Fall 2007

Generic algorithms and NULL Convention Logic hardware implementation for unsigned and signed quad-rail multiplication

Samarsen Reddy Mallepalli

Department:

GENERIC ALGORITHMS AND NULL CONVENTION LOGIC HARDWARE IMPLEMENTATION FOR UNSIGNED AND SIGNED QUAD-RAIL MULTIPLICATION

by

SAMARSEN REDDY MALLEPALLI

A THESIS

Presented to the Faculty of the Graduate School of the

UNIVERSITY OF MISSOURI-ROLLA

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

2007

_____	_____
Dr. Scott C. Smith, Advisor	Dr. Waleed K. Al-Assadi

_____

Dr. Daryl G. Beetner

ABSTRACT

This thesis focuses on designing generic quad-rail arithmetic circuits, such as signed and unsigned multipliers and Multiply and Accumulate (MAC) units, using the asynchronous delay-insensitive NULL Convention Logic (NCL) paradigm. This work helps to build a library of reusable components to be used for automated NCL circuit synthesis, which will aid in the integration of asynchronous design paradigms into the semiconductor industry.

First, an array structured partial product summation algorithm for quad-rail logic was developed. A number of NCL quad-rail adders required for this implementation were designed at the gate level; and the algorithm was implemented in hardware to design a generic VHDL implementation of an unsigned MAC, where the lengths of the multiplier, multiplicand, and accumulator are specified as generic constants. A number of different sized MACs were instantiated and tested with exhaustive VHDL testbenches, proving that the generic MAC is functionally correct.

Second, a partial product generation algorithm for 2^s Complement quad-rail logic was developed. New NCL quad-rail partial product generation components and adders required for this implementation were designed at the gate level; and the algorithm was implemented in hardware to design a generic VHDL implementation of a 2^s Complement multiplier, where the lengths of the multiplier and multiplicand are specified as generic constants. A number of different sized multipliers were instantiated and tested with exhaustive VHDL testbenches, proving that the generic multiplier is functionally correct.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Scott. C. Smith, for his guidance and continued advice and financial support throughout the research and preparation of this thesis. I would also like to express my gratitude to Dr. Waleed K. Al-Assadi and Dr. Daryl Beetner for serving on my thesis committee. I would like to thank Srikanth Kotla and Ravi Sankar Parameswaran Nair for helping me with the documentation work. Furthermore, I would like to thank my parents and friends for their continued encouragement and blessing toward achieving my goal.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Page

LIST OF TABLES

Page

1. INTRODUCTION

1.1. OBJECTIVE

This M.S. thesis is intended to familiarize the reader with the aysnchronous NULL Convention Logic (NCL) design paradigm for digital circuits and to develop arithmetic algorithms, along with their gate-level structural VHDL hardware implementation, for design of generic quad-rail NCL multipliers. The development of arithmetic circuits using NCL quad-rail signals will complement the work already done using NCL dual-rail signals. This will help in building a reusable design library that can be used for automated NCL synthesis thus assisting in the integration of NULL Convention Logic asynchronous design into the semiconductor industry.

1.2. ASYNCHRONOUS CIRCUITS

Synchronous design using a global clock has been the most prominent methodology for designing digital integrated circuits for the past few decades. However, with deep sub-micron technology's shrinking feature sizes, clock distribution, clock skew, power, and EMI are becoming issues of major concern for synchronous design. Therfore, a renewed interest has developed for research in the area of asynchronous design of digital logic circuits. Traditional asynchronous design, which eliminates the use of a clock, suffers from problems such as race-conditions and timing optimization. However, NULL Convention Logic (NCL), a delay-insensitive asynchronous design paradigm, combines control and data information for synchronization, thus eliminating the necessity of referencing time. NCL, being a delay-insensitive paradigm, has no glitch

power, produces less EMI, consumes less power, and allows for easier design re-use, compared to both synchronous and bounded delay asynchronous paradigms [1].

1.3. ARITHMETIC CIRCUITS

Delay-insensitive asynchronous design paradigms like NULL Convention Logic eliminate the use of a clock by combining control and data information in a single path. Stand-alone NCL functional blocks can be used as modules in larger NCL designs without any need for further timing analysis while interfacing, thus reducing design effort. Therefore, building libraries with basic functional blocks such as adders and multipliers facilitates design re-use. This thesis focuses on developing algorithms for a quad-rail array-structured multiply and accumulate unit and 2^s complement multiplier, and designing generic structural VHDL hardware implementations of both circuits.

1.4. OVERVIEW OF NCL

NCL offers a self-timed logic paradigm where control is inherent with each datum. NCL follows the so-called “weak conditions” of Seitz’s delay-insensitive signaling scheme [2]. As with other self-timed logic methods, the NCL paradigm assumes that forks in wires are isochronic [3]. The origins of various aspects of the paradigm, including the NULL (or spacer) logic state from which NCL derives its name, can be traced back to Muller’s work on speed-independent circuits in the 1950s and 1960s [4].

1.4.1. Delay-Insensitivity. NCL uses symbolic completeness of expression [5] to achieve delay-insensitive behavior. A symbolically complete expression is defined as an expression that only depends on the relationships of the symbols present in the expression

without a reference to their time of evaluation. In particular, dual-rail signals, quad-rail signals, or other Mutually Exclusive Assertion Groups (MEAGs) can be used to incorporate data and control information into one mixed signal path to eliminate time reference [6]. A dual-rail signal, *D*, consists of two wires, D^0 and D^1, which may assume any value from the set {DATA0, DATA1, NULL}. The DATA0 state ($D^0 = 1$, $D^1 = 0$) corresponds to a Boolean logic 0, the DATA1 state ($D^0 = 0$, $D^1 = 1$) corresponds to a Boolean logic1, and the NULL state ($D^0 = 0$, $D^1 = 0$) corresponds to the empty set, meaning that the value of *D* is not yet available. The two rails are mutually exclusive, so that both rails can never be asserted simultaneously; this state is defined as an illegal state.

A quad-rail signal, *Q*, consists of four wires, Q^0, Q^1, Q^2, and Q^3, which may assume any value from the set {DATA0, DATA1, DATA2, DATA3, NULL}. The DATA0 state ($Q^0 = 1$, $Q^1 = 0$, $Q^2 = 0$, $Q^3 = 0$) corresponds to two Boolean logic signals, *X* and *Y*, where X = 0 and Y = 0. The DATA1 state ($Q^0 = 0$, $Q^1 = 1$, $Q^2 = 0$, $Q^3 = 0$) corresponds to X = 0 and Y = 1. The DATA2 state ($Q^0 = 0$, $Q^1 = 0$, $Q^2 = 1$, $Q^3 = 0$) corresponds to X = 1 and Y = 0. The DATA3 state ($Q^0 = 0$, $Q^1 = 0$, $Q^2 = 0$, $Q^3 = 1$) corresponds to X = 1 and Y = 1, and the NULL state ($Q^0 = 0$, $Q^1 = 0$, $Q^2 = 0$, $Q^3 = 0$) corresponds to the empty set meaning that the result is not yet available. The four rails of a quad-rail NCL signal are mutually exclusive, so no two rails can ever be asserted simultaneously; these states are defined as illegal states. Both dual-rail and quad-rail signals are space optimal 1-out-of-N delay-insensitive codes, requiring two wires per bit. Other higher order MEAGs may not be wire count optimal; however, they can be more power efficient due to the decreased number of transitions per cycle.

1.4.2. Logic Gates. NCL differs from many other delay-insensitive paradigms in that these other paradigms only utilize one type of state-holding gate, the C-element [4]. A C-element behaves as follows: when all inputs assume the same value then the output assumes this value, otherwise the output does not change. On the other hand, all NCL gates are state-holding. Thus, NCL optimization methods can be considered as a subclass of the techniques for developing delay-insensitive circuits using a pre-defined set of more complex components, with built-in hysteresis behavior.

NCL uses threshold gates for its basic logic elements [7]. The primary type of threshold gate is the THmn gate, where $1 \leq m \leq n$, as depicted in Figure 1.1. THmn gates have *n* inputs. At least *m* of the *n* inputs must be asserted before the output will become asserted. Because NCL threshold gates are designed with hysteresis, all asserted inputs must be de-asserted before the output will be de-asserted. Hysteresis ensures a complete transition of inputs back to NULL before asserting the output associated with the next wavefront of input data. Therefore, a THnn gate is equivalent to an n-input C-element and a TH1n gate is equivalent to an n-input OR gate. In a THmn gate, each of the *n* inputs is connected to the rounded portion of the gate; the output emanates from the pointed end of the gate; and the gate's threshold value, *m*, is written inside of the gate.

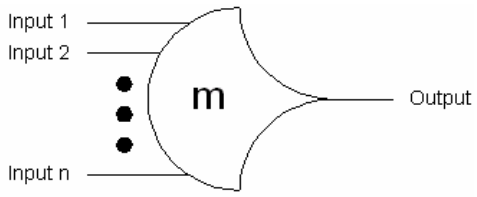


Figure 1.1. THmn threshold gate

Another type of threshold gate is referred to as a weighted threshold gate, denoted as $THmnWw_1w_2\ldots w_R$. Weighted threshold gates have an integer value, $m \geq w_R > 1$, applied to $input_R$. Here $1 \leq R < n$; where *n* is the number of inputs; *m* is the gate's threshold; and w_1, w_2, $\ldots w_R$, are the integer weights of $input_1$, $input_2$, $\ldots$ $input_R$, respectively. For example, consider a TH34W2 gate, whose n = 4 inputs are labeled *A*, *B*, *C*, and *D*. The weight of input *A*, W(A), is therefore 2. Since the gate's threshold, *m*, is 3, this implies that in order for the output to be asserted, either inputs *B*, *C*, and *D* must all be asserted, or input *A* must be asserted and any other input, *B*, *C*, or *D* must also be asserted. NCL threshold gates may also include a reset input to initialize the gate's output. Resetable gates are denoted by either a *D* or an *N* appearing inside the gate, along with the gate's threshold, referring to the gate being reset to logic 1 or logic 0, respectively.

Table 1.1 lists the 27 fundamental NCL gates, along with their corresponding Boolean equations, used to construct NCL circuits. These 27 gates constitute the set of all functions consisting of four or fewer variables. Since each rail of a NCL signal is considered a separate variable, a four variable function is not the same as a function of four literals, which would normally consist of eight variables.

Twenty four of these gates can be realized using complex threshold gates, identical to the standard threshold gate forms for functions of four or fewer variables [17, 18, 19]. The other three macros could be constructed from threshold gate networks, but have been implemented as standard gates to provide completeness. Table 1.1 also contains the transistor count for the static and semi-static implementaion of these 27 gates. The semi-staic implementation results in lesser number of transistors compared to the static implementaion of the gates.

Table 1.1. 27 fundamental NCL gates

NCL Gate	Boolean Function	Transistors (static)	Transistors (semi-static)
TH12	A + B	6	6
TH22	AB	12	8
TH13	A + B + C	8	8
TH23	AB + AC + BC	18	12
TH33	ABC	16	10
TH23w2	A + BC	14	10
TH33w2	AB + AC	14	10
TH14	A + B + C + D	10	10
TH24	AB + AC + AD + BC + BD + CD	26	16
TH34	ABC + ABD + ACD + BCD	24	16
TH44	ABCD	20	12
TH24w2	A + BC + BD + CD	20	14
TH34w2	AB + AC + AD + BCD	22	15
TH44w2	ABC + ABD + ACD	23	15
TH34w3	A + BCD	18	12
TH44w3	AB + AC + AD	16	12
TH24w22	A + B + CD	16	12
TH34w22	AB + AC + AD + BC + BD	22	14
TH44w22	AB + ACD + BCD	22	14
TH54w22	ABC + ABD	18	12
TH34w32	A + BC + BD	17	12
TH54w32	AB + ACD	20	12
TH44w322	AB + AC + AD + BC	20	14
TH54w322	AB + AC + BCD	21	14
THxor0	AB + CD	20	12
THand0	AB + BC + AD	19	13
TH24comp	AC + BC + AD + BD	18	12

By employing threshold gates for each logic rail, NCL is able to determine the output status without referencing time. Inputs are partitioned into two separate wavefronts, the NULL wavefront and the DATA wavefront. The NULL wavefront consists of all inputs to a circuit being NULL, while the DATA wavefront refers to all inputs being DATA, some combination of DATA0 and DATA1. Initially, all circuit elements are reset to the NULL state. First, a DATA wavefront is presented to the circuit. Once all of the outputs of the circuit transition to DATA, the NULL wavefront is presented to the circuit. Once all of the outputs of the circuit transition to NULL, the next DATA wavefront is presented to the circuit. This DATA/NULL cycle continues

repeatedly. As soon as all outputs of the circuit are DATA, the circuit's result is valid. The NULL wavefront then transitions all of these DATA outputs back to NULL. When they transition back to DATA again, the next output is available. This period is referred to as the DATA-to-DATA cycle time, denoted as T_{DD}, and has an analogous role to the clock period in a synchronous system.

1.4.3. Input Completeness. NCL combinational circuits must ensure input-completeness in order to maintain delay-insensitivity. The completeness of input criterion [5] requires that:

1. all inputs must transition from NULL to DATA before the outputs transition from NULL to DATA, and
2. all inputs must transition from DATA to NULL before the outputs transition from DATA to NULL.

In circuits with multiple outputs, it is accepatable, according to Seitz's weak conditions [2], for some of the outputs to transiton without having a complete input set present as long as all outputs cannot transition before all inputs arrive.

1.4.4. Observability. There is one more condition that must be met to ensure delay-insensitivity for NCL circuits and other delay-insensitive circuits. No orphans may propagate through a gate [8]. An orphan is defined as a wire that transitions during the current DATA wavefront, but is not used in the determination of the output. Orphans are caused by wire forks and can be neglected through the isochronic fork assumption [3], as long as they are not allowed to cross a gate boundary. This observability condition, also referred to as indicatability or stability, ensures that every gate transition is observable at

the output, which means that every gate that transitions is necessary to transition at least one of the outputs.

1.4.5. NCL Registration. NCL systems contain at least two delay-insensitive (DI) registers, one at both the input and at the output. Two adjacent register stages interact through their request and acknowledge signals, K_i and K_o, respectively, to prevent the current DATA wavefront from overwriting the previous DATA wavefront, by ensuring that the two DATA wavefronts are always separated by a NULL wavefront. The acknowledge signals are combined in the Completion Detection circuitry to produce the request signal(s) to the previous register stage. NCL registration is realized through cascaded arrangements of single-bit dual-rail registers or single-signal quad-rail registers, depicted in Figure 1.2. These registers consist of TH22 gates that pass a DATA value at the input only when K_i is request for data (rfd) (i.e., logic 1) and likewise pass NULL only when K_i is request for null (rfn) (i.e., logic 0). They also contain a NOR gate to generate K_o, which is *rfn* when the register output is DATA and *rfd* when the register output is NULL. The registers shown below are reset to NULL, since all TH22 gates are reset to logic 0. However, either register could be instead reset to a DATA value by replacing exactly one of the TH22n gates with a TH22d gate.

An N-bit register stage, comprised of N single-bit dual-rail NCL registers, requires N completion signals, one for each bit. The NCL completion component, shown in Figure 1.3, uses these N K_o lines to detect complete DATA and NULL sets at the output of every register stage and request the next NULL and DATA set, respectively. In full-word completion, the single-bit output of the completion component is connected to all K_i lines of the previous register stage. Since the maximum input threshold gate is the

TH44 gate, the number of logic levels in the completion component for an N-bit register is given by $\lceil \log_4 N \rceil$. Likewise, the completion component for an N-bit quad-rail registration stage requires $\frac{N}{2}$ inputs, and can be realized in a similar fashion using TH44 gates. Figures 1.4 and 1.5 shows the flow of DATA and NULL wavefronts through an NCL combinational circuit (i.e., an AND function) and an arbitrary pipeline stage, respectively.

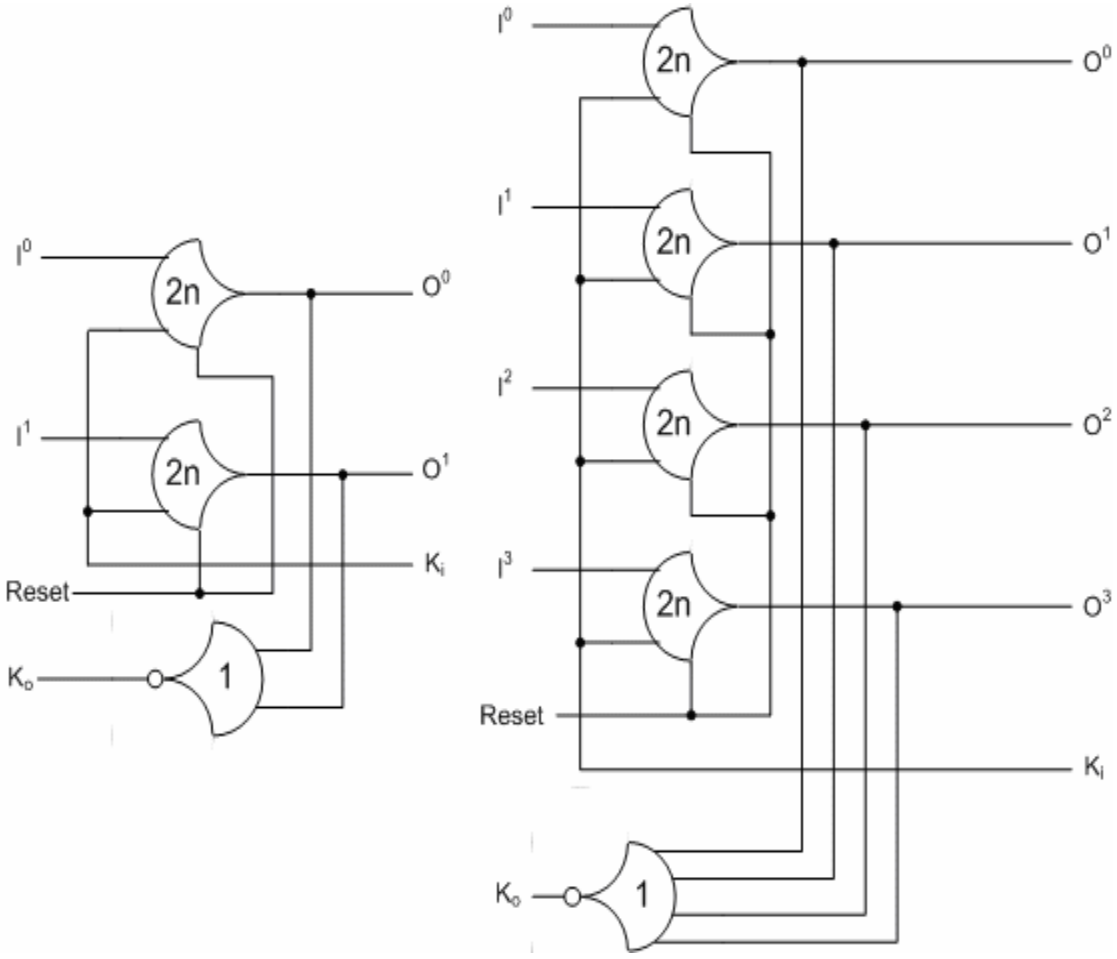


Figure 1.2. Single bit dual-rail register and single signal quad-rail register

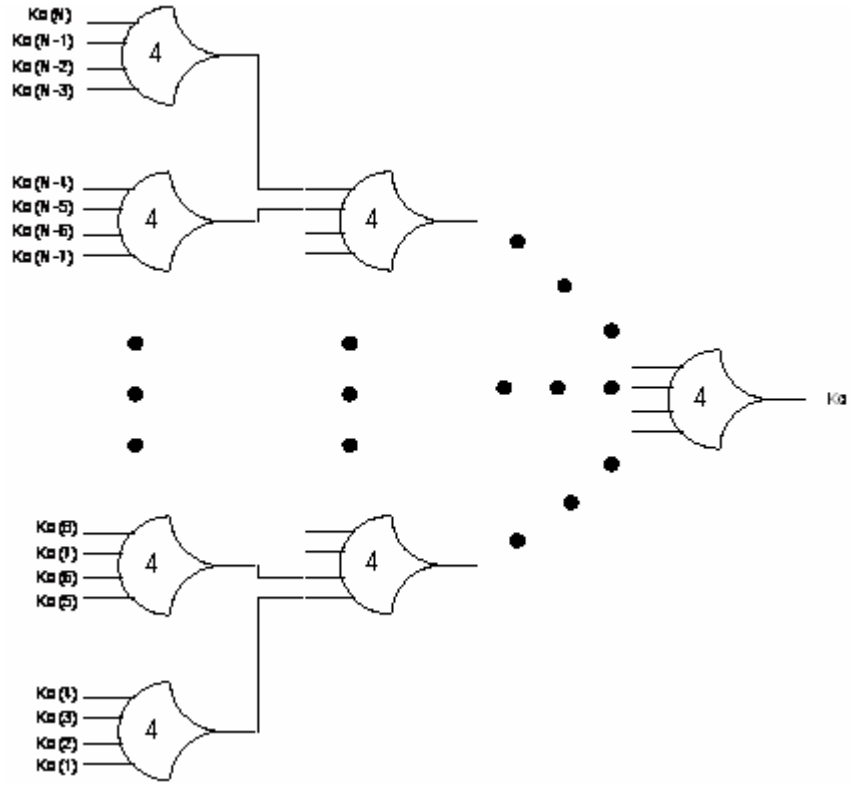


Figure 1.3. N-bit NCL completion component

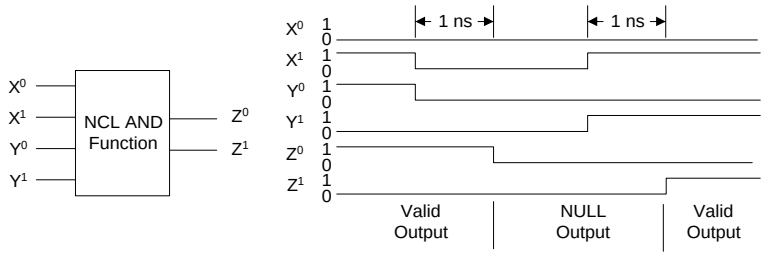
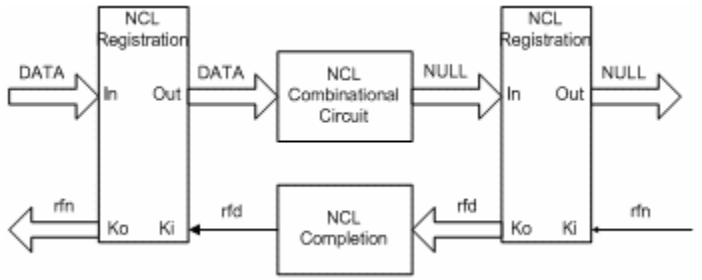
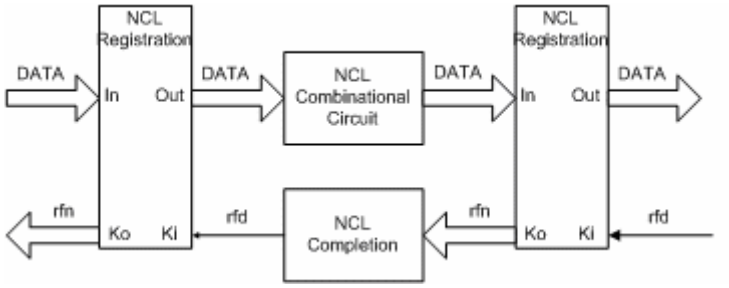


Figure 1.4. NCL AND function: Z = X • Y

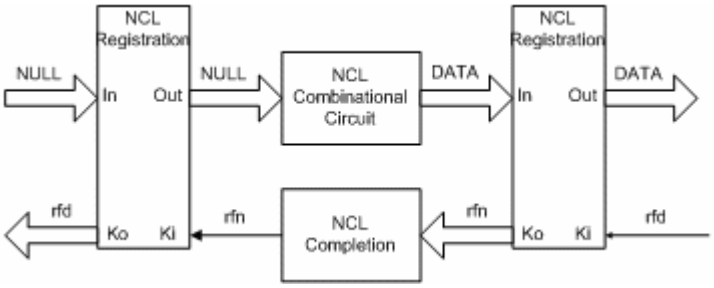
Initially X=DATA1 and Y=DATA0, so Z=DATA0; next X and Y both transition to NULL, so Z transitions to NULL; then X and Y both transition to DATA1, so Z transitions to DATA1.



