,mcand,ppv(5));
        a:=mier(13 downto 11) ;
        Booth_PP_gen(a,mcand,ppv(6));
        a:=mier(15 downto 13);
        Booth_PP_gen(a,mcand,ppv(7));
      pps<=ppv;
 end process PP_GEN;
--------------------------------------------------------------------------------------------------
---------------------------------------- PP Reduction----------------------------------------
PPS_REDN: process (pps)
      variable ppv1 : ary;
      variable ppv2,ppv3: ary1;
      variable ppv4,ppv5,ppv6: ary1;
      variable ppv7:ary2;
```

```
        variable ppv8:ary3;
        variable nc1:std_logic;
        variable cv3: std_logic_vector (15 downto 0);
        variable ci3: integer range -2**(15) to ( 2**(15))-1;
        variable ppv10: std_logic_vector(m+n downto 1);

begin
        ppv1:=pps;
        ------------------------------STAGE 1-------------------------
        ppv2(1)(1):=ppv1(0)(0);
        ppv2(0)(1):=ppv1(0)(1);
        ppv2(0)(2):=ppv1(0)(2);

        u3:csa_fa(ppv1(0)(3),ppv1(1)(1),ppv1(1)(0),ppv2(0)(3),ppv2(1)(4));
        u4:csa_ha(ppv1(0)(4),ppv1(1)(2),ppv2(0)(4),ppv2(1)(5));
        u5:csa_fa(ppv1(0)(5),ppv1(1)(3),ppv1(2)(1),ppv2(0)(5),ppv2(1)(6));
        ppv2(2)(5):=ppv1(2)(0);

        u6:csa_fa(ppv1(0)(6),ppv1(1)(4),ppv1(2)(2),ppv2(0)(6),ppv2(1)(7));

        u7:csa_fa(ppv1(0)(7),ppv1(1)(5),ppv1(2)(3),ppv2(0)(7),ppv2(1)(8));
        u1_7: csa_ha(ppv1(3)(1),ppv1(3)(0),ppv2(2)(7),ppv2(3)(8));

        u8:csa_fa(ppv1(0)(8),ppv1(1)(6),ppv1(2)(4),ppv2(0)(8),ppv2(1)(9));
        ppv2(2)(8):=ppv1(3)(2);

        u9:csa_fa(ppv1(0)(9),ppv1(1)(7),ppv1(2)(5),ppv2(0)(9),ppv2(1)(10));
        u1_9:csa_fa(ppv1(3)(3),ppv1(4)(1),ppv1(4)(0),ppv2(2)(9),ppv2(2)(10));

        u10:csa_fa(ppv1(3)(4),ppv1(1)(8),ppv1(2)(6),ppv2(0)(10),ppv2(1)(11));
        u1_10:csa_ha(ppv1(0)(10),ppv1(4)(2),ppv2(3)(10),ppv2(3)(11));

        u11:csa_fa(ppv1(3)(5),ppv1(1)(9),ppv1(2)(7),ppv2(0)(11),ppv2(1)(12));
        u1_11:csa_fa(ppv1(5)(0),ppv1(4)(3),ppv1(5)(1),ppv2(2)(11),ppv2(3)(12));
        ppv2(4)(11):=ppv1(0)(11);

        u12:csa_fa(ppv1(2)(8),ppv1(3)(6),ppv1(4)(4),ppv2(0)(12),ppv2(1)(13));
        u1_12:csa_fa(ppv1(0)(12),ppv1(1)(10),ppv1(5)(2),ppv2(2)(12),ppv2(3)(13));

        u13:csa_fa(ppv1(2)(9),ppv1(3)(7),ppv1(4)(5),ppv2(0)(13),ppv2(1)(14));
        u1_13:csa_fa(ppv1(6)(0),ppv1(6)(1),ppv1(5)(3),ppv2(2)(13),ppv2(3)(14));
        u2_13:csa_ha(ppv1(0)(13),ppv1(1)(11),ppv2(4)(13),ppv2(5)(14));
        u14:csa_fa(ppv1(0)(14),ppv1(1)(12),ppv1(2)(10),ppv2(2)(14),ppv2(3)(15));
        u1_14:csa_fa(ppv1(3)(8),ppv1(4)(6),ppv1(5)(4),ppv2(0)(14),ppv2(1)(15));
        ppv2(4)(14):=ppv1(6)(2);
```