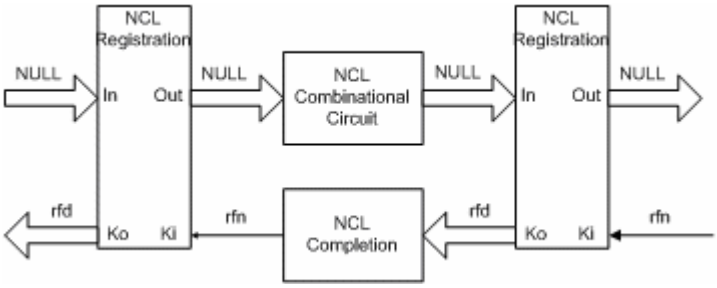
a) DATA flows through input register and combinational circuit



b) DATA flows through output register and rfn flows through completion circuit



c) NULL flows through input register and combinational circuit



d) NULL flows through output register and rfd flows through completion circuit

Figure 1.5. NCL DATA/NULL cycle

1.5. THESIS OVERVIEW

This thesis is organized into five sections. Section 2 reviews the previous work done in the development of NCL arithmetic circuits. In Section 3, a generic algorithm for multiplication of unsigned quad-rail vectors is developed. Partial product generation and various adder components are designed. Carry save addition with array structured partial product summation is used to obtain the multiplication product. Using the generic unsigned quad-rail multiplier and ripple carry accumulator, a multiply and accumulate unit is designed and implemented as a structural VHDL module. Section 4 presents a generic algorithm for quad-rail 2^s complement partial product generation and additional partial product generation and summation components are designed. Using these components and the ones already designed for unsigned multiplication, a generic quad-rail 2^s complement multiplier is designed and implemented as a structural VHDL module.

2. PREVIOUS WORK

Multiplication is an important arithmetic operation that requires a lot of computational effort. A number of efficient multiplication algorithms have been developed and implemented in hardware. Fast and computationally efficient algortihms whose hardware implementation requires less area and consumes less power are desired.

To integrate Asynchronous NULL Convention Logic design into the semiconductor industry, it is required to develop the key components of a reusable-design library, the foremost being arithmetic circuits, including multipliers. This chapter presents the previous work in the field of asynchronous NCL multiplier design.

2.1. DUAL-RAIL NCL MULTIPLIERS

A number of multiplication algorithms have been implemented in NCL using dual-rail signals, including a generic Baugh-Wooley multiplier, an 8×8 Modified Booth2 multiplier [9], a bit-serial multiplier [10], and a number of 72+32×32 Multiply and Accumulate (MAC) units [11, 12]. The Baugh-Wooley multiplier used a carry-save array structure [13] for partial product (PP) summation, while the rest of the multipliers/MACs utilized a Wallace Tree [13].

2.2. QUAD-RAIL NCL MULTIPLIERS

Additionally, unsigned quad-rail multipliers utilizing Wallace Tree PP summation have been designed [9, 14]; however, a Wallace Tree is not a regular structure and therefore cannot be implemented as a generic structure. This thesis addresses this drawback by developing two new quad-rail multiplication algorithms for generic

multipliers: array-structured PP summation and 2^s complement PP generation. These agorithms are then used in generic VHDL implementations of a quad-rail NCL unsigned MAC and 2^s complement multiplier.

3. DESIGN AND IMPLEMENTATION OF A GENERIC UNSIGNED QUAD-RAIL MULTIPLY AND ACCUMULATE UNIT

This section presents the design and VHDL implementation of a generic unsigned NULL Convention Logic (NCL) quad-rail Multiply and Accumulate (MAC) unit. The MAC consists of an unsigned quad-rail multiplier and ripple-carry type accumulator, along with NCL registers and full-word completion circuitry.

3.1. QUAD-RAIL MULTIPLICATION

3.1.1. Partial Product Generation. Since a single quad-rail signal represents two dual-rail or two binary bits, multiplication of two quad-rail signals is equivalent to multiplying two 2-bit binary vectors. The maximum value a quad-rail signal can represent is DATA3 (i.e., Boolean “11”); and the maximum product of two quad-rail signals, when both are DATA3, is Boolean “1001”, which can be represented by two quad-rail signals, partial product high (i.e., PPH = “10”) and partial product low (i.e., PPL = “01”). Since PPH has a maximum value of “10”, it can be represented using a three-rail signal (i.e., DATA0, DATA1, and DATA2) instead of a quad-rail signal, thus eliminating one wire. PPL has values from “00” to “11”, and therefore requires a quad-rail signal. The unsigned quad-rail PP generator, Q33MUL [9, 14], multiplies two quad-rail inputs and generates a three-rail signal, PPH, and a quad-rail signal, PPL.

3.1.2. Unsigned Quad-Rail Multiplication. Unsigned quad-rail multiplication is similar to binary multiplication except that each quad-rail signal represents two binary bits. So two bits in the multiplier are multiplied with two bits in the multiplicand to generate a PP which consists of partial product high and partial product low as explained in Section 3.1.1. Figure 3.1 shows multiplication of two 4-bit binary vectors with their

PPs and final product. Figure 3.2 shows a 2-bit quad-rail by 2-bit quad-rail multiplication where the quad-rail multiplicand represents the 4-bit binary multiplicand and the quad-rail multipier represents the 4-bit binary multiplier of Figure 3.1. Comparing the two figures, it can be seen that the sum of the first two PP rows in the binary multiplication is equivalent to the sum of the 1st PP in the quad-rail multiplication. The same holds true for the sum of the 3rd and 4th PP rows of the binary multiplication and the 2nd PP of the quad-rail multiplication.

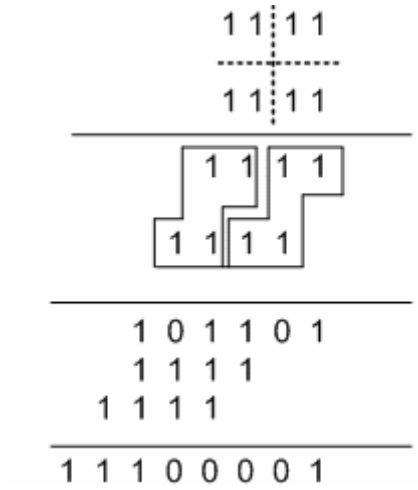


Figure 3.1. Binary multiplication

Multiplying LSB X_0 of the quad-rail multiplier with LSB Y_0 of the quad-rail multiplicand results in a value which is equal to the sum of multiplication of the LSB of the binary multiplier with the last two LSBs of the binary multiplicand and multiplication of the 2nd LSB of the multiplier with the last two LSBs of the multiplicand, with the

second PP row shifted one place to the left. A similar explanation holds true for X_0 multipied with Y_1, X_1 with Y_0, etc. Therefore, for any vector length of quad-rail multiplicand, the sum of the terms in the first PP of a quad-rail multiplication correspond to the sum of the 1st and 2nd PP rows of binary multiplication. The sum of terms in the 2nd PP of quad-rail multiplication correspond to the sum of the 3rd and 4th PP rows of binary multiplication, etc. Thus, one PP in quad-rail multiplication corresponds to two PP rows in binary multiplication. Each subsequent quad-rail multiplication PP is shifted one position to the left, the same as for binary multiplication; however one position to the left in quad-rail multiplication is equivalent to two bit positions, since a quad-rail signal corresponds to two bits. This is correct, since each quad-rail PP corresponds to two binary PPs, and hence, each subsequent PP should be left shifted by two bit positions. The quad-rail PPs can be summed using various algorithms (e.g., array-structured , Wallace tree).

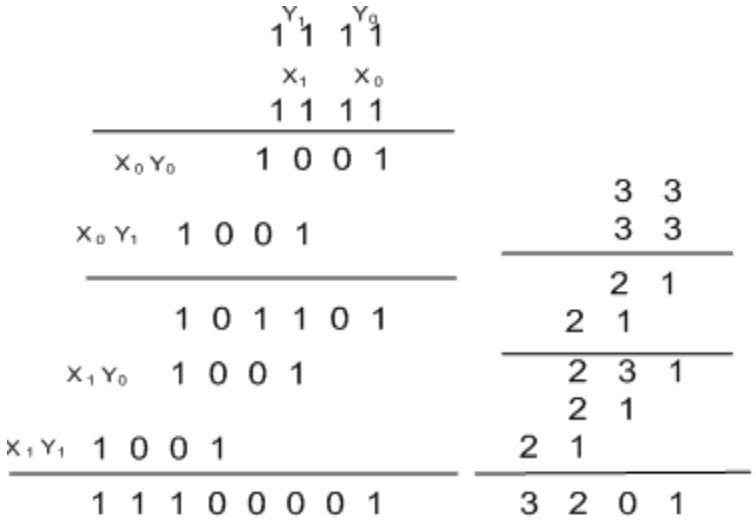


Figure 3.2. Quad-rail multiplication

3.2. MULTIPLY AND ACCUMULATE

A MAC utilizes a multiplier and adds the product of the current inputs to the previously accumulated value, which requires the previously accumulated value to be fed back from the output to add it to the current product. If the accumulator exceeds its maximum value, *OV* (overflow) is asserted. The block diagram of an NCL MAC is shown in Figure 3.3. Any NCL system requires at least two registers, one at the input and and one at the output, and feedback requires at least three registers in the feedback loop to avoid deadlock [15].

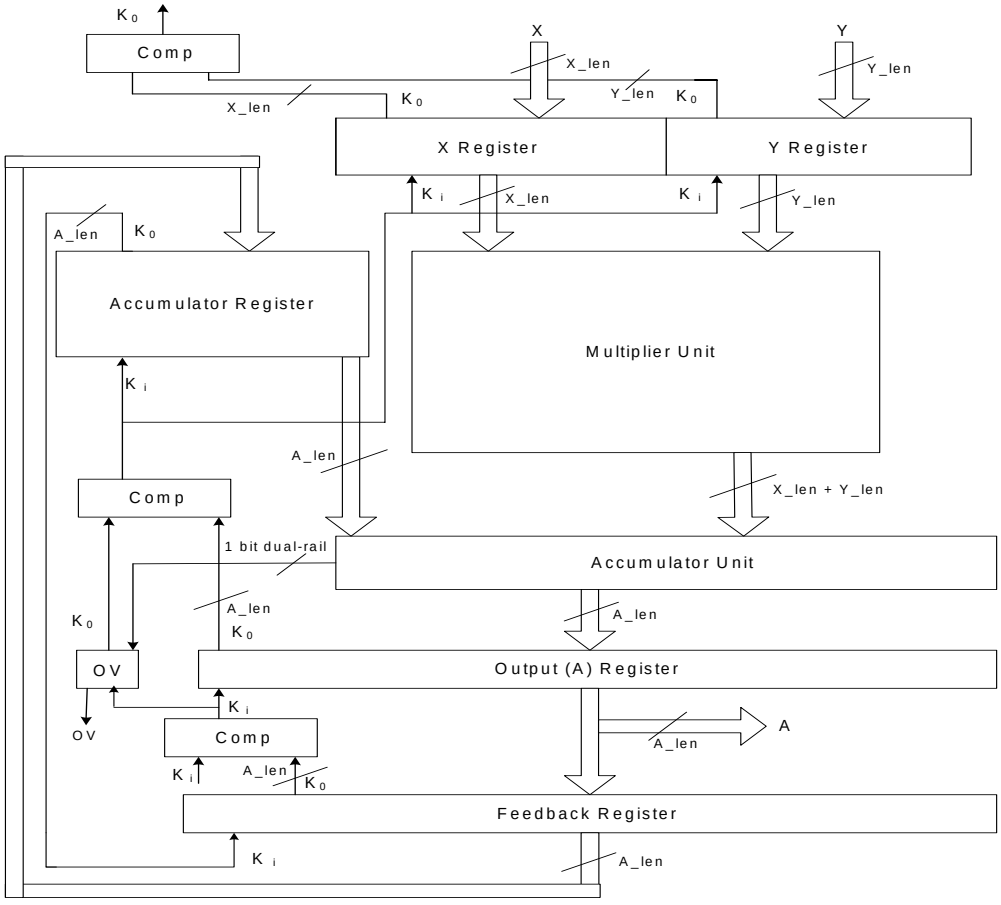


Figure 3.3. MAC block diagram

3.2.1. Registers. In order to avoid deadlock, three registers are used in the MAC feedback loop. All registers are reset to NULL except for the feedback register, which is reset to DATA0 to initialize the accumulator.

3.2.2. Partial Product Generation. Partial products are generated using the Q33mul component, explained in Section 3.1.2 and shown in Figure 3.4, which generates the product of two single unsigned quad-rail signals. For an 8×8 multiplication (i.e., 4×4 quad-rail) the total number of partial products is 4, with each partial product having two terms (i.e., partial product high and partial product low).

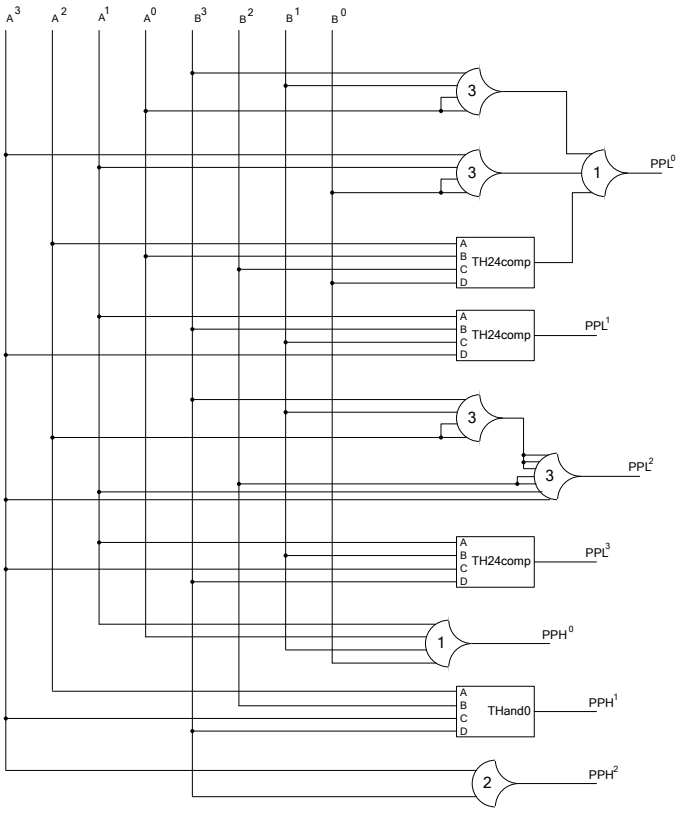


Figure 3.4. Q33mul circuitry [9, 14]

3.2.3. Partial Product Summation. After generating the PPs, they need to be summed to obtain the final product. PPs can be summed with Carry-Save Adders (CSAs) using a Wallace Tree, Dadda Tree, or array structure [13]. While the Wallace and Dadda Tree implementations are faster (i.e., O(Log N) vs. O(N) [13], where *N* is the length of the multiplier), they do not yield a regular structure that can be used to design a generic version of the PP summation hardware, where the multiplier and multiplicand lengths can be varied; whereas the array multiplier is a regular structure. All three PP summation algorithms require approximately the same amount of hardware, proportional to O(N) [13].

Carry-save addition is applied continuously until only a Sum and Carry PP row remain, at which time a Carry-Propogate Adder (CPA) is used to generate the final product. For array-structured multiplication, the CPA is usually implemented using a Ripple-Carry Adder (RCA); whereas some type of Carry-Lookahead Adder (CLA) is normally used with a Wallace Tree or Dadda Tree. This is because a CLA has a worse-case propogation delay of O(Log N); whereas a RCAs worse-case delay is O(N) [13]. However, for asynchronous circuits that operate based on average-case delay, a RCA is preferred since both CLAs and RCAs have O(Log N) average-case delay [13] and RCAs have a regular structure and normally require less hardware.

Hence, for the generic quad-rail MAC designed in this thesis, an array-structured PP summation algorithm was developed and utilized, followed by a RCA for final product generation, as shown in Figures 3.5 and 3.6 for 8-bit operands (i.e., 4 quad-rail signals). Note that the quad-rail PP sumation requires $\frac{N}{2}-1$ levels of carry-save addition (i.e., 3 for the 8×8 case); whereas a binary multiplier requires N-1 levels of carry-save

addition. Figure 3.5 shows the $\frac{M}{2} \times \frac{N}{2}$ PP summation array of adders, where *M* is the length of the multiplicand, in bits. Note that the equivalent binary multiplier requires an (M-1)×N adder array. The design of the specific adders are detailed in Section 3.2.5.

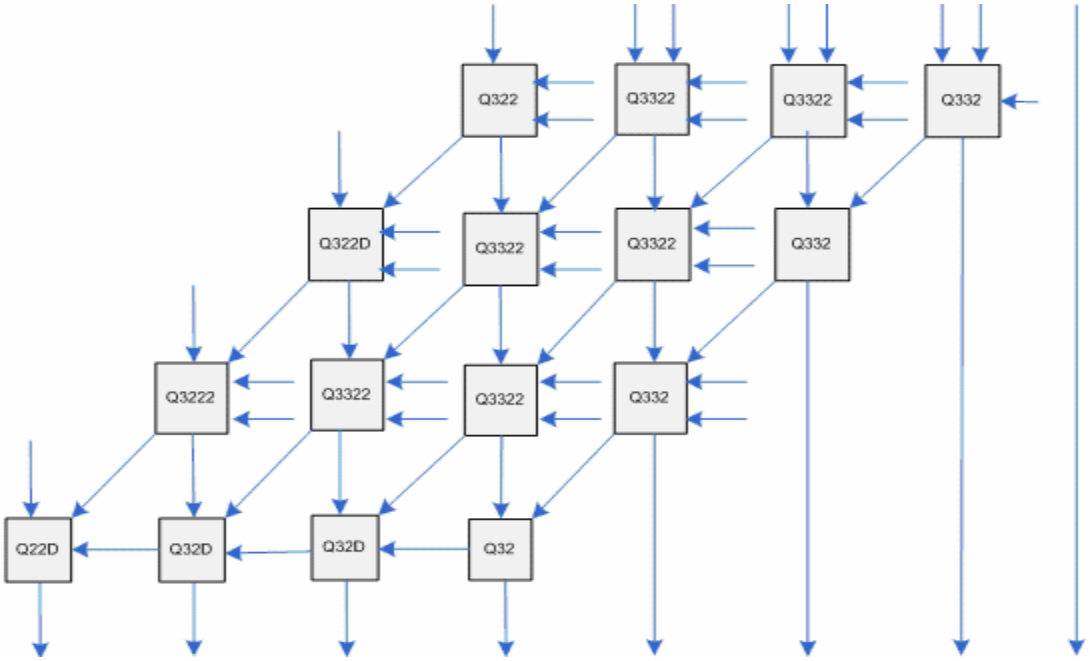


Figure 3.5. Array multiplier

3.2.4. Accumulator. The Accumulator adds the current multiplication product to the previously accumulated value. This is implemented using a RCA; and the output of this block is fed to the output register from which the output of the MAC is obtained, and is also fed back to the Accumulator. Figure 3.6 depicts a quad-rail RCA, which adds a 16-bit (i.e., 8 quad-rail signals) product to a 20-bit (i.e., 10 quad-rail signals) accumulator.