```
u15:csa_fa(ppv1(0)(15),ppv1(1)(13),ppv1(2)(11),ppv2(2)(15),ppv2(3)(16));
u1_15:csa_fa(ppv1(3)(9),ppv1(4)(7),ppv1(5)(5),ppv2(0)(15),ppv2(1)(16));
u2_15:csa_fa(ppv1(6)(3),ppv1(7)(0),ppv1(7)(1),ppv2(4)(15),ppv2(5)(16));

u16:csa_fa(ppv1(0)(16),ppv1(1)(14),ppv1(2)(12),ppv2(0)(16),ppv2(1)(17));
u1_16:csa_fa(ppv1(3)(10),ppv1(4)(8),ppv1(5)(6),ppv2(2)(16),ppv2(3)(17));
u2_16:csa_ha(ppv1(6)(4),ppv1(7)(2),ppv2(4)(16),ppv2(5)(17));

u17:csa_fa(ppv1(0)(17),ppv1(1)(15),ppv1(2)(13),ppv2(0)(17),ppv2(1)(18));
u1_17:csa_fa(ppv1(3)(11),ppv1(4)(9),ppv1(5)(7),ppv2(2)(17),ppv2(3)(18));
u2_17:csa_ha(ppv1(6)(5),ppv1(7)(3),ppv2(4)(17),ppv2(5)(18));

u18:csa_fa(ppv1(1)(16),ppv1(2)(14),ppv1(3)(12),ppv2(0)(18),ppv2(1)(19));
u1_18:csa_fa(ppv1(4)(10),ppv1(5)(8),ppv1(6)(6),ppv2(2)(18),ppv2(3)(19));
ppv2(4)(18):=ppv1(7)(4);

u19:csa_fa(ppv1(1)(17),ppv1(2)(15),ppv1(3)(13),ppv2(0)(19),ppv2(1)(20));
u1_19:csa_fa(ppv1(4)(11),ppv1(5)(9),ppv1(6)(7),ppv2(2)(19),ppv2(3)(20));
ppv2(4)(19):=ppv1(7)(5);

u20:csa_fa(ppv1(2)(16),ppv1(3)(14),ppv1(4)(12),ppv2(0)(20),ppv2(1)(21));
u1_20:csa_fa(ppv1(5)(10),ppv1(6)(8),ppv1(7)(6),ppv2(2)(20),ppv2(3)(21));

u21:csa_fa(ppv1(2)(17),ppv1(3)(15),ppv1(4)(13),ppv2(0)(21),ppv2(1)(22));
u1_21:csa_fa(ppv1(5)(11),ppv1(6)(9),ppv1(7)(7),ppv2(2)(21),ppv2(3)(22));

u22:csa_fa(ppv1(3)(16),ppv1(4)(14),ppv1(5)(12),ppv2(0)(22),ppv2(1)(23));
u1_22:csa_ha(ppv1(6)(10),ppv1(7)(8),ppv2(2)(22),ppv2(3)(23));

u23:csa_fa(ppv1(3)(17),ppv1(4)(15),ppv1(5)(13),ppv2(0)(23),ppv2(1)(24));
u1_23:csa_ha(ppv1(6)(11),ppv1(7)(9),ppv2(2)(23),ppv2(2)(24));

u24:csa_fa(ppv1(4)(16),ppv1(5)(14),ppv1(6)(12),ppv2(0)(24),ppv2(1)(25));
ppv2(3)(24):=ppv1(7)(10);

u25:csa_fa(ppv1(4)(17),ppv1(5)(15),ppv1(6)(13),ppv2(0)(25),ppv2(1)(26));
ppv2(2)(25):=ppv1(7)(11);

u26:csa_fa(ppv1(5)(16),ppv1(6)(14),ppv1(7)(12),ppv2(0)(26),ppv2(1)(27));

u27:csa_fa(ppv1(5)(17),ppv1(6)(15),ppv1(7)(13),ppv2(0)(27),ppv2(1)(28));

u28:csa_ha(ppv1(6)(16),ppv1(7)(14),ppv2(0)(28),ppv2(1)(29));

u29:csa_ha(ppv1(6)(17),ppv1(7)(15),ppv2(0)(29),ppv2(1)(30));
```

```
ppv2(0)(30):= ppv1(7)(16);
ppv2(0)(31):= ppv1(7)(17);
----------------------------STAGE 2-------------------------
ppv3(0)(1):=ppv2(0)(1);
ppv3(0)(2):=ppv2(0)(2);
ppv3(0)(3):=ppv2(0)(3);
ppv3(1)(1):=ppv2(1)(1);
ppv3(0)(4):=ppv2(0)(4);
ppv3(1)(4):=ppv2(1)(4);