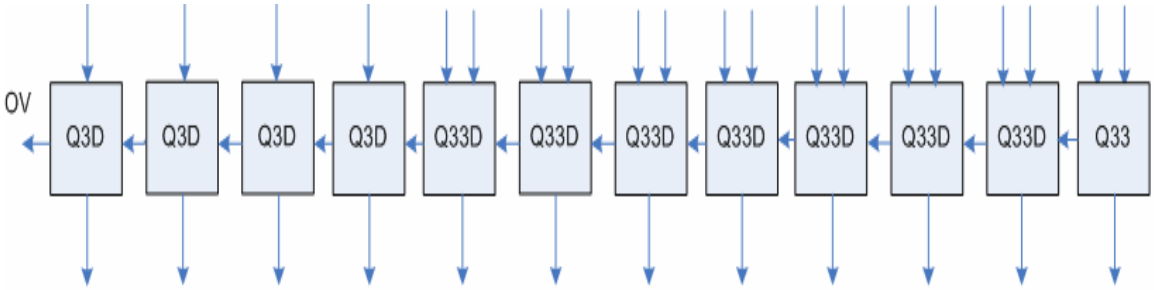
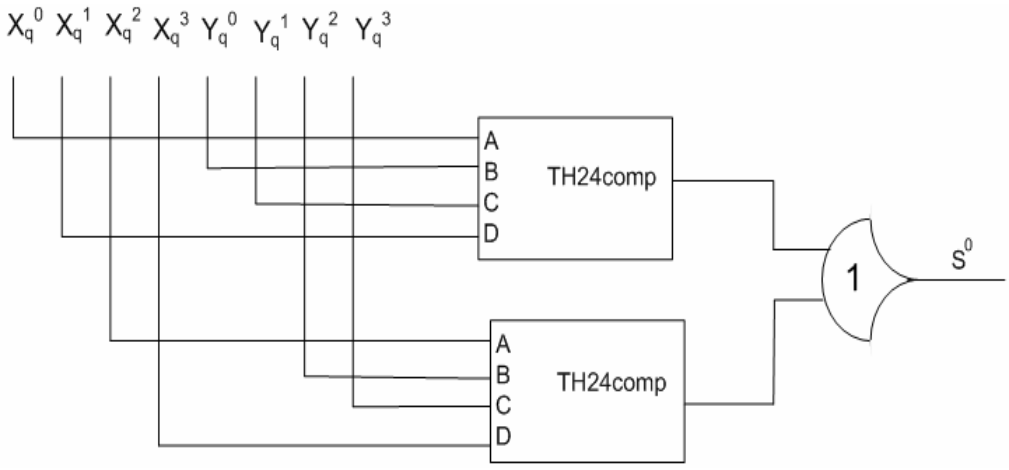


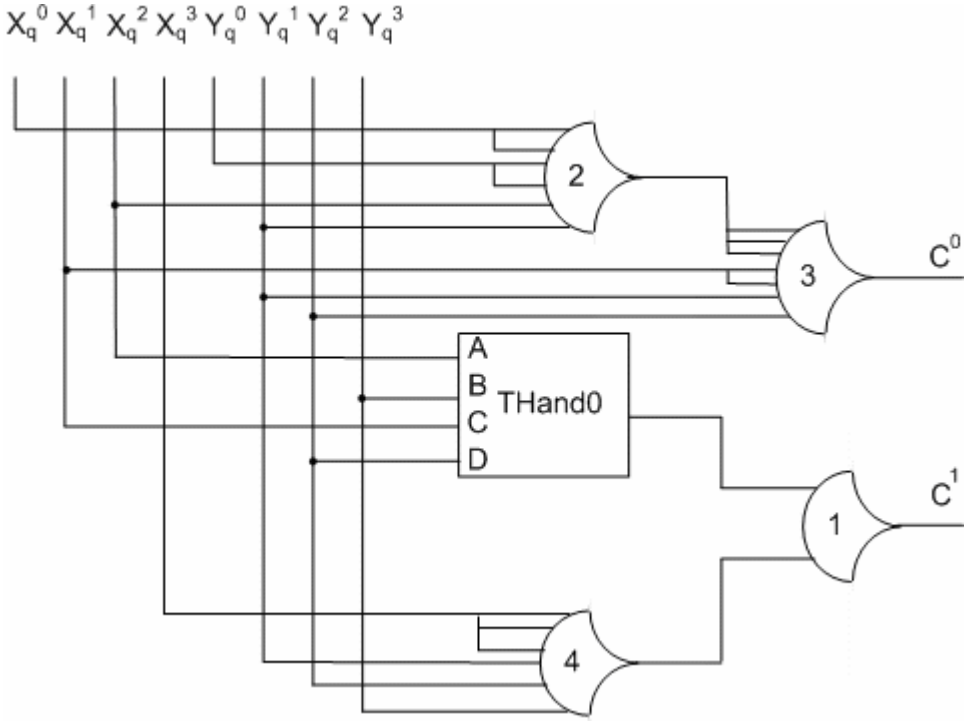
Figure 3.6. Accumulator unit

3.2.5. Adders for Array Multiplier and Accumulator. Since the partial products have both 3-rail MEAG signals (DATA0=”00”, DATA1=”01”, DATA2=”10”) and quad-rail signals, partial product summation stages need special adders which can add a combination of quad-rail, 3-rail and dual-rail signals. Dual-rail signals are produced as carries from some adders whose maximum carry value is ’1’. Following is the list of required adders, along with their functional description and NCL implementation. Adders Q3Dadd, Q33Dadd and Q33add are used in the accumulator unit, while the remaining adders are used in the PP summation of the array multiplier. The circuits were designed following the method detailed in [15,16].

1. Q33add – This adder, shown in Figure 3.7, is used to add two quad-rail signals. *Sum* is a quad-rail signal and *Carry* is a dual-rail signal. Note that the *Sum* output rails are symmetrical, so only rail0 is depicted.This adder is used in the RCA of the accumulator unit in the least significant position. This adder is used only once and is used to add partial product low of the least significant partial product in the first row of partial products to the least significant quad-rail signal of the previously accumlated value.



a) rail0 of the Sum output



b) Carry output

Figure 3.7. Q33add circuitry

2. Q32add – This adder, shown in Figure 3.8, is used to add a quad-rail signal and a three-rail signal. *Sum* is a quad-rail signal and *Carry* is a dual-rail signal.

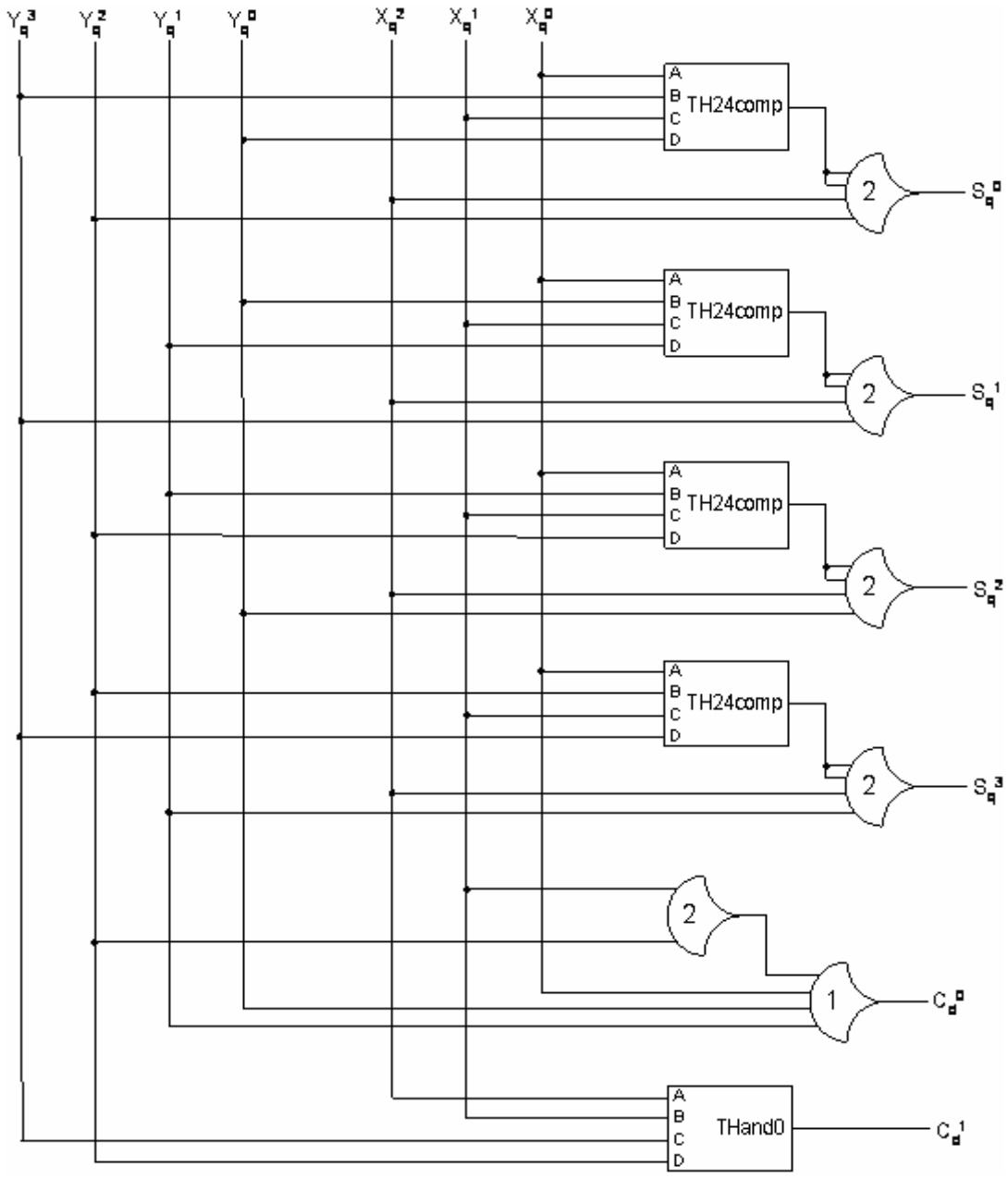
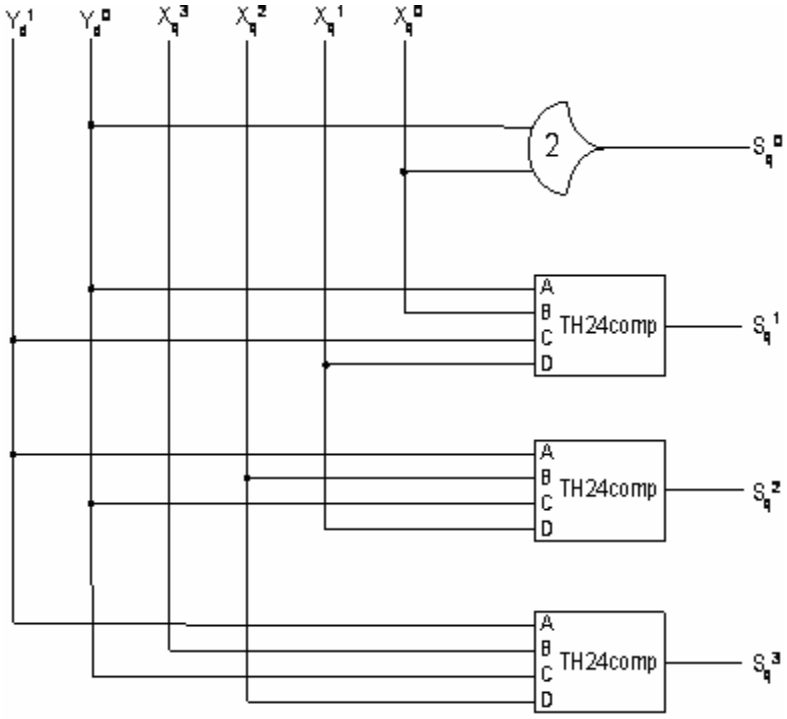
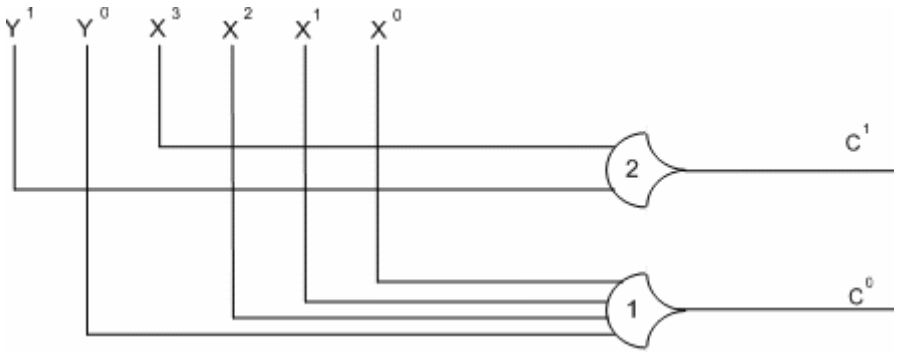


Figure 3.8. Q32add circuitry [9, 14]

3. Q3Dadd – This adder, shown in Figure 3.9, is used to add a quad-rail signal and a dual-rail signal. *Sum* is a quad-rail signal and *Carry* is a dual-rail signal.



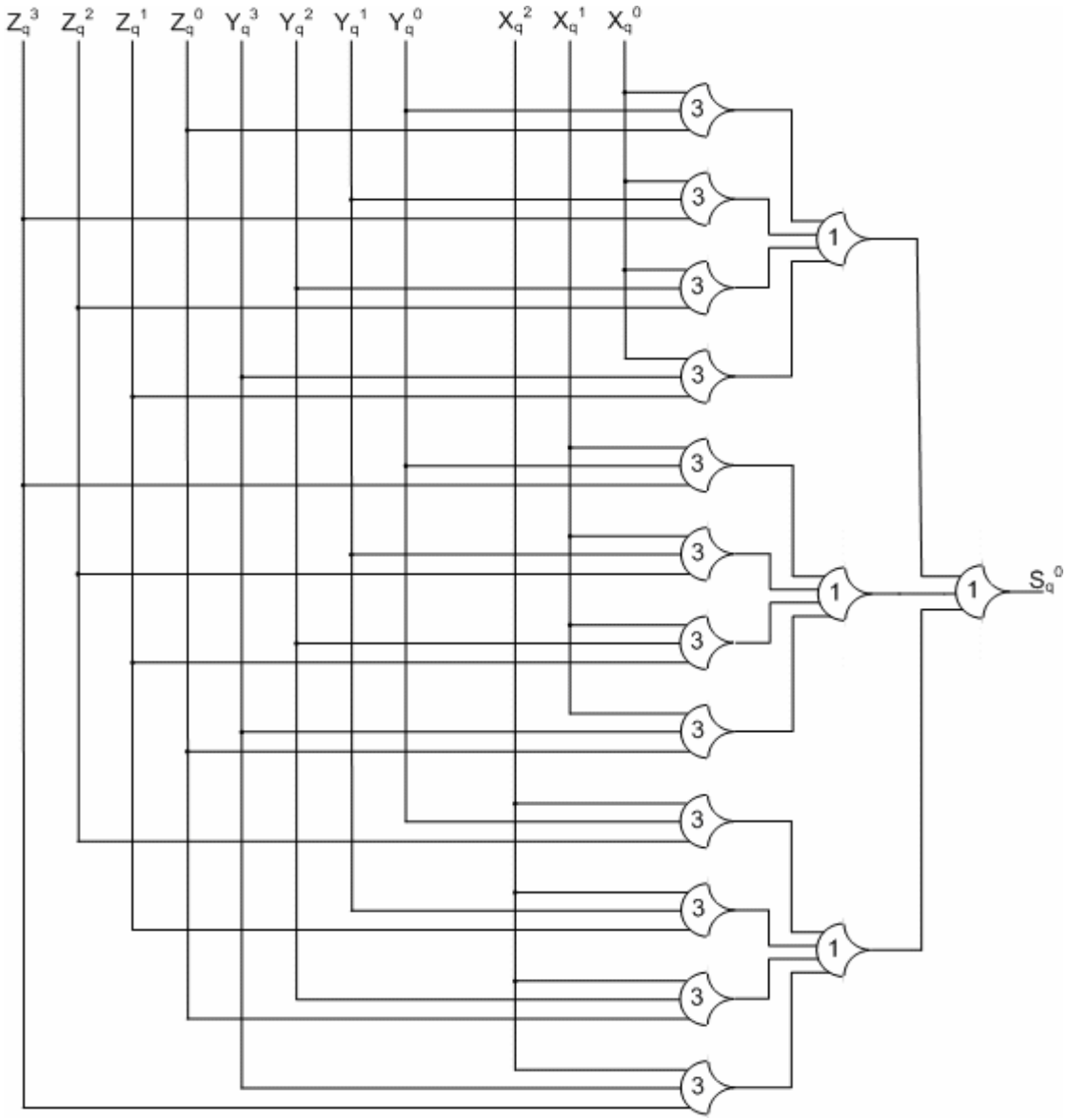
a) Sum output



b) Carry output

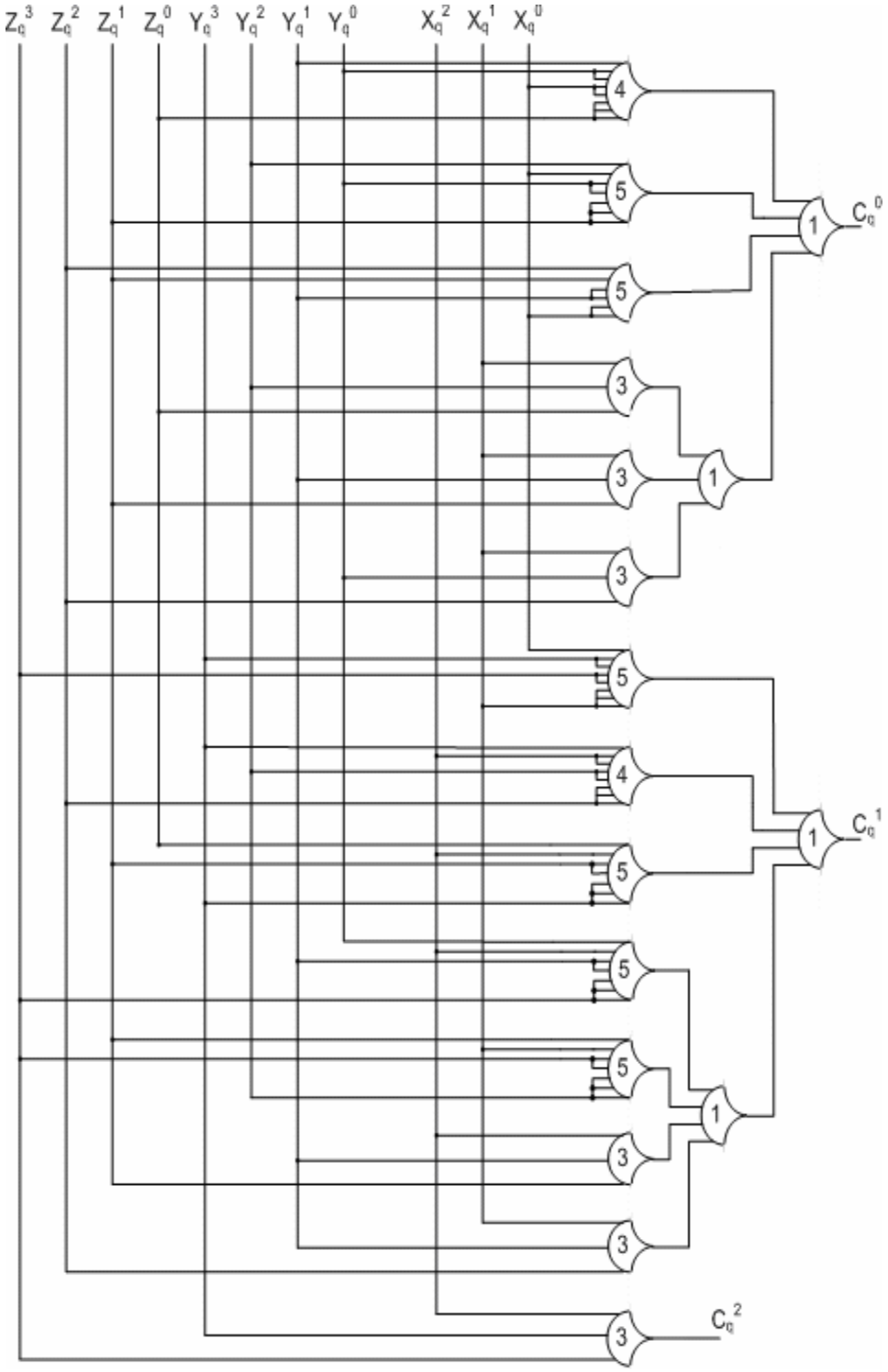
Figure 3.9. Q3Dadd circuitry [9,14]

4. Q332add – This adder, shown in Figure 3.10, adds two quad-rail signals and a three-rail signal. *Sum* is a quad-rail signal and *Carry* is a three-rail signal. Note that the *Sum* output rails are semetrical, so only rail0 is depicted.