u1_0_5:csa_fa(ppv2(0)(5),ppv2(1)(5),ppv2(2)(5),ppv3(0)(5),ppv3(1)(6));

u1_0_6:csa_ha(ppv2(0)(6),ppv2(1)(6),ppv3(0)(6),ppv3(1)(7));
u1_0_7:csa_fa(ppv2(0)(7),ppv2(1)(7),ppv2(2)(7),ppv3(0)(7),ppv3(1)(8));
u1_0_8:csa_fa(ppv2(0)(8),ppv2(1)(8),ppv2(2)(8),ppv3(0)(8),ppv3(1)(9));
ppv3(2)(8):=ppv2(3)(8);
u1_0_9:csa_fa(ppv2(0)(9),ppv2(1)(9),ppv2(2)(9),ppv3(0)(9),ppv3(1)(10));
u1_0_10:csa_fa(ppv2(0)(10),ppv2(1)(10),ppv2(2)(10),ppv3(0)(10),ppv3(1)(11));
ppv3(2)(10):=ppv2(3)(10);

u1_0_11:csa_fa(ppv2(0)(11),ppv2(1)(11),ppv2(2)(11),ppv3(0)(11),ppv3(1)(12));
u1_1_11:csa_ha(ppv2(3)(11),ppv2(4)(11),ppv3(2)(11),ppv3(3)(12));

u1_0_12:csa_fa(ppv2(0)(12),ppv2(1)(12),ppv2(2)(12),ppv3(0)(12),ppv3(1)(13));
ppv3(2)(12):=ppv2(3)(12);

u1_0_13:csa_fa(ppv2(0)(13),ppv2(1)(13),ppv2(2)(13),ppv3(0)(13),ppv3(1)(14));
u1_1_13:csa_ha(ppv2(3)(13),ppv2(4)(13),ppv3(2)(13),ppv3(3)(14));

u1_0_14:csa_fa(ppv2(0)(14),ppv2(1)(14),ppv2(2)(14),ppv3(0)(14),ppv3(1)(15));
u1_1_14:csa_fa(ppv2(3)(14),ppv2(4)(14),ppv2(5)(14),ppv3(2)(14),ppv3(3)(15));

u1_0_15:csa_fa(ppv2(0)(15),ppv2(1)(15),ppv2(2)(15),ppv3(0)(15),ppv3(1)(16));
u1_1_15:csa_ha(ppv2(3)(15),ppv2(4)(15),ppv3(2)(15),ppv3(3)(16));

u1_0_16:csa_fa(ppv2(0)(16),ppv2(1)(16),ppv2(2)(16),ppv3(0)(16),ppv3(1)(17));
u1_1_16:csa_fa(ppv2(3)(16),ppv2(4)(16),ppv2(5)(16),ppv3(2)(16),ppv3(3)(17));

u1_0_17:csa_fa(ppv2(0)(17),ppv2(1)(17),ppv2(2)(17),ppv3(0)(17),ppv3(1)(18));
u1_1_17:csa_fa(ppv2(3)(17),ppv2(4)(17),ppv2(5)(17),ppv3(2)(17),ppv3(3)(18));

u1_0_18:csa_fa(ppv2(0)(18),ppv2(1)(18),ppv2(2)(18),ppv3(0)(18),ppv3(1)(19));
u1_1_18:csa_fa(ppv2(3)(18),ppv2(4)(18),ppv2(5)(18),ppv3(2)(18),ppv3(3)(19));

u1_0_19:csa_fa(ppv2(0)(19),ppv2(1)(19),ppv2(2)(19),ppv3(0)(19),ppv3(1)(20));
u1_1_19:csa_ha(ppv2(3)(19),ppv2(4)(19),ppv3(2)(19),ppv3(3)(20));
```