a) rail0 of the Sum output

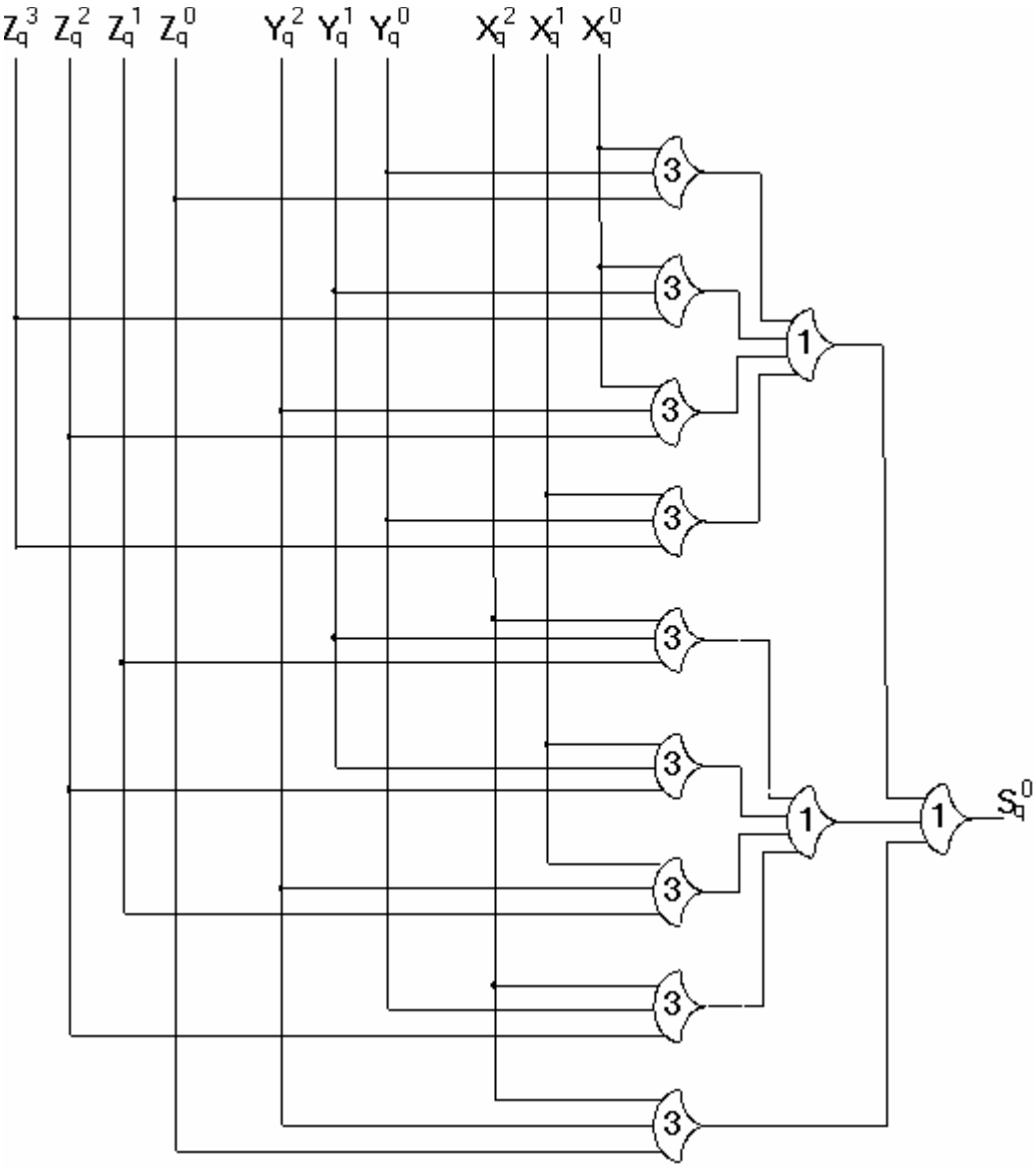
Figure 3.10. Q332add circuitry [9, 14]



b) Carry output

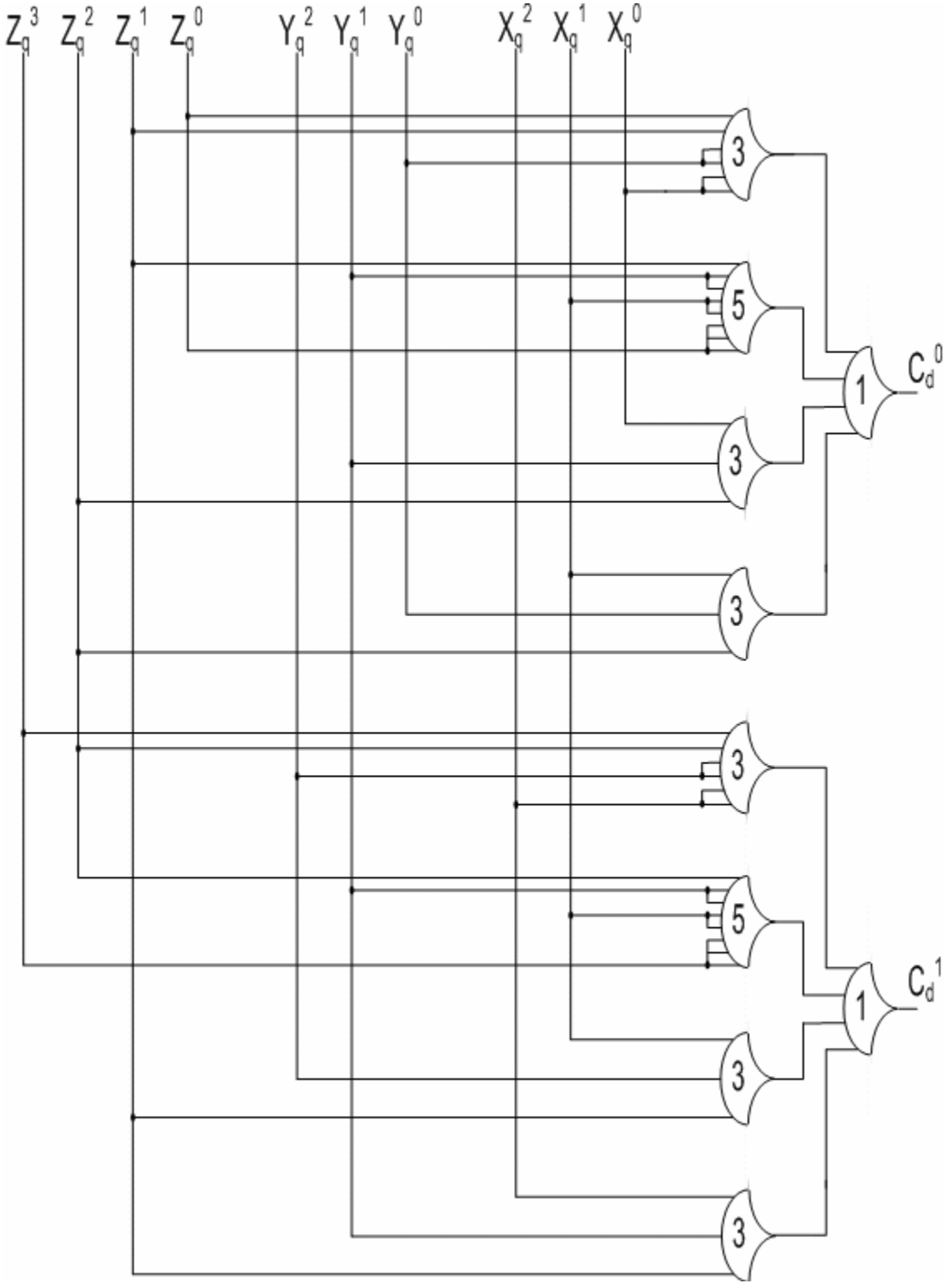
Figure 3.10. Q332add circuitry [9,14] (cont.)

5. Q322add – This adder, shown in Figure 3.11, is used to add a quad-rail signal and two three-rail signals. *Sum* is a quad-rail signal and *Carry* is a dual-rail signal. Note that the *Sum* output rails are symmetrical, so only rail0 is depicted.



a) rail0 of the Sum output

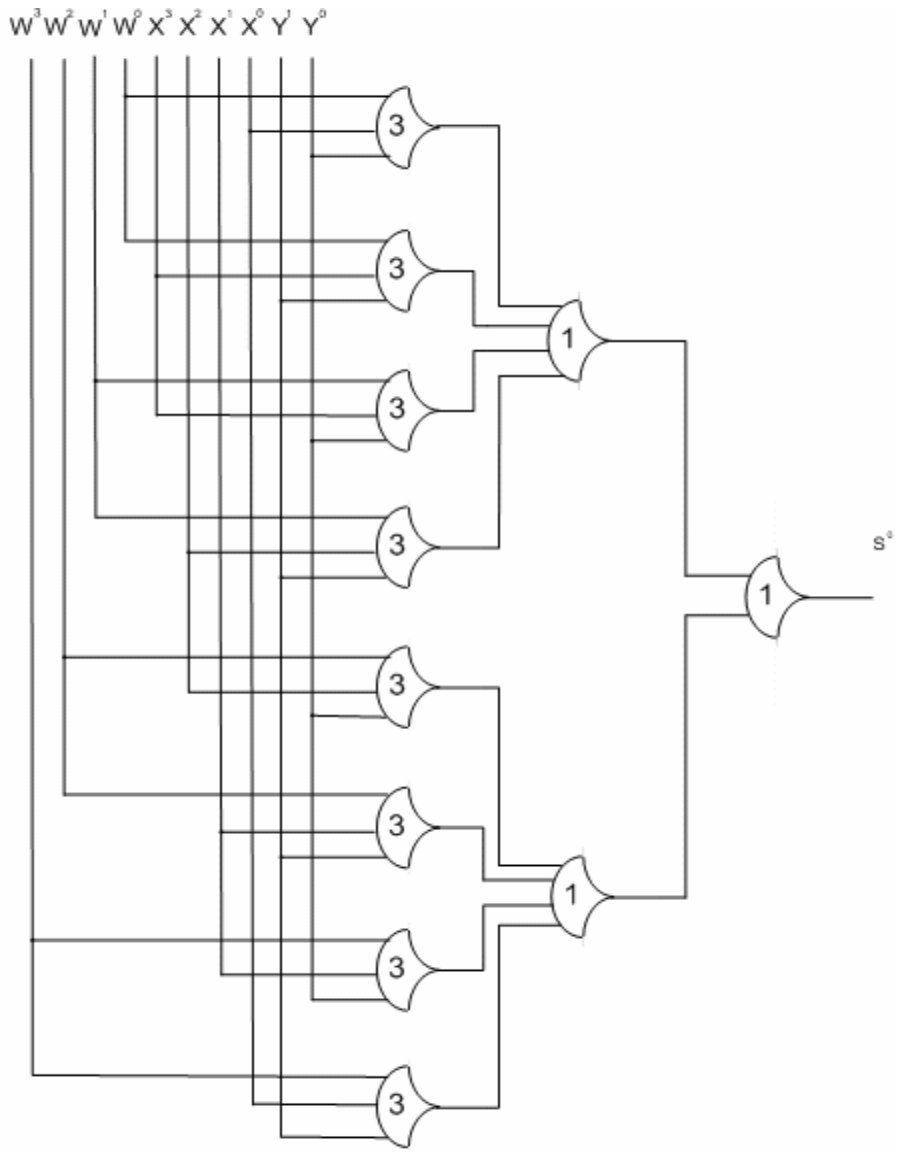
Figure 3.11. Q322add circuitry [9, 14]



b) Carry output

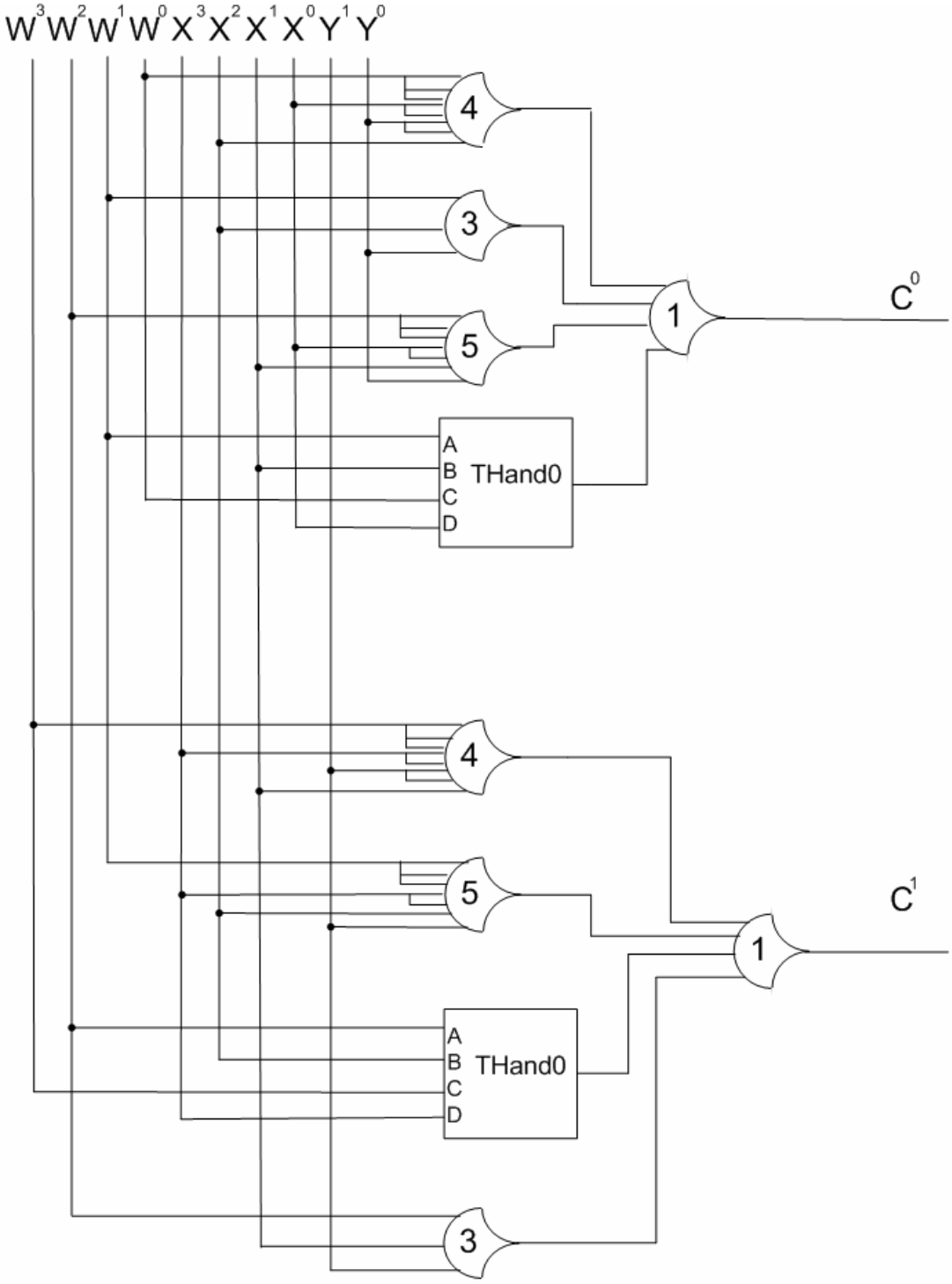
Figure 3.11. Q322add circuitry [9, 14] (cont.)

6. Q33Dadd – This adder, shown in Figure 3.12, adds two quad-rail signals and a dual-rail signal. *Sum* is a quad-rail signal and *Carry* is a dual-rail signal. Note that the *Sum* output rails are semetrical, so only rail0 is depicted.



a) rail0 of the Sum output

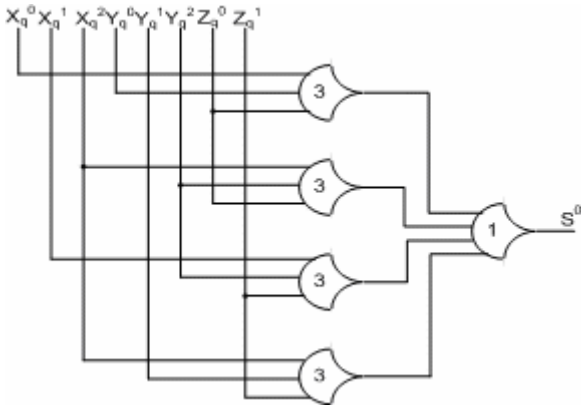
Figure 3.12. Q33Dadd circuitry



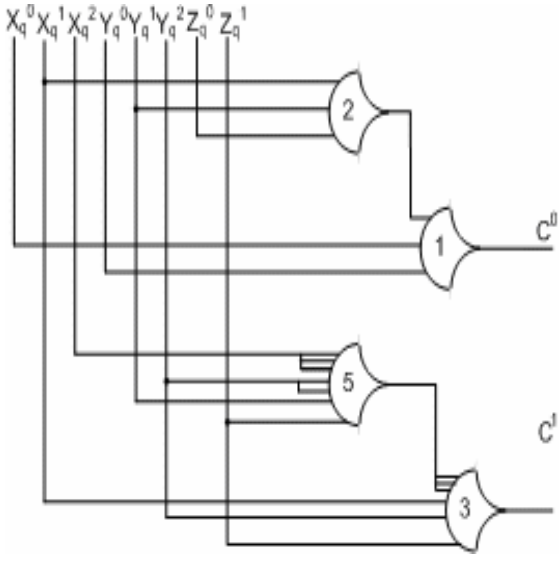
b) Carry output

Figure 3.12. Q33Dadd circuitry (cont.)

7. Q22Dadd – This adder, shown in Figure 3.13, is used to add two three-rail signals and a dual-rail signal. *Sum* is a quad-rail signal and *Carry* is a dual-rail signal. Note that the *Sum* output rails are semetrical, so only rail0 is depicted.



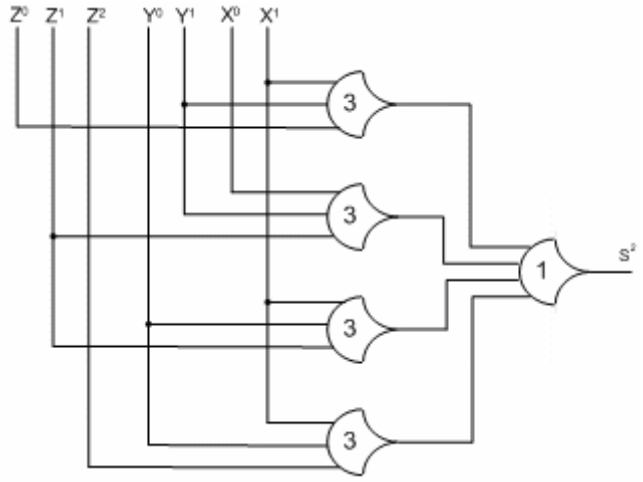
a) rail0 of the Sum output



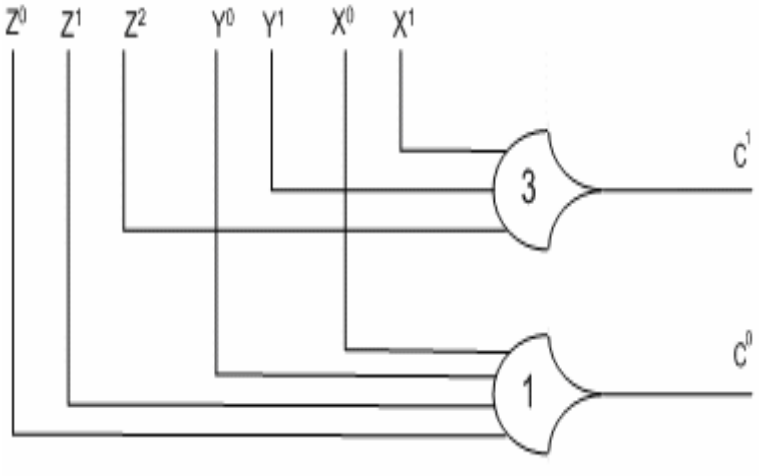
b) Carry output

Figure 3.13. Q22Dadd circuitry

8. Q2DDadd – This adder, shown in Figure 3.14, is used to add two dual-rail signals and a three-rail signal. *Sum* is a quad-rail signal and *Carry* is a dual-rail signal. Note that the *Sum* output rails are semetrical, so only rail2 is depicted.



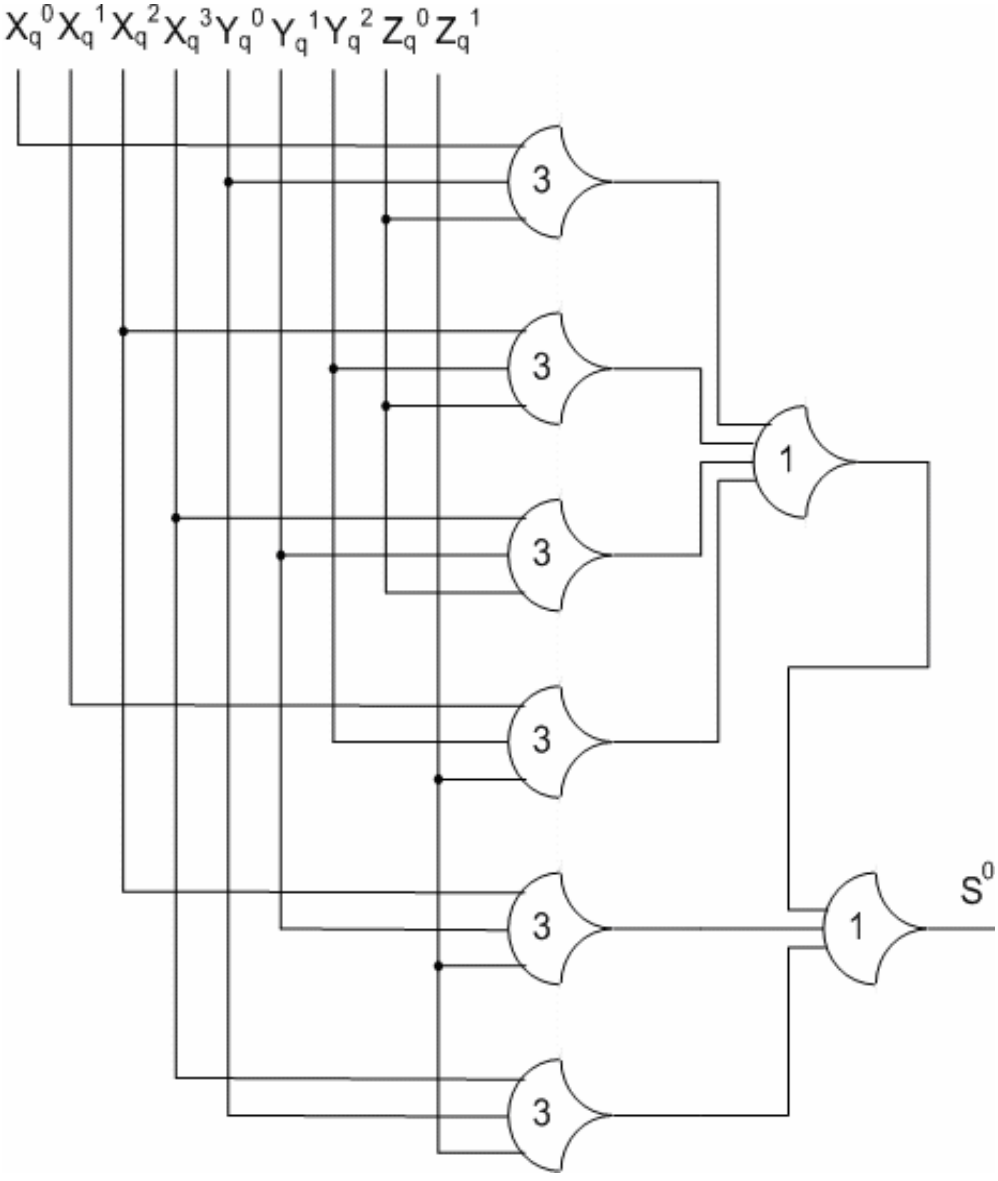
a) rail2 of the Sum output



b) Carry output

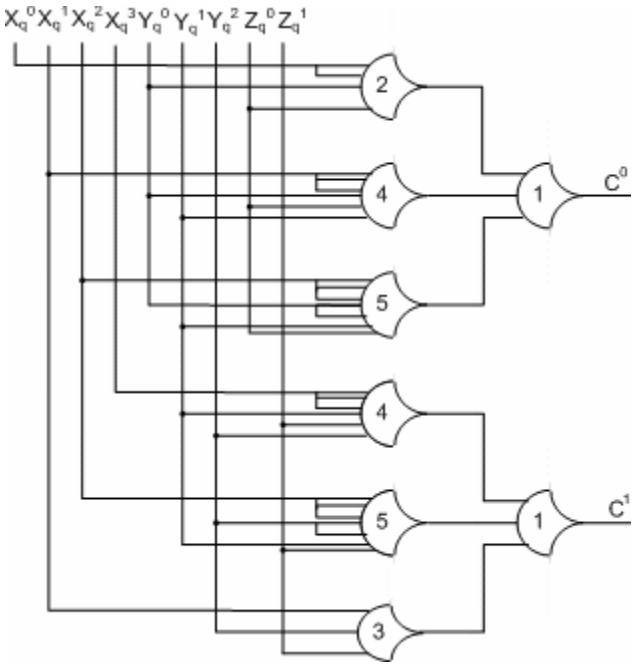
Figure 3.14. Q2DDadd circuitry

9. Q32Dadd – This adder, shown in Figure 3.15, adds a quad-rail signal, a three-rail signal, and a dual-rail signal. *Sum* is a quad-rail signal and *Carry* is a dual-rail signal. Note that the *Sum* output rails are semetrical, so only rail0 is depicted.



a) rail0 of the Sum output

Figure 3.15. Q32Dadd circuitry



b) Carry output

Figure 3.15. Q32Dadd circuitry (cont.)

10. Q322Dadd – This adder is used to add a quad-rail signal, two three-rail signals, and a dual-rail signal. *Sum* is a quad-rail signal and *Carry* is a three-rail signal. The *Sum* and *Carry* have a maximum of 4 and 3 gate delays, respectively, and are not shown due to the large size of the circuit.
11. Q3222add – This adder is used to add a quad-rail signal and three three-rail signals. *Sum* is a quad-rail signal and *Carry* is a three-rail signal. Both the *Sum* and *Carry* have a maximum of 4 gate delays, and are not shown due to the large size of the circuit.
12. Q3322add – This adder is used to add two quad-rail signals and two three-rail signals. *Sum* is a quad-rail signal and *Carry* is a three-rail signal. Both the *Sum*

and *Carry* have a maximum of 4 gate delays, and are not shown due to the large size of the circuit. Gate delays for each adder and partial product generation components is given in the Table 3.1 shown below.

Table 3.1. Gate delays for MAC components

COMPONENT	GATE DELAY	
	SUM/PPL	CARRY/PPH
Q3322	4	4
Q3222	4	4
Q322D	4	3
Q332	3	3
Q322	3	2
Q33D	3	2
Q32D	3	2
Q33	2	2
Q22D	3	2
Q32	2	2
Q3D	1	1
Q33mul	2	1

3.3. VHDL IMPLEMENTATION

The combinational circuitry components of the MAC were designed and implemented as structural, gate-level VHDL modules. Every component was subjected to exhaustive testing using VHDL testbenches specific to the component. These basic components were then used to create a generic MAC unit, where the length of the input registers, the number of Q33mul components and the number of rows and columns in the

array multiplier change according to the length of the input vectors, *X* and *Y*, given as generic constants, and the length of the accumulator RCA, output register, and feedback registers change based on the length of the accumulator output, *A*, also a generic constant.

The functionality of the MAC unit has been verified using exhaustive testbenches for a 12+6×4 (i.e., 6+3×2 in quad-rail) and 8+4×4 MAC, and a 256 testvector testbench for a 72+44×24 MAC. Simulation results confirmed that all of the various sized MACs tested functioned correctly, and showed that the average time per MAC operation increased proportinal to the length of the multiplier, *N* as the MAC size grew larger (i.e., 5.2 ns, 5.9 ns, and 35.6 ns for the 8+4×4, 12+6×4, and 72+44×24 MACs, respectively), as expected. Simulation results for one operation of the 12+6×4 MAC are given in Figure 3.16, showing that $230333_4 + 223_4 \times 21_4 = 303002_4$ (i.e., $2879_{10} + 43_{10} \times 9_{10} = 3266_{10}$), as expected. The complete system diagrams of a 24+8×8 (i.e., 12+4×4 in quad-rail), 22+10×8, and 16+8×8 MAC are shown in Figures 3.17, 3.18, and 3.19, respectively.

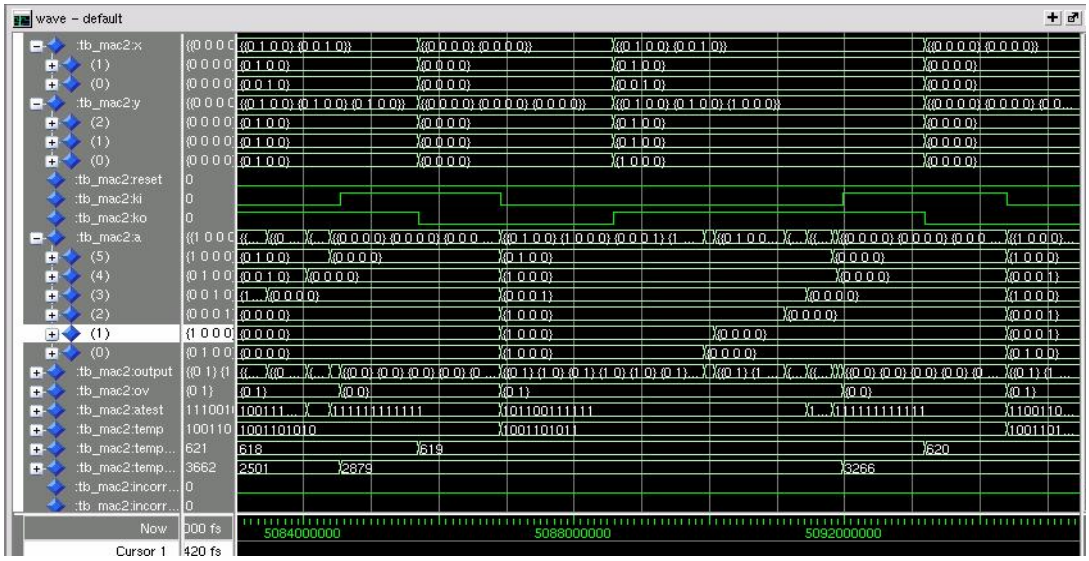


Figure 3.16. Simulation for 12+6×4 MAC

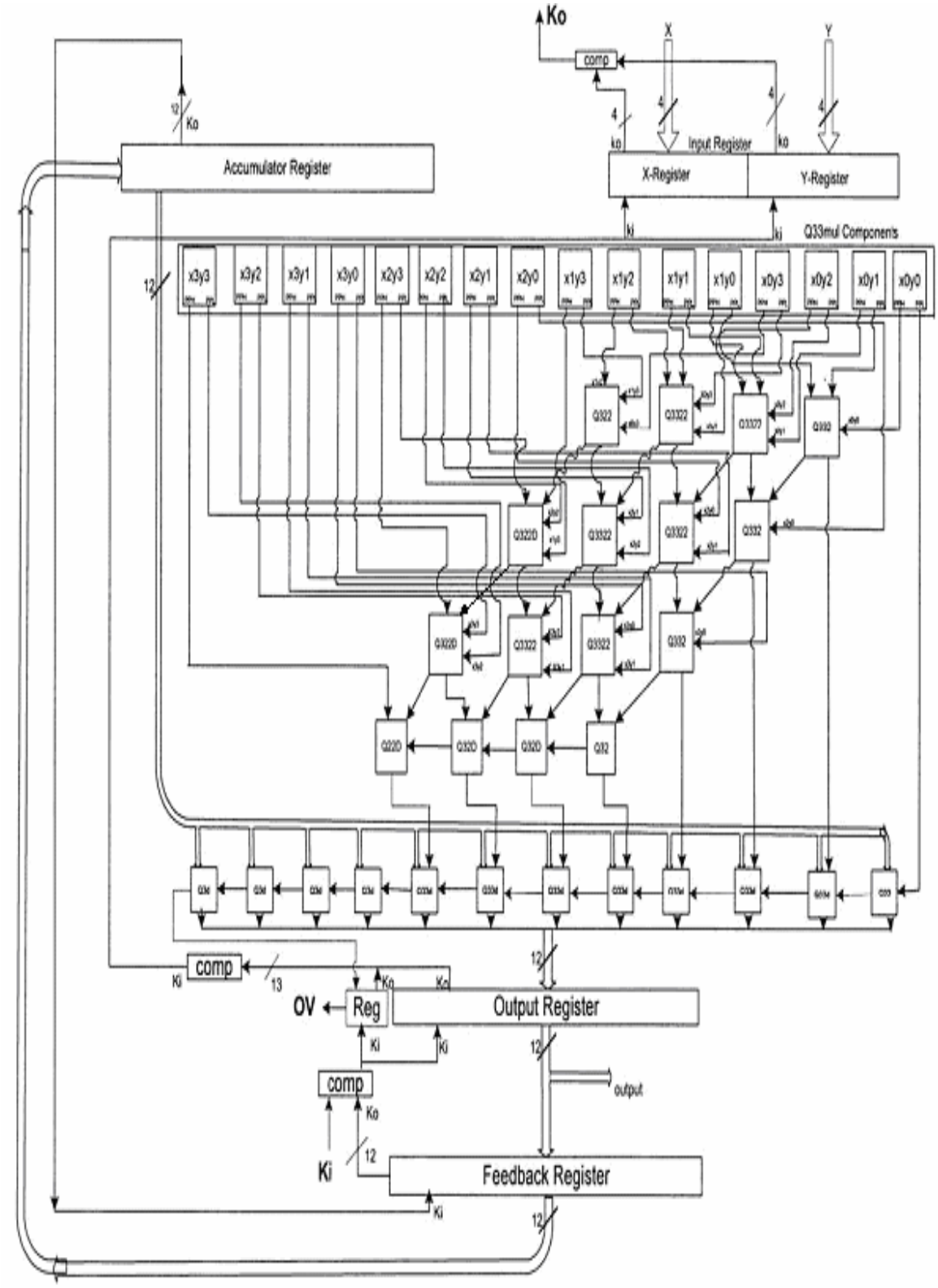


Figure 3.17. Complete system diagram of 24+8×8 MAC

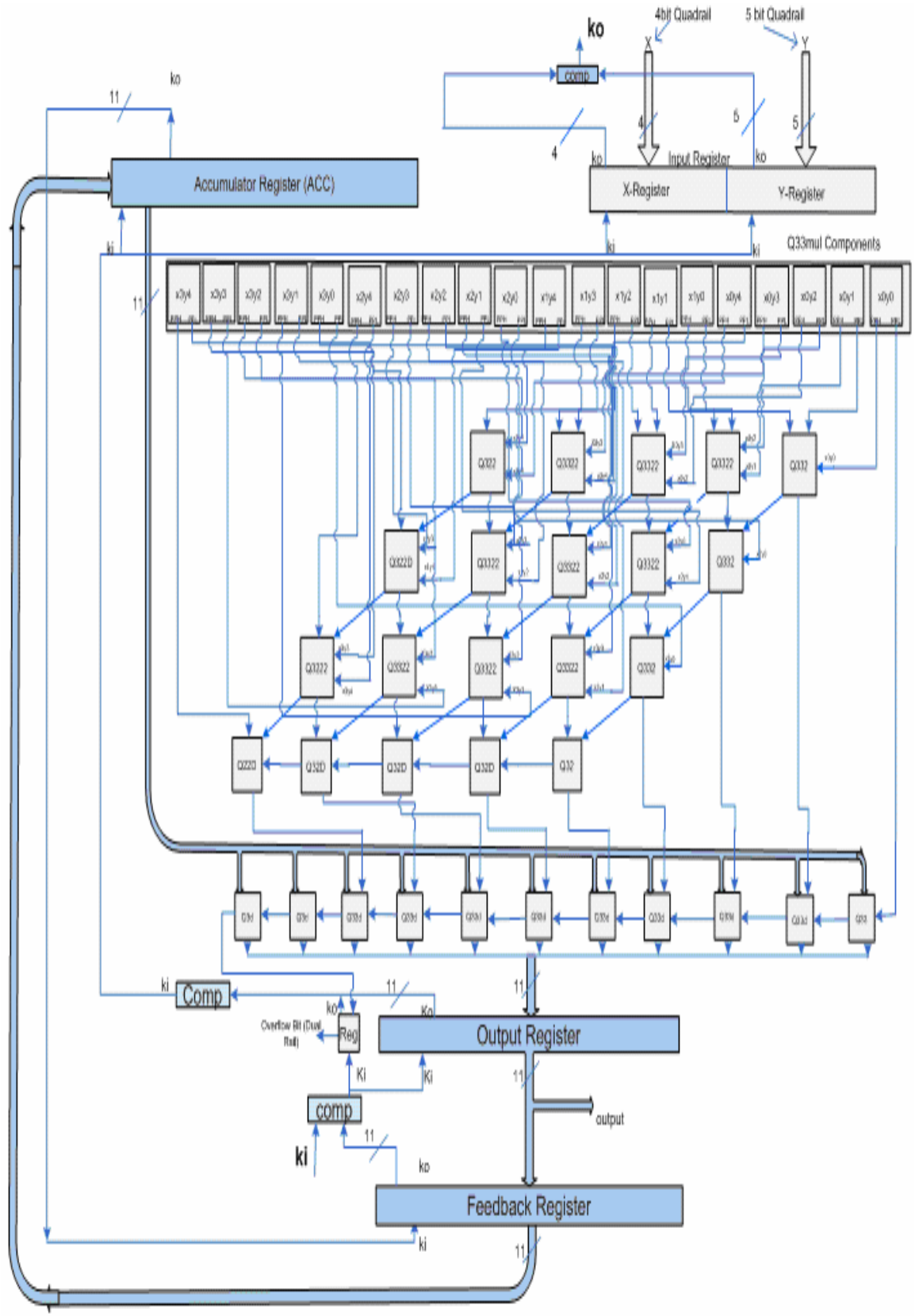


Figure 3.18. 22+10×8 MAC

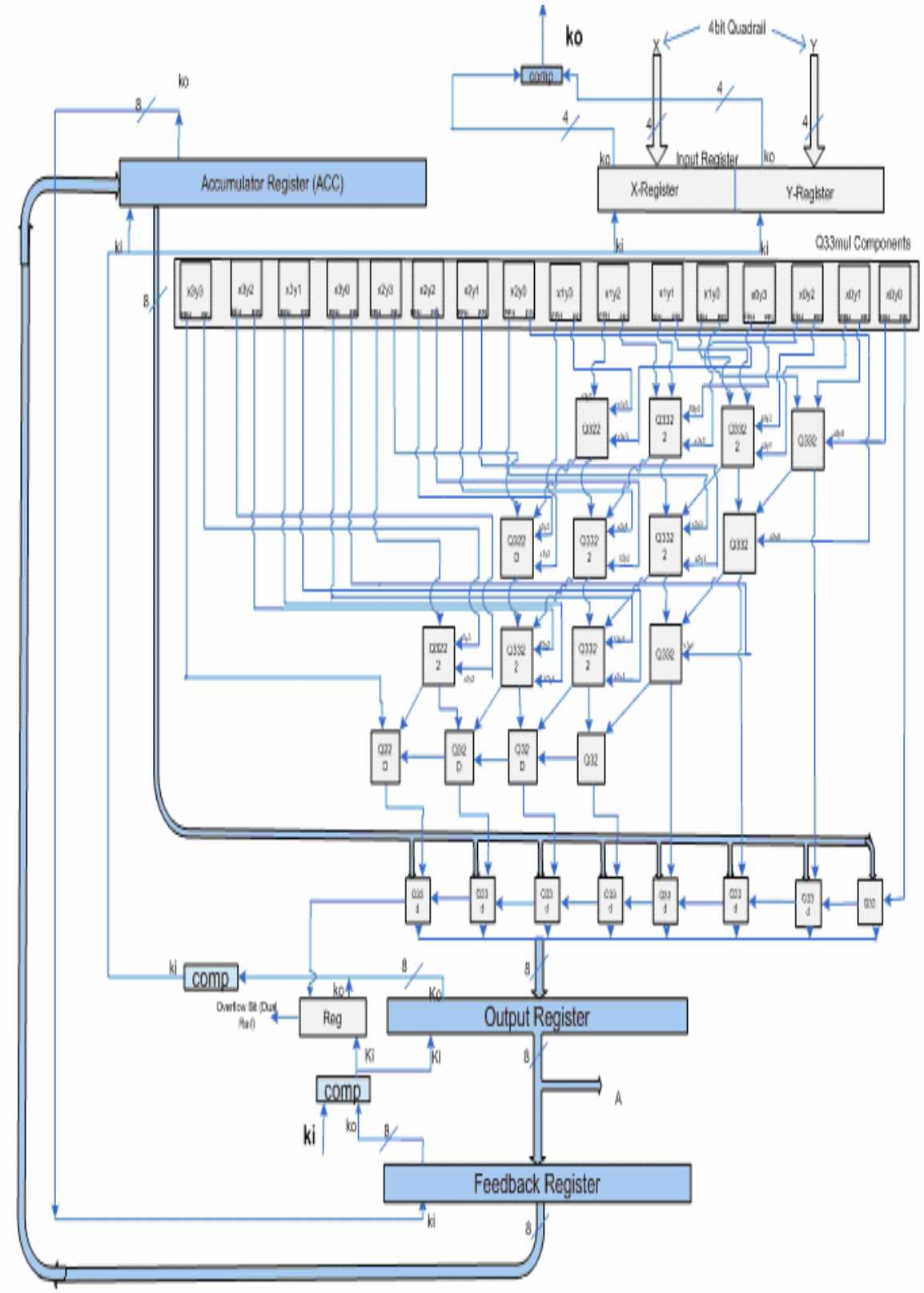


Figure 3.19. 16+8×8 MAC

4. DESIGN OF A GENERIC 2^S COMPLEMENT QUAD-RAIL MULTIPLIER

This chapter presents the design and VHDL implementation of a generic 2^s complement NULL Convention Logic quad-rail multiplier. The multiplier is non-pipelined and utilizes full-word completion; and the partial product generation algorithm is derived from the modified Baugh-Wooley algorithm [13] for binary mulitplication.

4.1. 2^S COMPLEMENT QUAD-RAIL MULTIPLICATION

4.1.1. Registers. The block diagram of a 2^s complement NCL quad-rail miltiplier is shown in Figure 4.1. The circuit has an input register for *X* and *Y* inputs and an output register for the multiplication product, *P*. The main components of the multiplier are detailed below.

4.1.2. Partial Product Generation. The 2^s complement quad-rail multiplication algorithm is derived from the binary modified Baugh-Wooley multiplication algorithm [13]. In binary Baugh-Wooley multiplication, as shown in Figure 4.2 for a 6-bit × 4-bit multiplier, PPs are generated the same way as for unsigned multiplication, except for the MSB of all rows, excluding the last row, is complemented, and all PP bits in the last row, excluding the MSB, are complemented. Additionally, a logic 1 is added at positions *M*, *N*, and *M+N*, where *M* and *N* are the lengths of the multiplicand and multiplier, respectively. Note that if *M* and *N* are equal, logic 1 is added at position M+1 instead of at positions *M* and *N*.

For unsigned quad-rail multiplication, as detailed in Section 3.1, each PP corresponds to two rows of PPs in binary multiplication, and are generated using the Q33mul component, which produces a three-rail PPH and quad-rail PPL output. Hence,

for 2^s complement PP generation, the Q33mul component can be used for all PPs except for the last row and the most significant PP of the other rows, as shown in Figure 4.3.

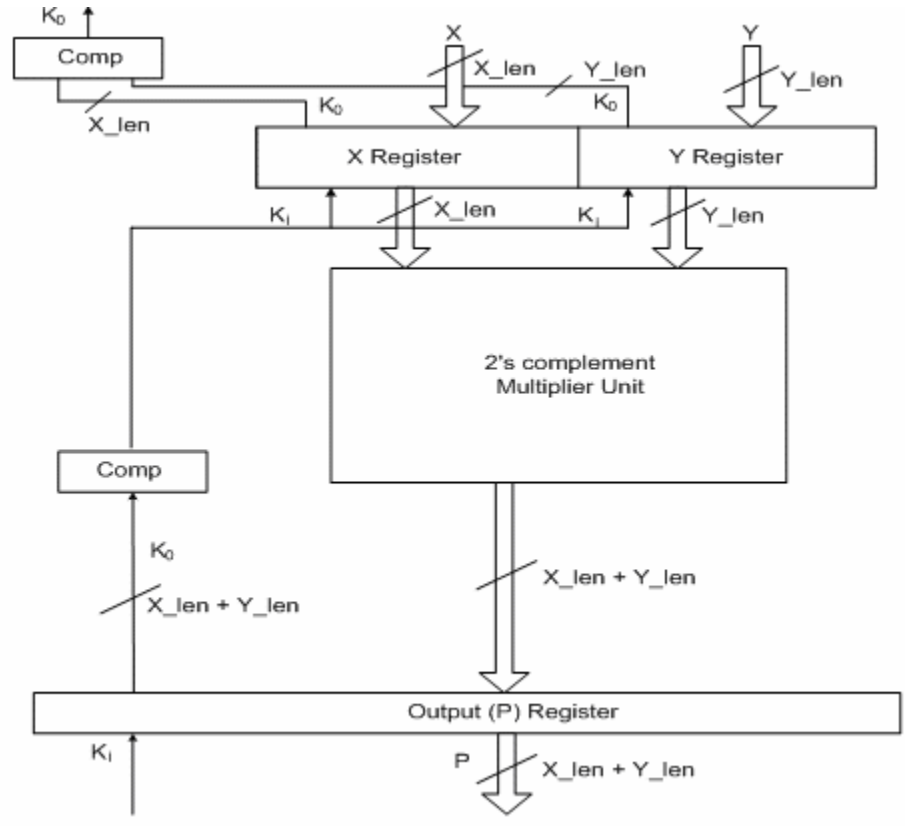


Figure 4.1. 2^s complement multiplier block diagram

The most significant PP of any row, excluding the last row, is generated using the multiplicand's most significant quad-rail signal and the multiplier's quad-rail signal corresponding to the particular PP row. A new PP generation component, called MSPP, generates these most significant PPs, and is designed from Truth Table 4.1, derived from the binary modified Baugh-Wooley algorithm. This component generates PPH and PPL,

the sum of which is equal to the sum of the two MSB PP postions in each of the 2 corresponding binary PP rows, as demonstrated in Figure 4.3.