```
u1_0_20:csa_fa(ppv2(0)(20),ppv2(1)(20),ppv2(2)(20),ppv3(0)(20),ppv3(1)(21));
ppv3(2)(20):=ppv2(3)(20);

u1_0_21:csa_fa(ppv2(0)(21),ppv2(1)(21),ppv2(2)(21),ppv3(0)(21),ppv3(1)(22));
ppv3(2)(21):=ppv2(3)(21);

u1_0_22:csa_fa(ppv2(0)(22),ppv2(1)(22),ppv2(2)(22),ppv3(0)(22),ppv3(1)(23));
ppv3(2)(22):=ppv2(3)(22);

u1_0_23:csa_fa(ppv2(0)(23),ppv2(1)(23),ppv2(2)(23),ppv3(0)(23),ppv3(1)(24));
ppv3(2)(23):=ppv2(3)(23);

u1_0_24:csa_fa(ppv2(0)(24),ppv2(1)(24),ppv2(2)(24),ppv3(0)(24),ppv3(1)(25));
ppv3(2)(24):=ppv2(3)(24);

u1_0_25:csa_fa(ppv2(0)(25),ppv2(1)(25),ppv2(2)(25),ppv3(0)(25),ppv3(1)(26));

u1_0_26:csa_ha(ppv2(0)(26),ppv2(1)(26),ppv3(0)(26),ppv3(1)(27));
u1_0_27:csa_ha(ppv2(0)(27),ppv2(1)(27),ppv3(0)(27),ppv3(1)(28));
u1_0_28:csa_ha(ppv2(0)(28),ppv2(1)(28),ppv3(0)(28),ppv3(1)(29));
u1_0_29:csa_ha(ppv2(0)(29),ppv2(1)(29),ppv3(0)(29),ppv3(1)(30));
u1_0_30:csa_ha(ppv2(0)(30),ppv2(1)(30),ppv3(0)(30),ppv3(1)(31));

ppv3(0)(31):=ppv2(0)(31);
ppv3(1)(2):='0';
ppv3(1)(3):='0';
ppv3(1)(5):='0';
ppv3(2)(7 downto 1):=(others=>'0');
ppv3(3)(32 downto 21):=(others=>'0');

 ppv4:=ppv3;
----------------------------STAGE 3-------------------------
ppv5(0)(7 downto 1):= ppv4(0)(7 downto 1);
ppv5(1)(1):=ppv4(1)(1);
ppv5(1)(4):=ppv4(1)(4);
ppv5(1)(6):=ppv4(1)(6);
ppv5(1)(7):=ppv4(1)(7);

 u3_8:csa_fa(ppv4(0)(8),ppv4(1)(8),ppv4(2)(8),ppv5(0)(8),ppv5(1)(9));
u3_9:csa_ha(ppv4(0)(9),ppv4(1)(9),ppv5(0)(9),ppv5(1)(10));
u3_10:csa_fa(ppv4(0)(10),ppv4(1)(10),ppv4(2)(10),ppv5(0)(10),ppv5(1)(11));
u3_11:csa_fa(ppv4(0)(11),ppv4(1)(11),ppv4(2)(11),ppv5(0)(11),ppv5(1)(12));
u3_12:csa_fa(ppv4(0)(12),ppv4(1)(12),ppv4(2)(12),ppv5(0)(12),ppv5(1)(13));
ppv5(2)(12):=ppv4(3)(12);
```

```
u3_13:csa_fa(ppv4(0)(13),ppv4(1)(13),ppv4(2)(13),ppv5(0)(13),ppv5(1)(14));
u3_14:csa_fa(ppv4(0)(14),ppv4(1)(14),ppv4(2)(14),ppv5(0)(14),ppv5(1)(15));
ppv5(2)(14):=ppv4(3)(14);

u3_15:csa_fa(ppv4(0)(15),ppv4(1)(15),ppv4(2)(15),ppv5(0)(15),ppv5(1)(16));
ppv5(2)(15):=ppv4(3)(15);

u3_16:csa_fa(ppv4(0)(16),ppv4(1)(16),ppv4(2)(16),ppv5(0)(16),ppv5(1)(17));
ppv5(2)(16):=ppv4(3)(16);

u3_17:csa_fa(ppv4(0)(17),ppv4(1)(17),ppv4(2)(17),ppv5(0)(17),ppv5(1)(18));
ppv5(2)(17):=ppv4(3)(17);

u3_18:csa_fa(ppv4(0)(18),ppv4(1)(18),ppv4(2)(18),ppv5(0)(18),ppv5(1)(19));
ppv5(2)(18):=ppv4(3)(18);

u3_19:csa_fa(ppv4(0)(19),ppv4(1)(19),ppv4(2)(19),ppv5(0)(19),ppv5(1)(20));
ppv5(2)(19):=ppv4(3)(19);

u3_20:csa_fa(ppv4(0)(20),ppv4(1)(20),ppv4(2)(20),ppv5(0)(20),ppv5(1)(21));
ppv5(2)(20):=ppv4(3)(20);