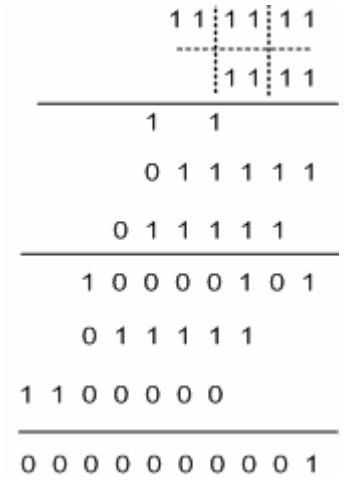


Figure 4.2. Binary 6×4 2^{s} complement multiplication

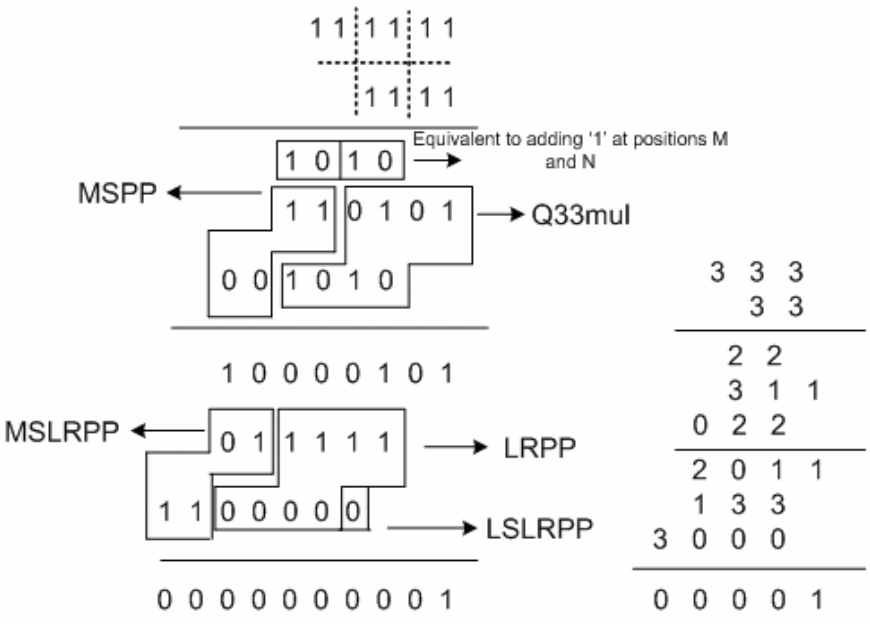


Figure 4.3. Quad-rail 2^{s} complement multiplication

Table 4.1. Truth table for partial product generation

MD	MR	MSPP		MSLRPP		LRPP	LSLRPP
		PPH	PPL	PPH	PPL	PPL	C
0	0	1	2	2	2	0	2
0	1	1	2	2	2	0	2
0	2	1	2	2	2	0	2
0	3	1	2	2	2	0	2
1	0	1	2	2	2	0	2
1	1	1	3	2	3	1	2
1	2	2	0	2	2	0	0
1	3	2	1	2	3	1	0
2	0	1	2	2	2	0	2
2	1	1	0	2	0	2	2
2	2	0	2	3	2	0	2
2	3	0	0	3	0	2	2
3	0	1	2	2	2	0	2
3	1	1	1	2	1	3	2
3	2	1	0	3	2	0	0
3	3	0	3	3	1	3	0

The last quad-rail PP row corresponds to the last two rows in binary Baugh-Wooley multiplication, the last of which is complemented; hence, PP generation components are needed for this last row, including LSLRPP, which generates the least significant dual-rail PP, *C*, MSLRPP, which generates the most significant PP, and LRPP, which generates the rest of the last row's PPs. These three circuits are designed from Truth Table 4.1 and the K-map shown in Figure 4.4, derived from the binary modified Baugh-Wooley algorithm, such that the sum of the last quad-rail PP is equivalent to the sum of the last two binary PP rows.

LSLRPP, utilizes the most significant quad-rail signal of the multiplier and the least significant quad-rail signal of the multiplicand to generate a dual-rail signal, *C*, which has a value of either 0 or 2. MSLRPP, utilizes the most significant quad-rail signal of the multiplier and the most significant quad-rail signal of the multiplicand to generate PPH and PPL, where PPH is a dual-rail signal, with value 2 or 3, and PPL is a quad-rail signal. Note that this component also takes into account the the logic 1 added at the $(M+N)^{th}$ bit postion. The remaining last row PPs are generated by LRPP. This component requires the most significant quad-rail signal of the multiplier, *MR*, and the multiplicand's quad-rail signal corresponding to the particular PP position, MD_i, as well as the multiplicand's subsequent quad-rail signal, MD_{i+1}, to generate PPH and PPL, both of which are quad-rail signals.

The logic 1s that need to be added at the M^{th} and N^{th} positons, or $(M+1)^{th}$ position if *M* equals *N*, are taken into consideration during PP summation by utilizing special adders, as detailed in the next section. Circuit diagrams for components MSPP, MSLRPP, LRPP and LSLRPP are shown in Figures 4.5, 4.6, 4.7 and 4.8, respectively.

K-map type minimization diagram for partial product high of the component LRPP is shown below.

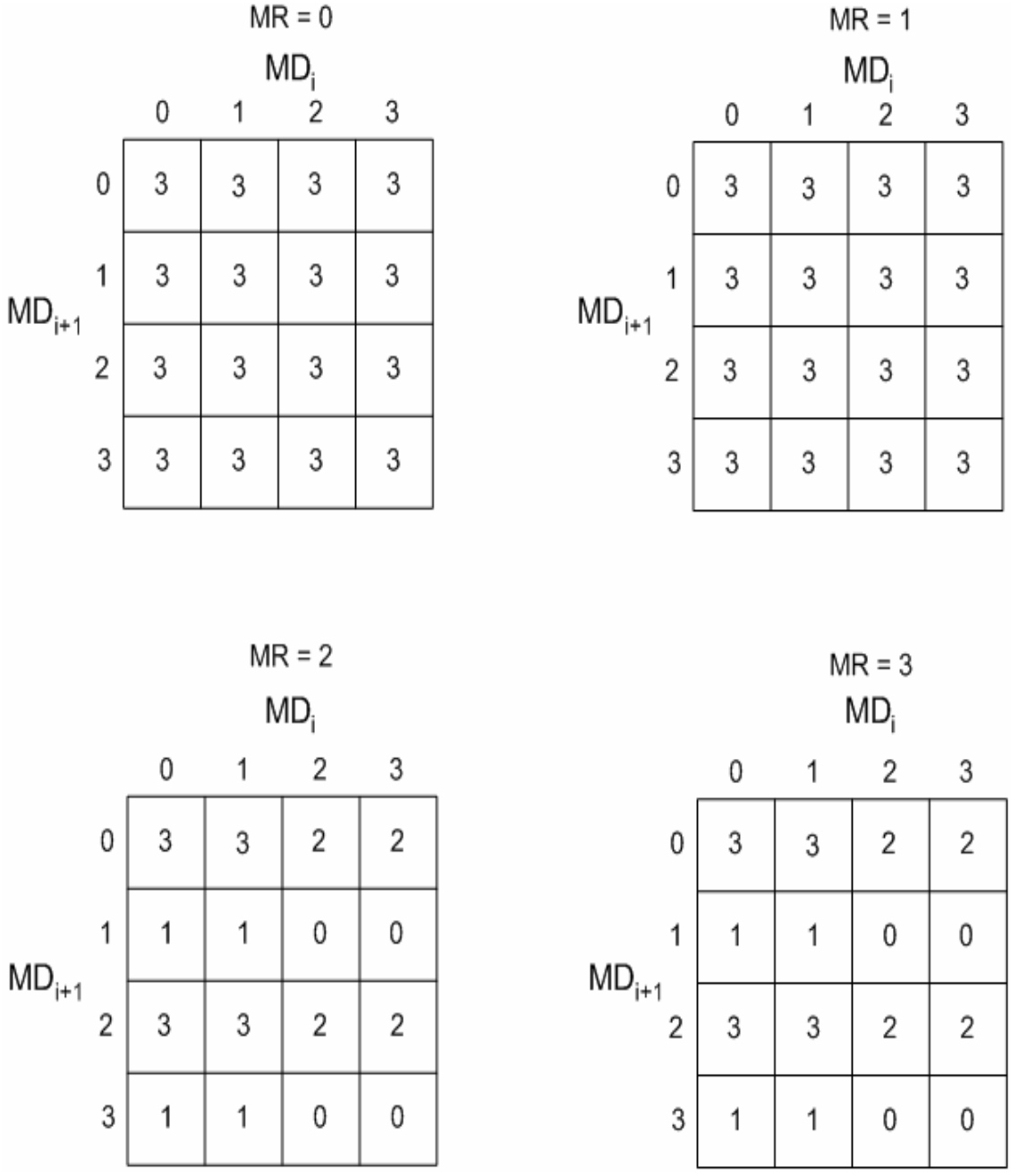
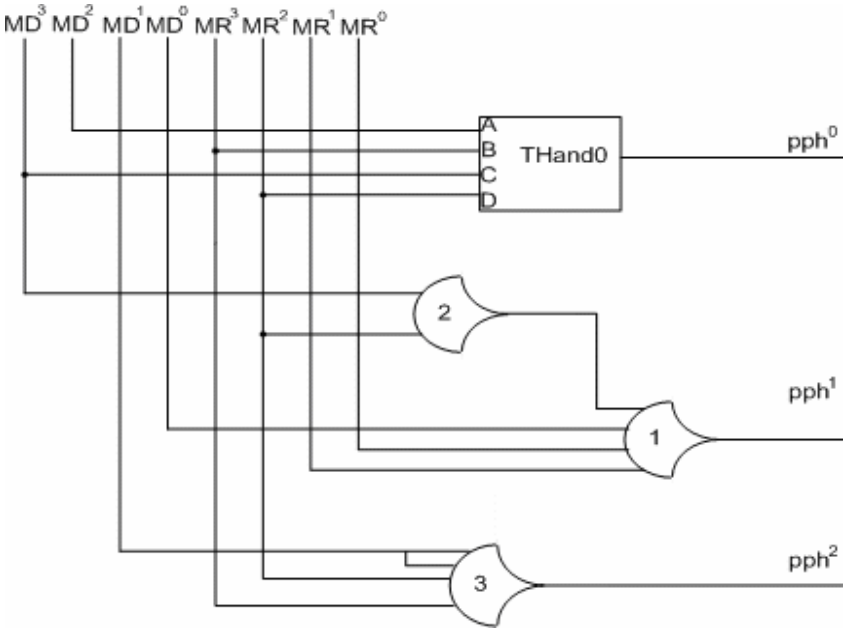
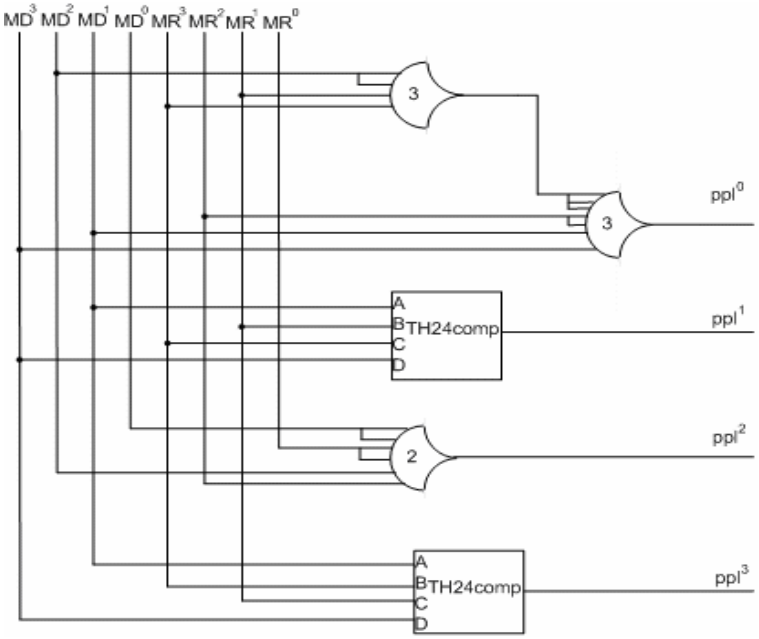


Figure 4.4. K-map for LRPP PPH

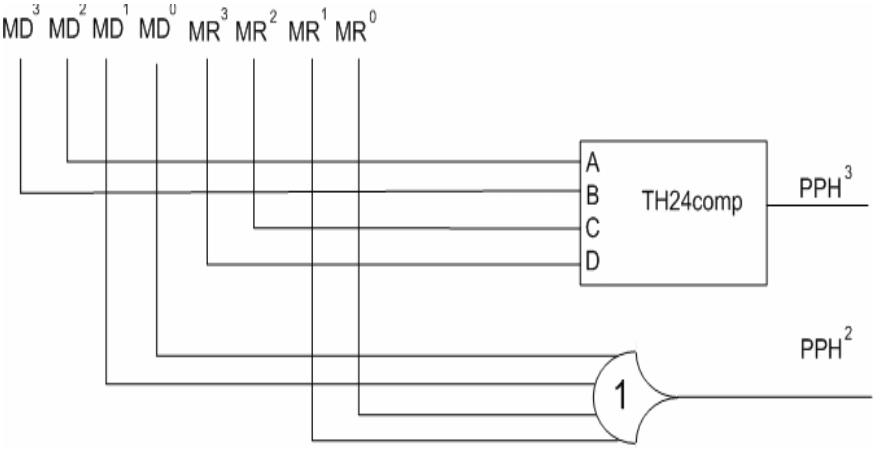


a) PPH output

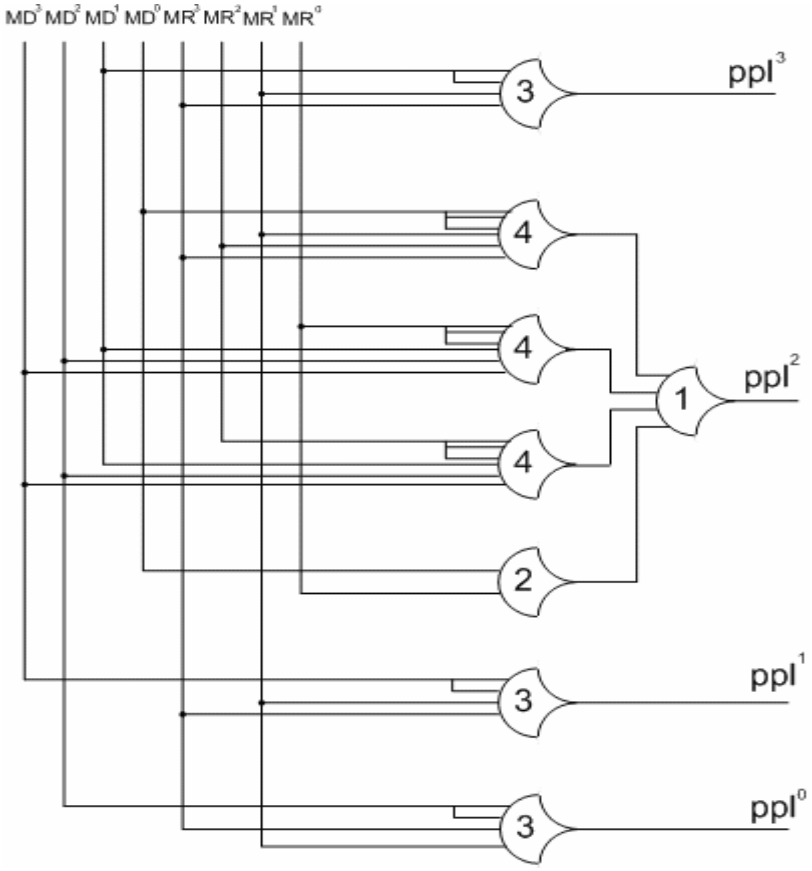


b) PPL output

Figure 4.5. MSPP circuitry

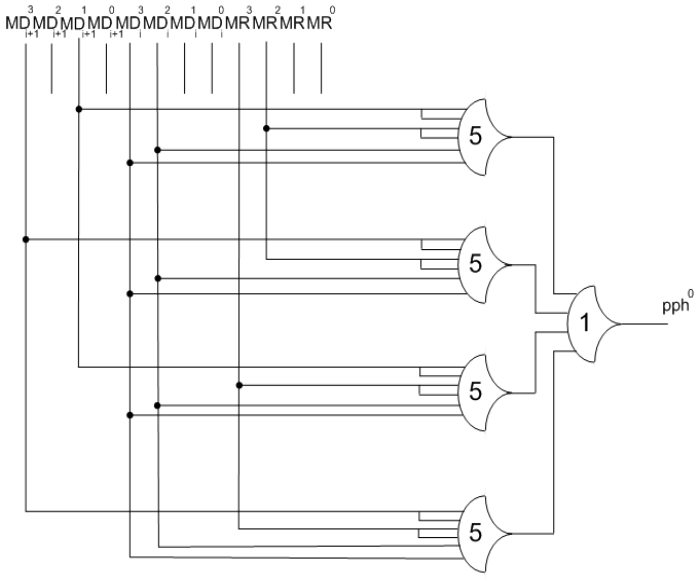


a) PPH output

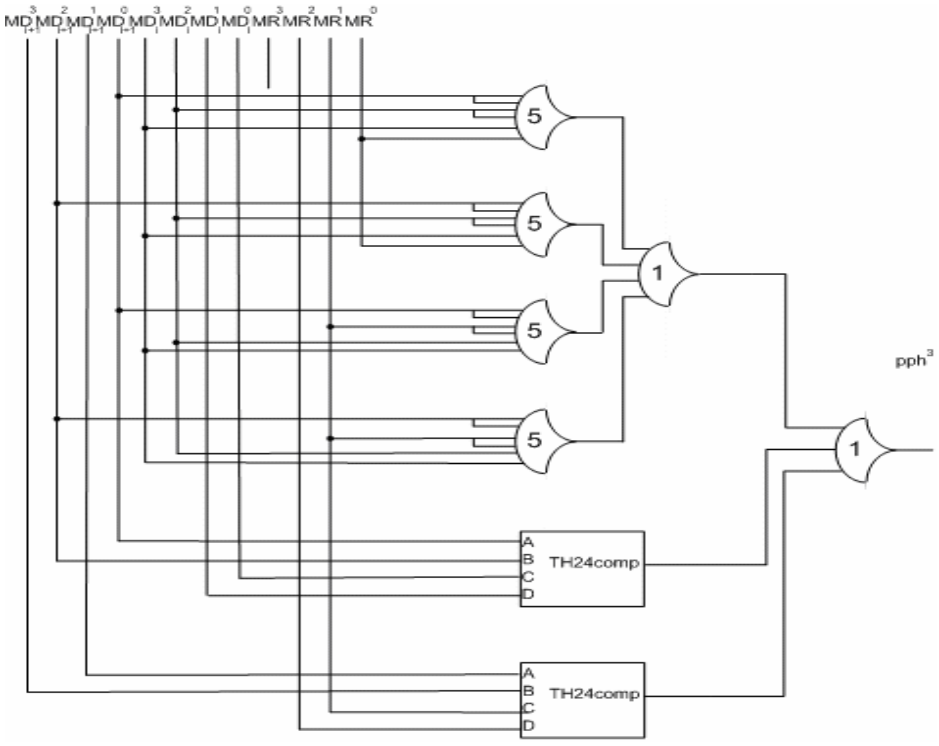


b) PPL output

Figure 4.6. MSLRPP circuitry

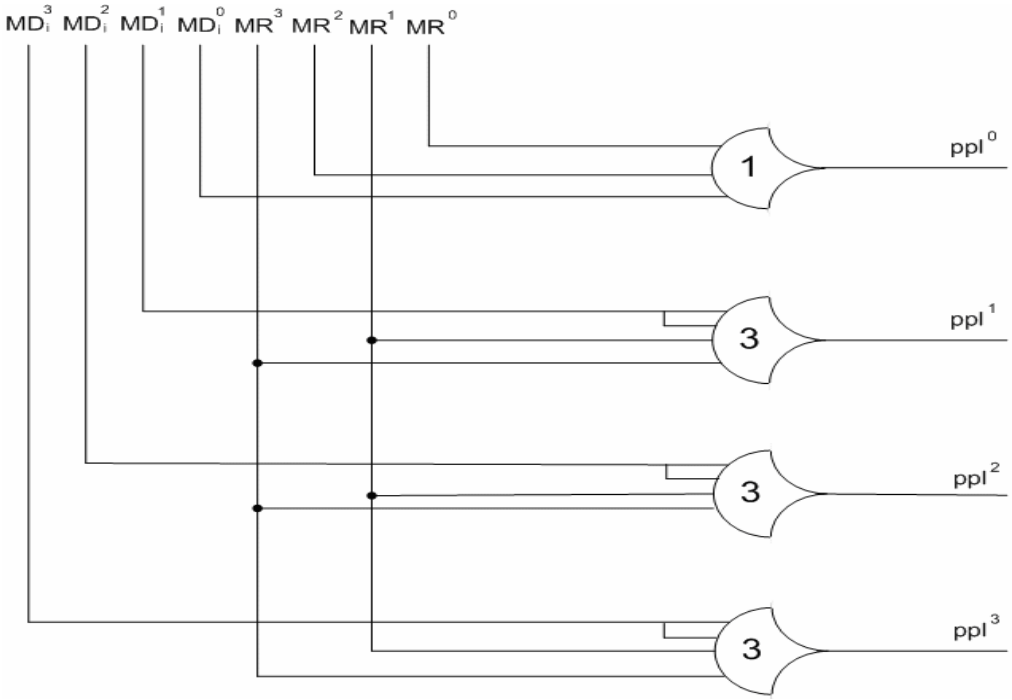


a) rail0 of the PPH output (rail1 and rail2 are symetrical to rail0)



b) rail3 of the PPH output

Figure 4.7. LRPP circuitry



c) PPL output

Figure 4.7. LRPP circuitry (cont.)