u3_21:csa_fa(ppv4(0)(21),ppv4(1)(21),ppv4(2)(21),ppv5(0)(21),ppv5(1)(22));
u3_22:csa_fa(ppv4(0)(22),ppv4(1)(22),ppv4(2)(22),ppv5(0)(22),ppv5(1)(23));
u3_23:csa_fa(ppv4(0)(23),ppv4(1)(23),ppv4(2)(23),ppv5(0)(23),ppv5(1)(24));
u3_24:csa_fa(ppv4(0)(24),ppv4(1)(24),ppv4(2)(24),ppv5(0)(24),ppv5(1)(25));
u3_25:csa_ha(ppv4(0)(25),ppv4(1)(25),ppv5(0)(25),ppv5(1)(26));
u3_26:csa_ha(ppv4(0)(26),ppv4(1)(26),ppv5(0)(26),ppv5(1)(27));
u3_27:csa_ha(ppv4(0)(27),ppv4(1)(27),ppv5(0)(27),ppv5(1)(28));
u3_28:csa_ha(ppv4(0)(28),ppv4(1)(28),ppv5(0)(28),ppv5(1)(29));
u3_29:csa_ha(ppv4(0)(29),ppv4(1)(29),ppv5(0)(29),ppv5(1)(30));
u3_30:csa_ha(ppv4(0)(30),ppv4(1)(30),ppv5(0)(30),ppv5(1)(31));
u3_31:csa_ha(ppv4(0)(31),ppv4(1)(31),ppv5(0)(31),ppv5(0)(32));
ppv5(0)(31):=ppv4(0)(31);
ppv5(2)(12):=ppv4(3)(12);
ppv5(2)(20 downto 14):=ppv4(3)(20 downto 14);

----------STAGE 4----------

ppv6(0)(11 downto 1):= ppv5(0)(11 downto 1);
ppv6(1)(1):=ppv5(1)(1);
ppv6(1)(4):=ppv5(1)(4);
ppv6(1)(6):=ppv5(1)(6);
ppv6(1)(7):=ppv5(1)(7);
ppv6(1)(11 downto 9):= ppv5(1)(11 downto 9);
ppv6(1)(12):='0';
```

```
u4_1_12:csa_fa(ppv5(0)(12),ppv5(1)(12),ppv5(2)(12),ppv6(0)(12),ppv6(1)(13));
u4_1_13:csa_ha(ppv5(0)(13),ppv5(1)(13),ppv6(0)(13),ppv6(1)(14));
u4_1_14:csa_fa(ppv5(0)(14),ppv5(1)(14),ppv5(2)(14),ppv6(0)(14),ppv6(1)(15));
u4_1_15:csa_fa(ppv5(0)(15),ppv5(1)(15),ppv5(2)(15),ppv6(0)(15),ppv6(1)(16));
u4_1_16:csa_fa(ppv5(0)(16),ppv5(1)(16),ppv5(2)(16),ppv6(0)(16),ppv6(1)(17));
u4_1_17:csa_fa(ppv5(0)(17),ppv5(1)(17),ppv5(2)(17),ppv6(0)(17),ppv6(1)(18));
u4_1_18:csa_fa(ppv5(0)(18),ppv5(1)(18),ppv5(2)(18),ppv6(0)(18),ppv6(1)(19));
u4_1_19:csa_fa(ppv5(0)(19),ppv5(1)(19),ppv5(2)(19),ppv6(0)(19),ppv6(1)(20));
u4_1_20:csa_fa(ppv5(0)(20),ppv5(1)(20),ppv5(2)(20),ppv6(0)(20),ppv6(1)(21));
u4_1_21:csa_ha(ppv5(0)(21),ppv5(1)(21),ppv6(0)(21),ppv6(1)(22));
u4_1_22:csa_ha(ppv5(0)(22),ppv5(1)(22),ppv6(0)(22),ppv6(1)(23));
u4_1_23:csa_ha(ppv5(0)(23),ppv5(1)(23),ppv6(0)(23),ppv6(1)(24));
u4_1_24:csa_ha(ppv5(0)(24),ppv5(1)(24),ppv6(0)(24),ppv6(1)(25));
u4_1_25:csa_ha(ppv5(0)(25),ppv5(1)(25),ppv6(0)(25),ppv6(1)(26));
u4_1_26:csa_ha(ppv5(0)(26),ppv5(1)(26),ppv6(0)(26),ppv6(1)(27));
u4_1_27:csa_ha(ppv5(0)(27),ppv5(1)(27),ppv6(0)(27),ppv6(1)(28));
u4_1_28:csa_ha(ppv5(0)(28),ppv5(1)(28),ppv6(0)(28),ppv6(1)(29));
u4_1_29:csa_ha(ppv5(0)(29),ppv5(1)(29),ppv6(0)(29),ppv6(1)(30));
u4_1_30:csa_ha(ppv5(0)(30),ppv5(1)(30),ppv6(0)(30),ppv6(1)(31));
u4_1_31:csa_ha(ppv5(0)(31),ppv5(1)(31),ppv6(0)(31),ppv6(1)(32));