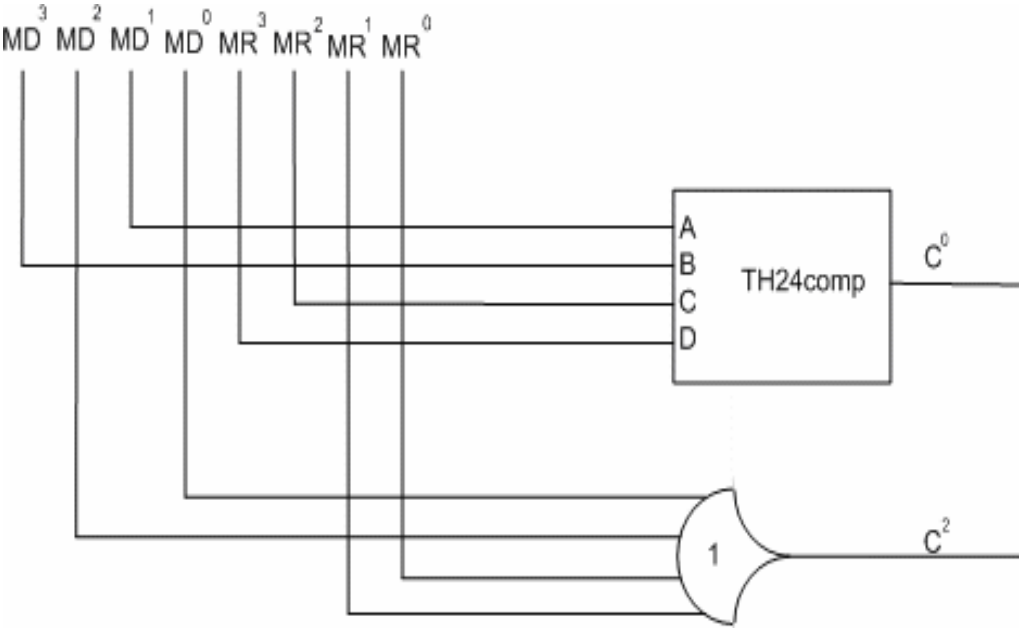


Figure 4.8. LSLRPP circuitry

4.1.3. Partial Product Summation. PPs are summed using array-structured CSAs, similar to unsigned multiplication, as detailed in Section 3.2.3. However, the structure needs to be modified slightly to account for the added logic 1s and the slight variance in PPs, as shown in Figures 4.9 and 4.10. This requires a few additional adders than the ones explained in Section 3.2.5.

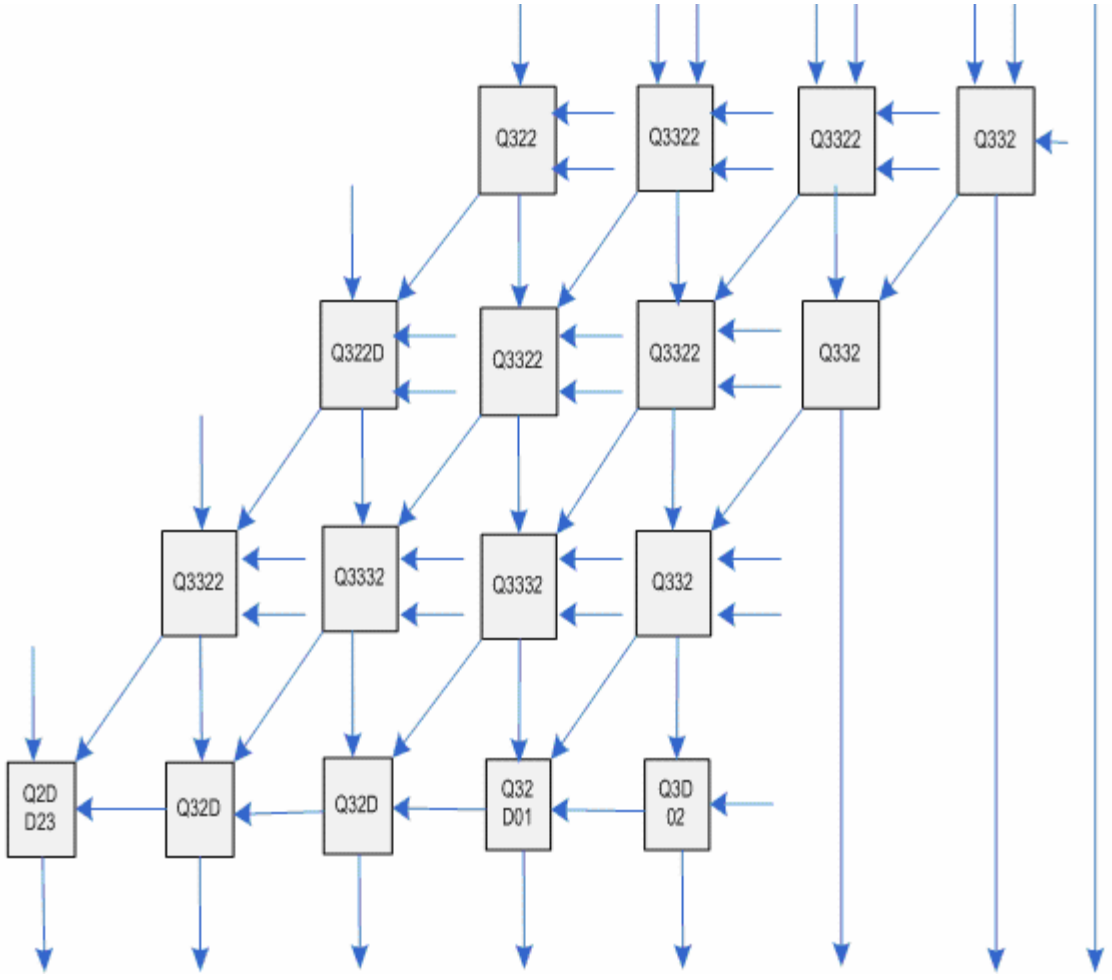


Figure 4.9. 8×8 quad-rail 2^s complement partial product summation

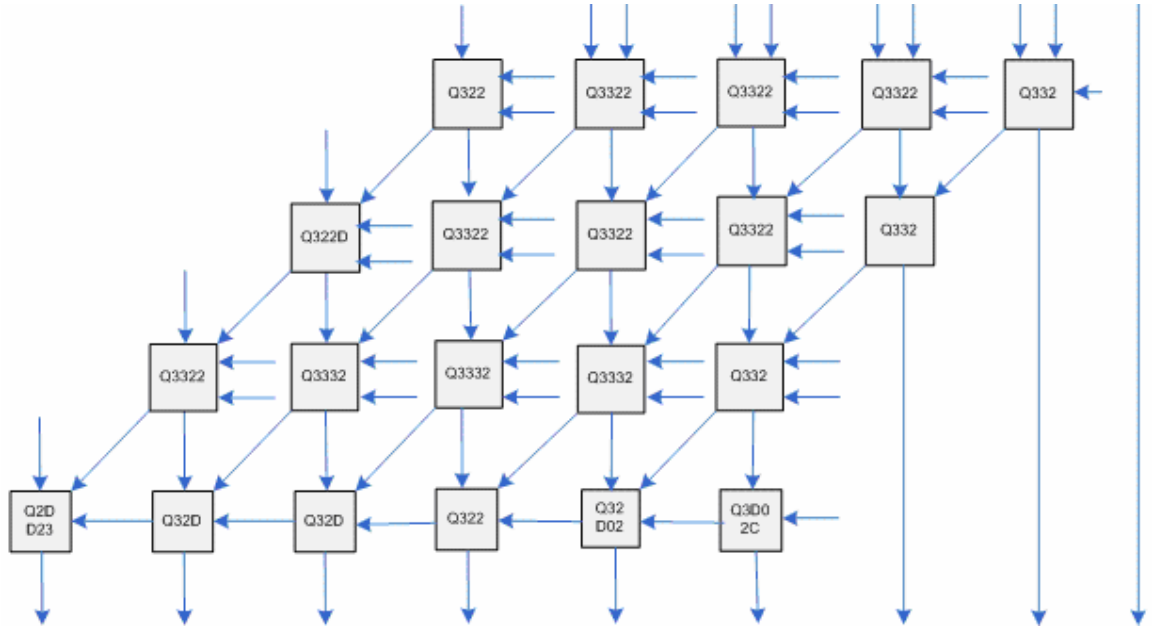
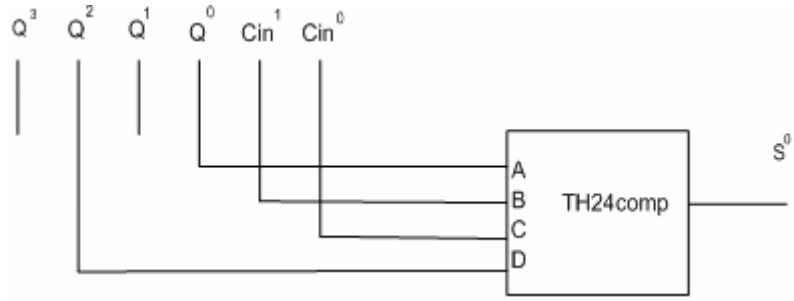


Figure 4.10. 10×8 quad-rail 2^s complement partial product summation

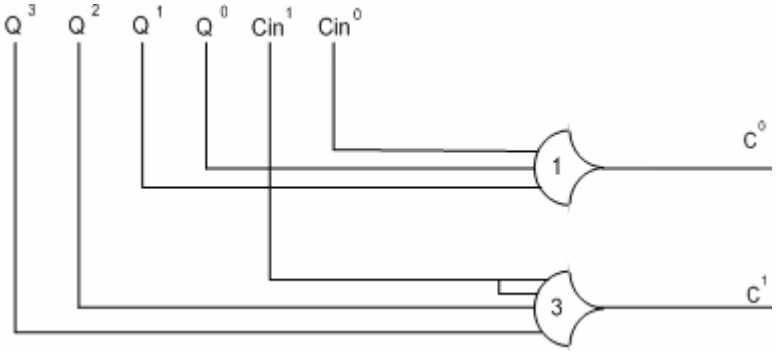
4.1.4. Adders for 2^s Complement Array Multiplication. The same adders used for the unsigned quad-rail multiplier are used for the 2^s Complement quad-rail multiplier, except for Q3Dadd, which is not needed. Additionally, a number of new adders are needed. PP summation is similar to that in unsigned quad-rail multiplication except for the last stage, the RCA. This requires an adder, called Q3D2add, to add the least significant dual-rail signal, *C*, which has a value of 0 or 2. Depending on the input vector lengths, different adders are instantiated at positons *M* and *N*, or *M+1*, to account for the logic 1s. Additionally, the last row's PPH signals are quad-rail instead of three-rail, therefore requiring a different adder. These new adder components are detailed below.

1. Q3D02add – This adder, shown in Figure 4.11, is used to add a quad-rail signal and a dual-rail signal, representing values 0 or 2. *Sum* is a quad-rail signal while *Carry*

is a dual-rail signal. Note that the *Sum* output rails are semetrical, so only rail0 is depicted.



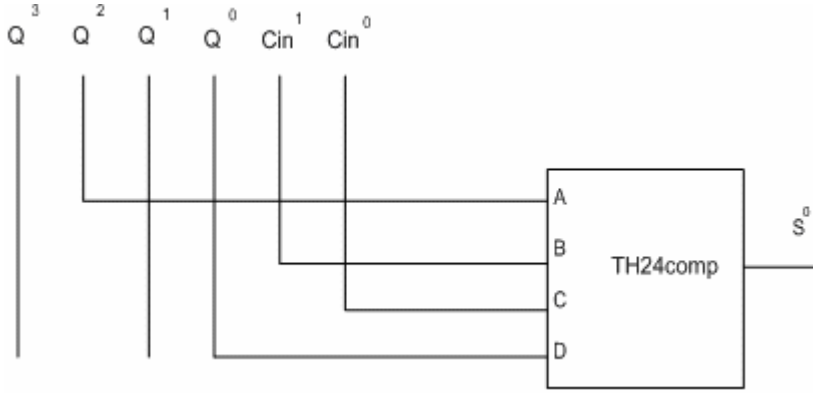
a) rail0 of the Sum output



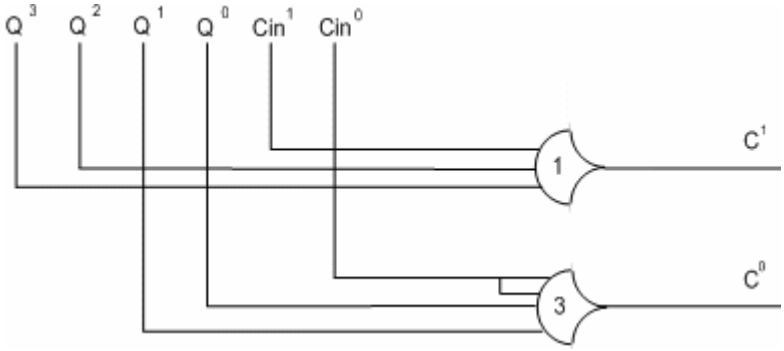
b) Carry output

Figure 4.11. Q3D02add circuitry

2. Q3D02Cadd – This adder, shown in Figure 4.12, is used to add a quad-rail signal, a dual-rail signal, representing values 0 or 2, and a constant value of 2, to account for a logic 1 added at position *N*. *Sum* is a quad-rail signal while *Carry* is a dual-rail signal. Note that the *Sum* output rails are semetrical, so only rail0 is depicted.



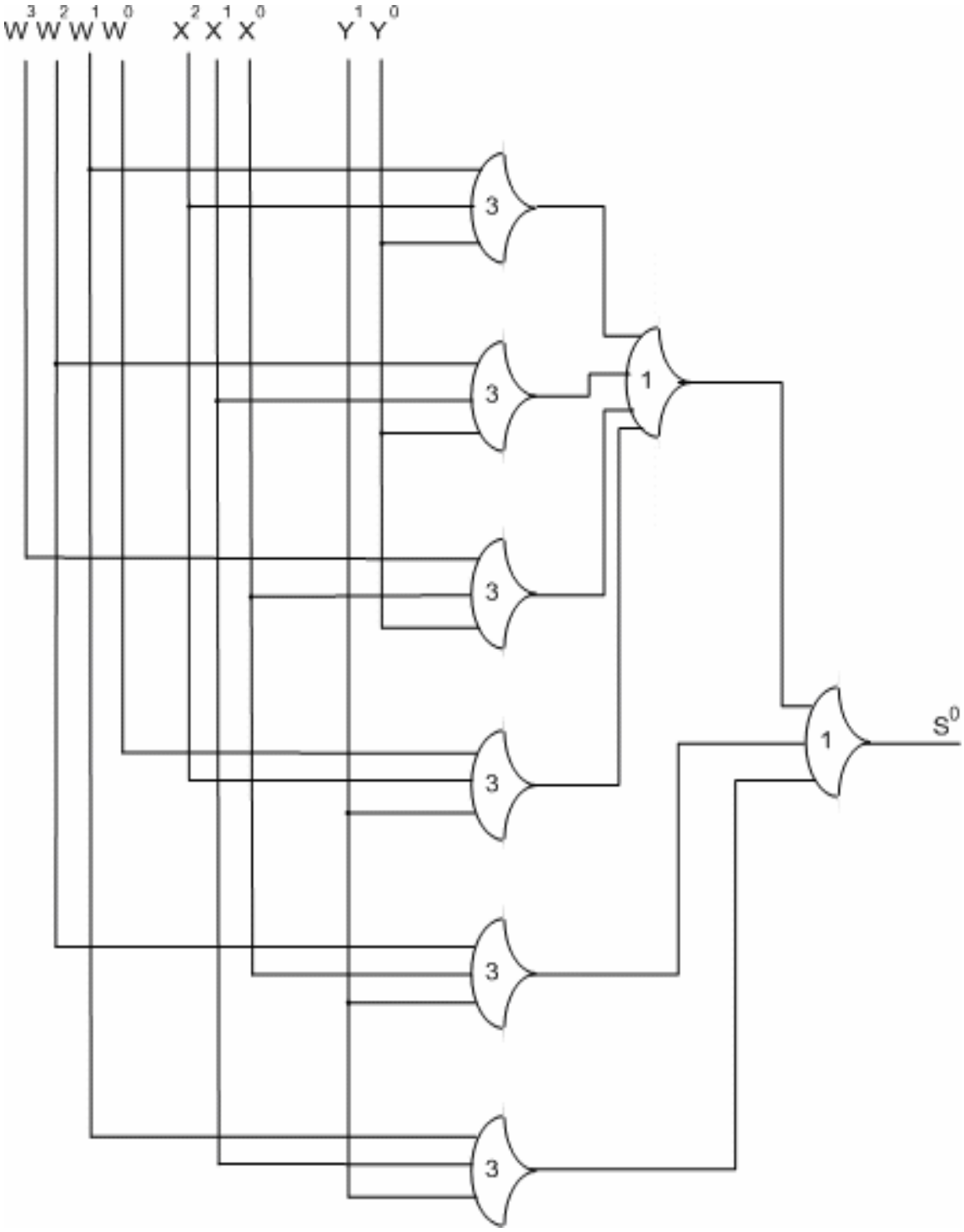
a) rail0 of the Sum output



b) Carry output

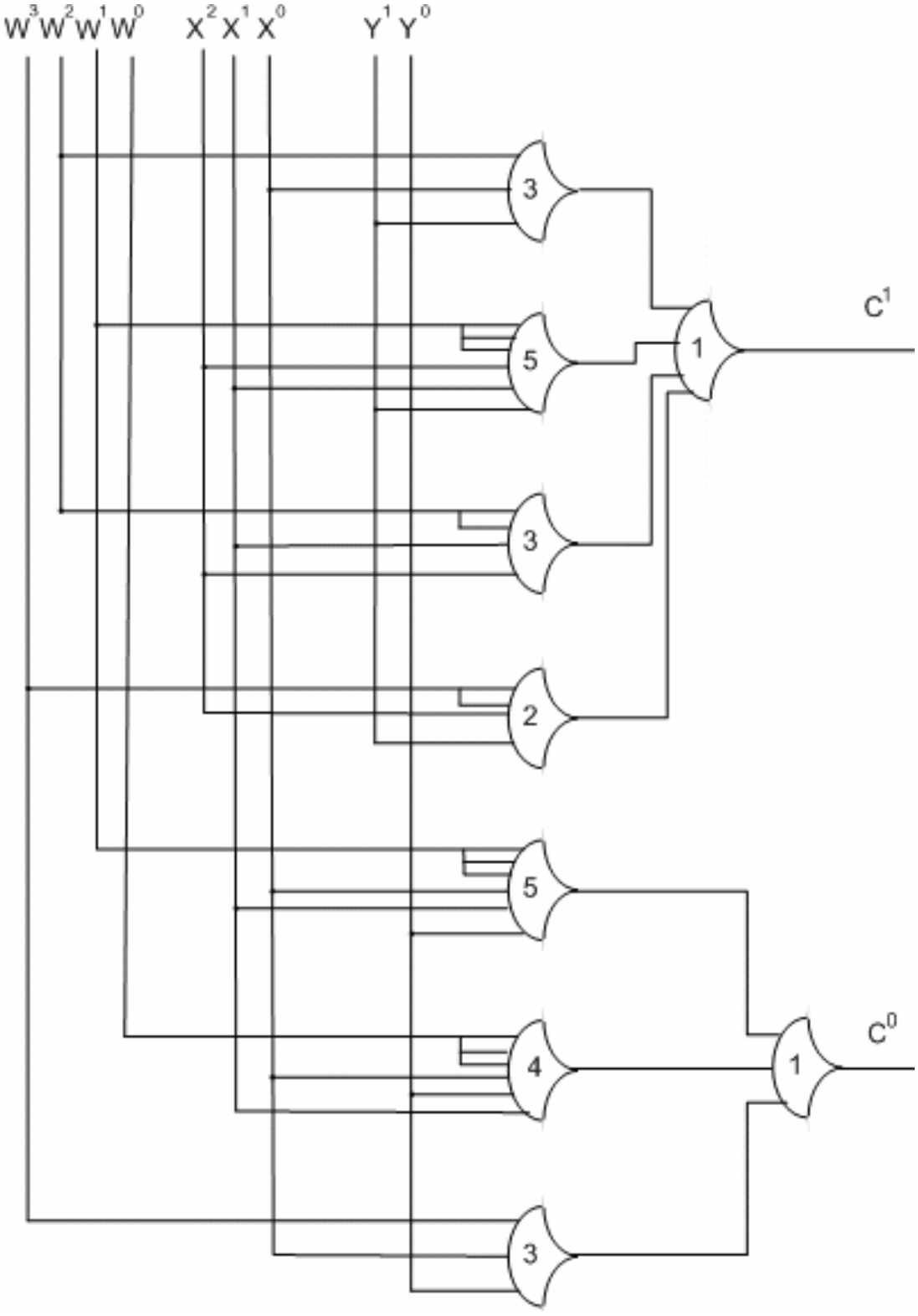
Figure 4.12. Q3D02Cadd circuitry

3. Q32D01add – This adder, shown in Figure 4.13, is used to add a quad-rail signal, a three-rail signal, a dual-rail signal, and a constant value of 1, to account for a logic 1 added at position *M+N*. *Sum* is a quad-rail signal and *Carry* is a dual-rail signal. Note that the *Sum* output rails are semetrical, so only rail0 is depicted.