ppv6(0)(32):=ppv5(0)(32);
ppv6(1)(2):='0';
ppv6(1)(3):='0';
ppv6(1)(5):='0';
ppv6(1)(8):='0';
----------------------------STAGE 5-------------------------
ppv7(0):=ppv6(0);
ppv7(1):=ppv6(1);
ci3:= -(((2**16)-1)/3);
cv3 := std_logic_vector(conv_unsigned(ci3,16));

ppv7(2)(32 downto 17):=cv3;
ppv8(0)(16 downto 1):= ppv7(0)(16 downto 1);
ppv8(1)(16 downto 1):= ppv7(1)(16 downto 1);
ppv8(1)(17):='0';

u_F_17:csa_fa(ppv7(0)(17),ppv7(1)(17),ppv7(2)(17),ppv8(0)(17),ppv8(1)(18));
u_F_18:csa_fa(ppv7(0)(18),ppv7(1)(18),ppv7(2)(18),ppv8(0)(18),ppv8(1)(19));
u_F_19:csa_fa(ppv7(0)(19),ppv7(1)(19),ppv7(2)(19),ppv8(0)(19),ppv8(1)(20));
u_F_20:csa_fa(ppv7(0)(20),ppv7(1)(20),ppv7(2)(20),ppv8(0)(20),ppv8(1)(21));
u_F_21:csa_fa(ppv7(0)(21),ppv7(1)(21),ppv7(2)(21),ppv8(0)(21),ppv8(1)(22));
u_F_22:csa_fa(ppv7(0)(22),ppv7(1)(22),ppv7(2)(22),ppv8(0)(22),ppv8(1)(23));
u_F_23:csa_fa(ppv7(0)(23),ppv7(1)(23),ppv7(2)(23),ppv8(0)(23),ppv8(1)(24));
u_F_24:csa_fa(ppv7(0)(24),ppv7(1)(24),ppv7(2)(24),ppv8(0)(24),ppv8(1)(25));
u_F_25:csa_fa(ppv7(0)(25),ppv7(1)(25),ppv7(2)(25),ppv8(0)(25),ppv8(1)(26));
```

```
        u_F_26:csa_fa(ppv7(0)(26),ppv7(1)(26),ppv7(2)(26),ppv8(0)(26),ppv8(1)(27));
        u_F_27:csa_fa(ppv7(0)(27),ppv7(1)(27),ppv7(2)(27),ppv8(0)(27),ppv8(1)(28));
        u_F_28:csa_fa(ppv7(0)(28),ppv7(1)(28),ppv7(2)(28),ppv8(0)(28),ppv8(1)(29));
        u_F_29:csa_fa(ppv7(0)(29),ppv7(1)(29),ppv7(2)(29),ppv8(0)(29),ppv8(1)(30));
        u_F_30:csa_fa(ppv7(0)(30),ppv7(1)(30),ppv7(2)(30),ppv8(0)(30),ppv8(1)(31));
        u_F_31:csa_fa(ppv7(0)(31),ppv7(1)(31),ppv7(2)(31),ppv8(0)(31),ppv8(1)(32));
        u_F_32:csa_fa(ppv7(0)(32),ppv7(1)(32),ppv7(2)(32),ppv8(0)(32),nc1);
      rca(ppv8(0),ppv8(1),ppv10);
  p_prod0(31 downto 0)<=ppv10(32 downto 1);
end process PPS_REDN;
--------------          PIPELINING LOGIC   (six stages)        ------------------------------
PIPELINING : process(clk)
begin
      if clk'event and clk='1'then
        p_prod1<=p_prod0;
        p_prod2<=p_prod1;
        p_prod3<=p_prod2;
        p_prod4<=p_prod3;
        p_prod5<=p_prod4;
        prod<=p_prod5;
      end if;
end process PIPELINING;

end Behavioral;
```