a) rail0 of the Sum output

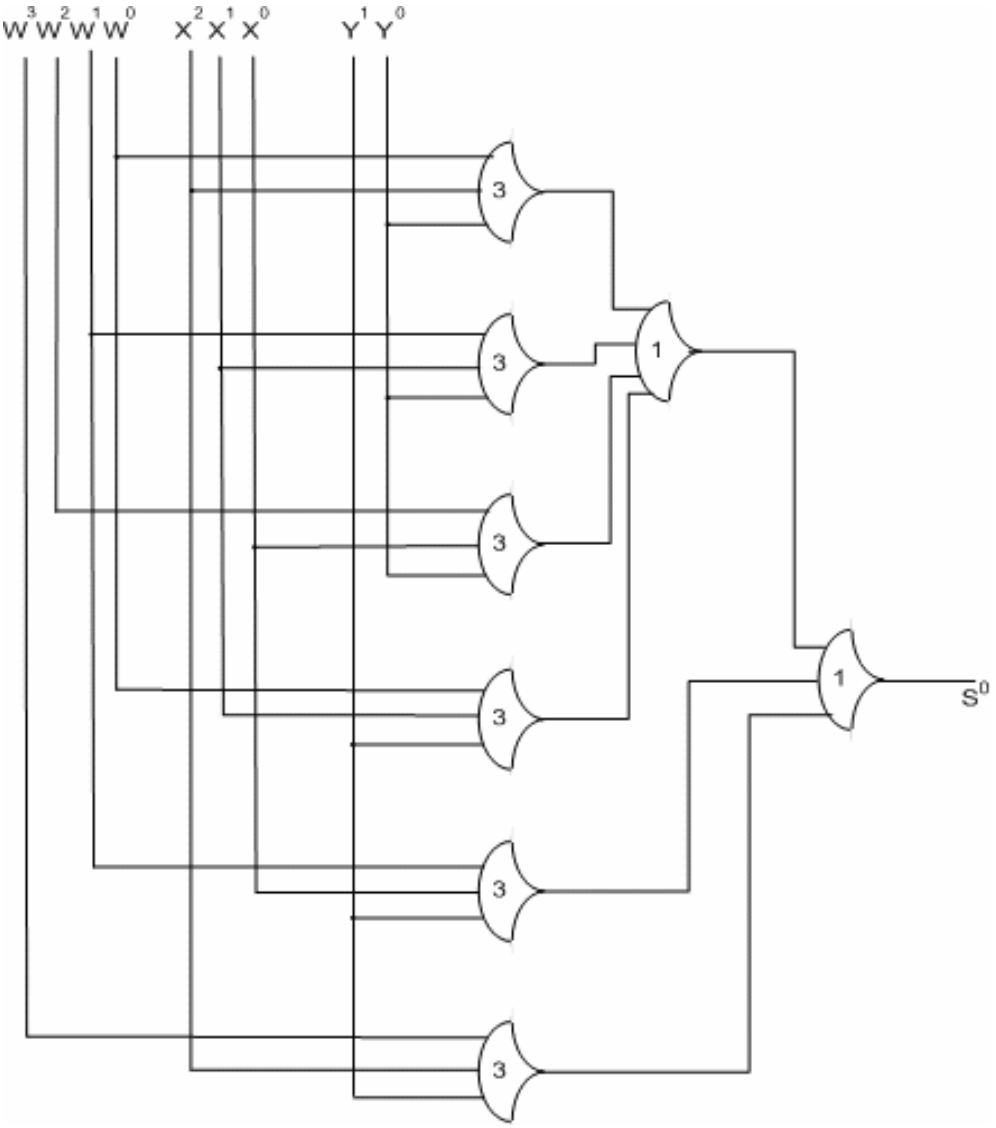
Figure 4.13. Q32D01add circuitry



b) Carry output

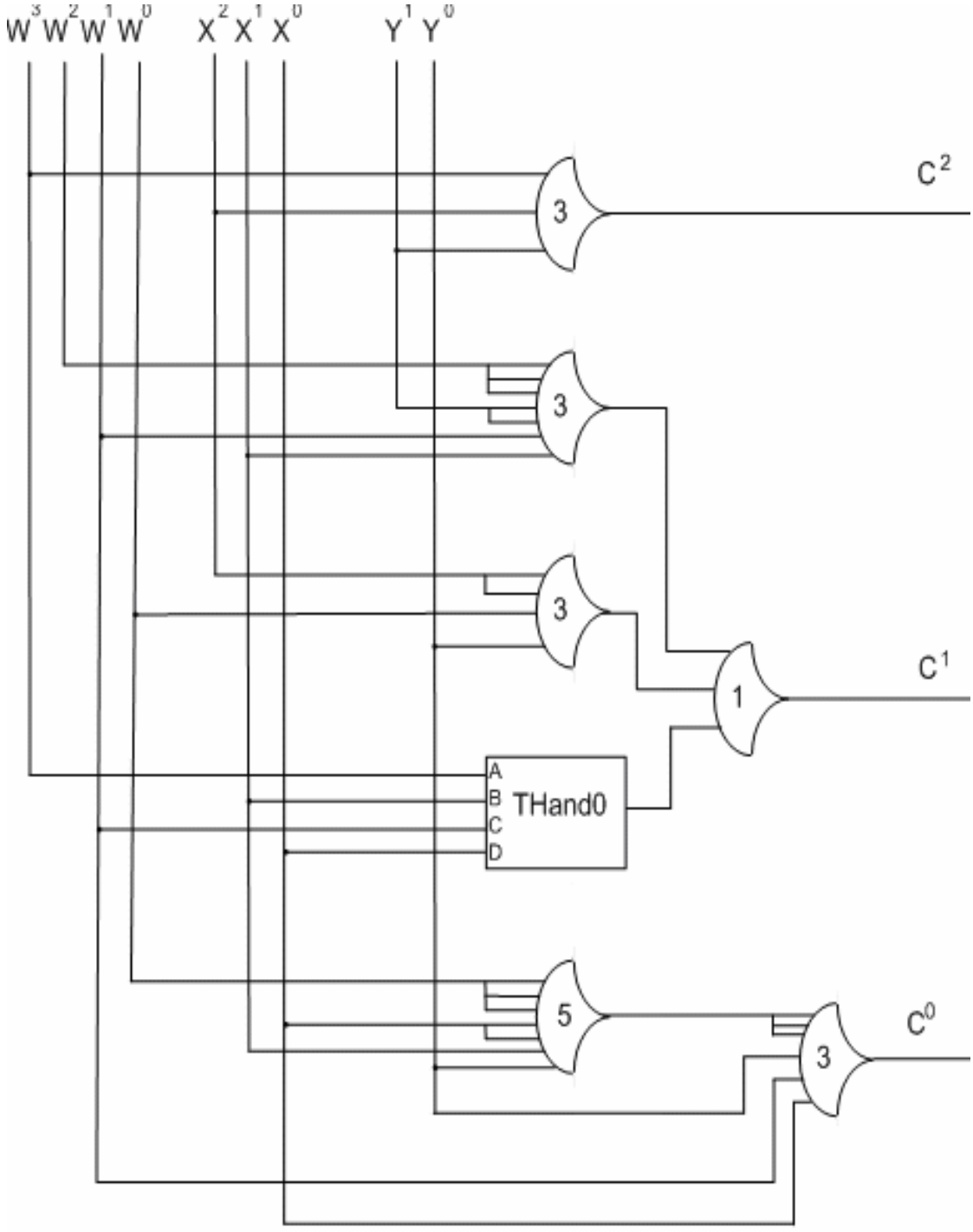
Figure 4.13. Q32D01add circuitry (cont.)

4. Q32D02add – This adder, shown in Figure 4.14, is used to add a quad-rail signal, a three-rail signal, a dual-rail signal, and a constant value of 2, to account for a logic 1 added at position *M*. *Sum* is a quad-rail signal and *Carry* is a three-rail signal. Note that the *Sum* output rails are semetrical, so only rail0 is depicted.



a) rail0 of the Sum output

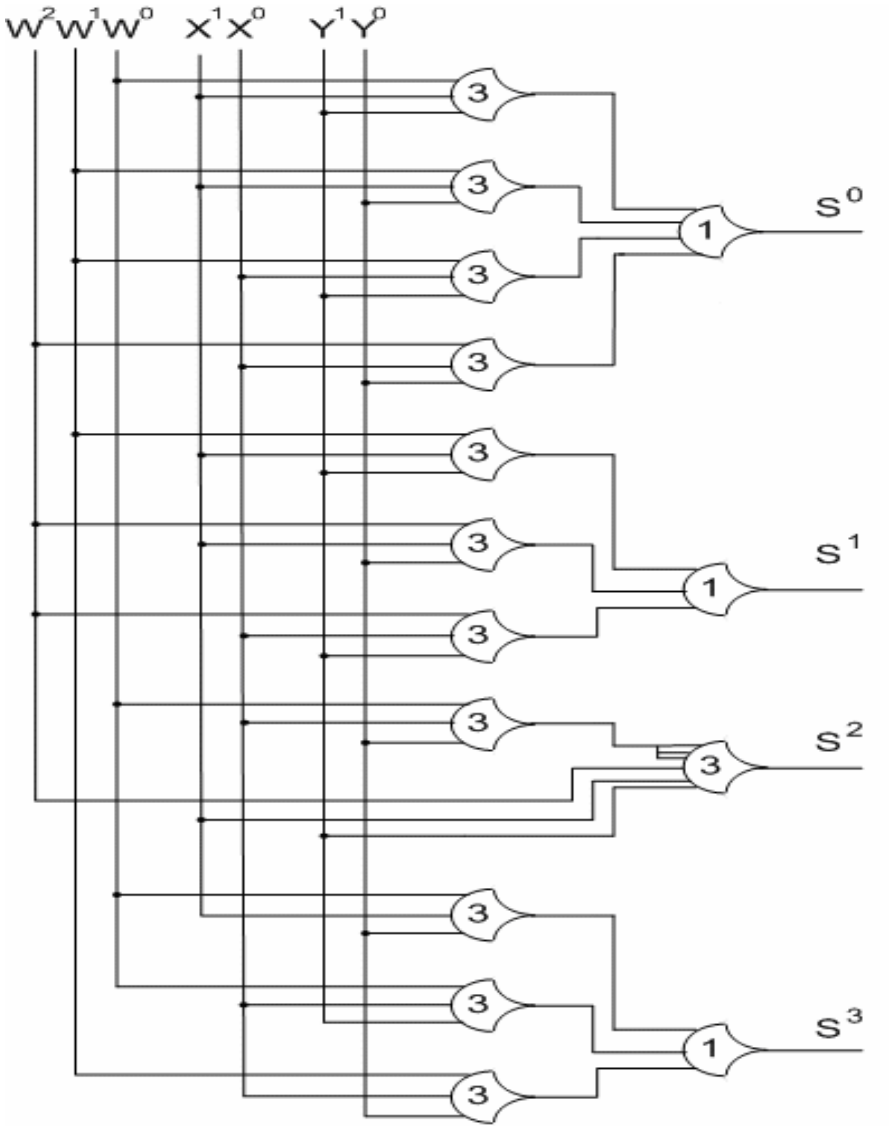
Figure 4.14. Q32D02add circuitry



b) Carry output

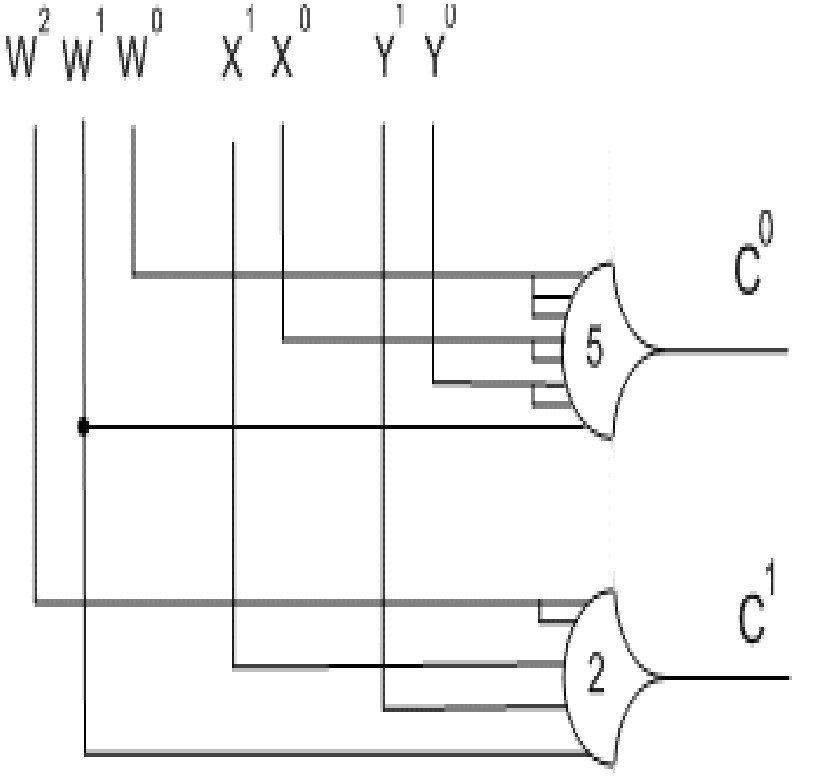
Figure 4.14. Q32D02add circuitry (cont.)

5. Q2DD23add – This adder, shown in Figure 4.15, is used to add a three-rail signal, a dual-rail signal and another dual-rail signal which has a value of 2 or 3. *Sum* is a quad-rail signal and *Carry* is a dual-rail signal.



a) Sum output

Figure 4.15. Q2DD23add circuitry



b) Carry output

Figure 4.15. Q2DD23add circuitry (cont.)

6. Q3332add – This adder is used to add three quad-rail signals and a three-rail signal. *Sum* is a quad-rail signal and *Carry* is a three-rail signal. Both the *Sum* and *Carry* have a maximum of 4 gate delays, and are not shown due to the large size of the circuit.
7. Q332Dadd – This adder is used to add two quad-rail signals, a three-rail signal, and a dual-rail signal. *Su*m is a quad-rail signal and *Carry* is a three-rail signal. The *Sum* and *Carry* have a maximum of 4 and 3 gate delays, respectively, and are not shown due to the large size of the circuit. Gate delays for each adder and partial product generation component is given in the Table 4.2 shown below.

Table 4.2. Gate delays for 2^S complement multiplier additional components

COMPONENT	GATE DELAY	
	SUM/PPL	CARRY/PPH
Q3332	4	4
Q332D	4	3
Q32D02	3	2
Q32D01	3	2
Q3D02C	1	1
Q3D02	1	1
Q2DD23	2	1
MSPP	2	2
MSLRPP	2	1
LRPP	1	3
LSLRPP	1	-

4.2.VHDL IMPLEMENTATION

The combinational circuitry components of the 2^s complement quad-rail multiplier were designed and implemented as structural, gate-level VHDL modules. Every component was subjected to exhaustive testing using VHDL testbenches specific to the components. These basic components were then used to create a generic multiplier, where the length of the input registers, output register, the number of Q33mul, MSPP, and LRPP components, and the number of rows and columns in the array multiplier, change according to the length of the input vectors, *X* and *Y*, given as generic constants.

The functionality of the quad-rail 2^s complement multiplier has been verified using exhaustive testbenches for a 6×4 (i.e., 3×2 in quad-rail) and 4×4 multiplier, and a 256 testvector testbench for a 44×24 multiplier. Simulation results confirmed that all of the various sized multipliers tested functioned correctly, and showed that the average time per multiplcation operation increased proportional to the length of the multiplier, N

as the multiplier size grew larger (i.e., 3.9 ns, 4.81 ns, and 35.7 ns for the 4×4, 6×4, and 44×24 multipliers, respectively), as expected. Simulation results for one operation of the 6×4 multiplier are given in Figure 4.16 showing that $123_4 \times 21_4 = 31003_4$ (i.e., $27_{10} \times -7_{10} = -189_{10}$), as expected. The complete system diagrams of an 8×8 (i.e., 4×4 in quad-rail) and 10×8 multiplier are shown in Figures 4.17 and 4.18, respectively.

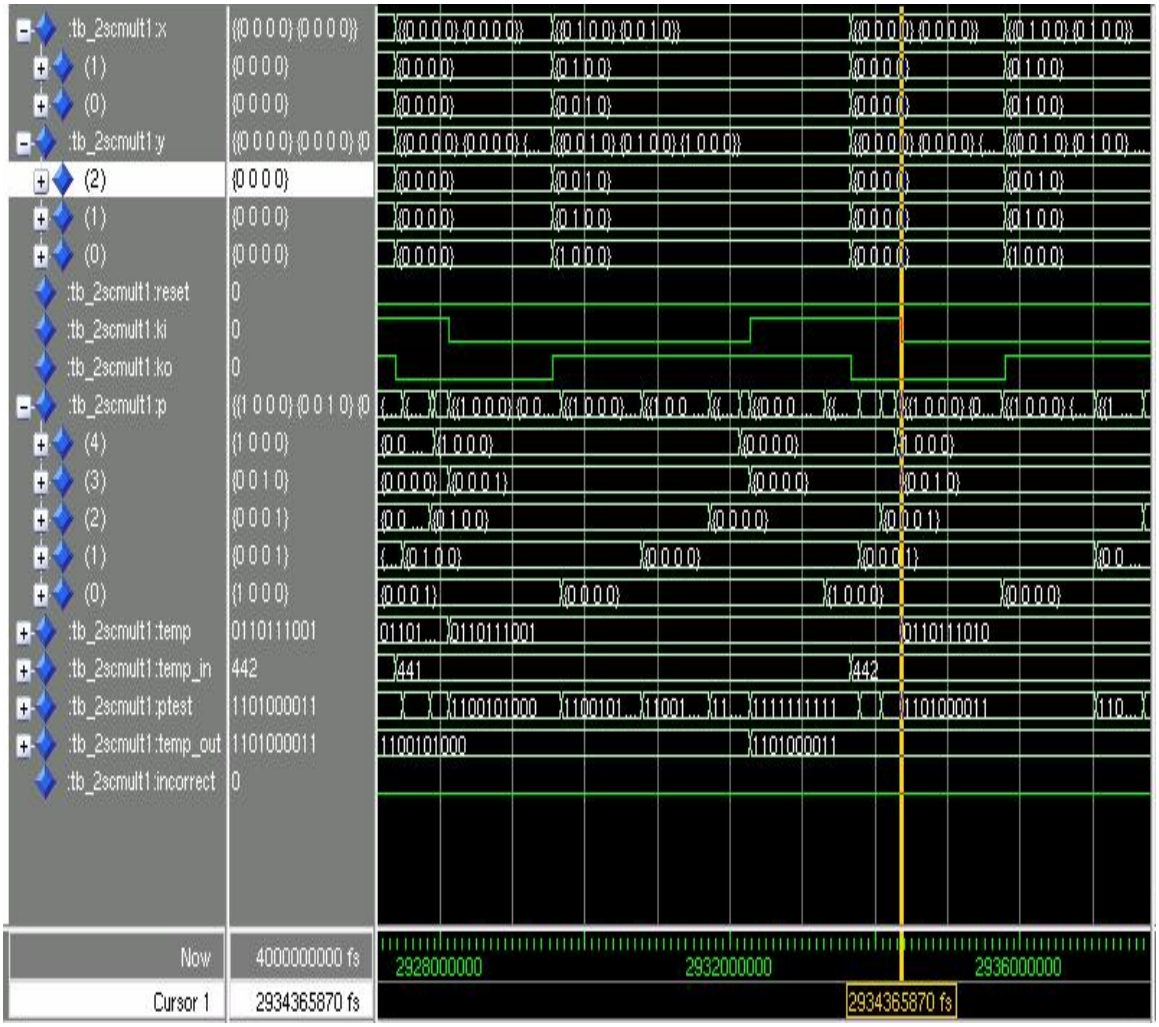


Figure 4.16. Simulation results of 6×4 multiplication

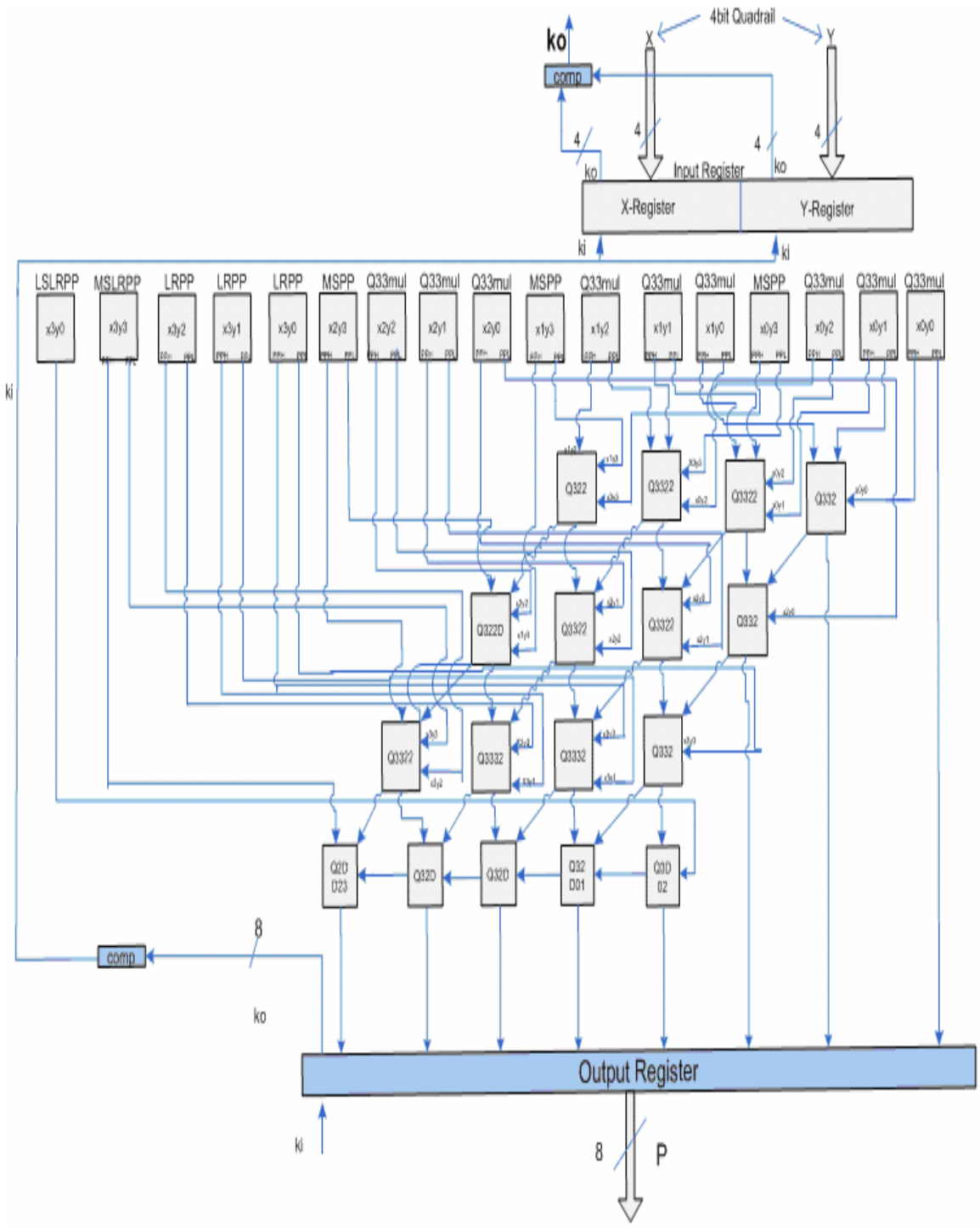


Figure 4.17. Quad-rail 2^s complement 8×8 multiplier

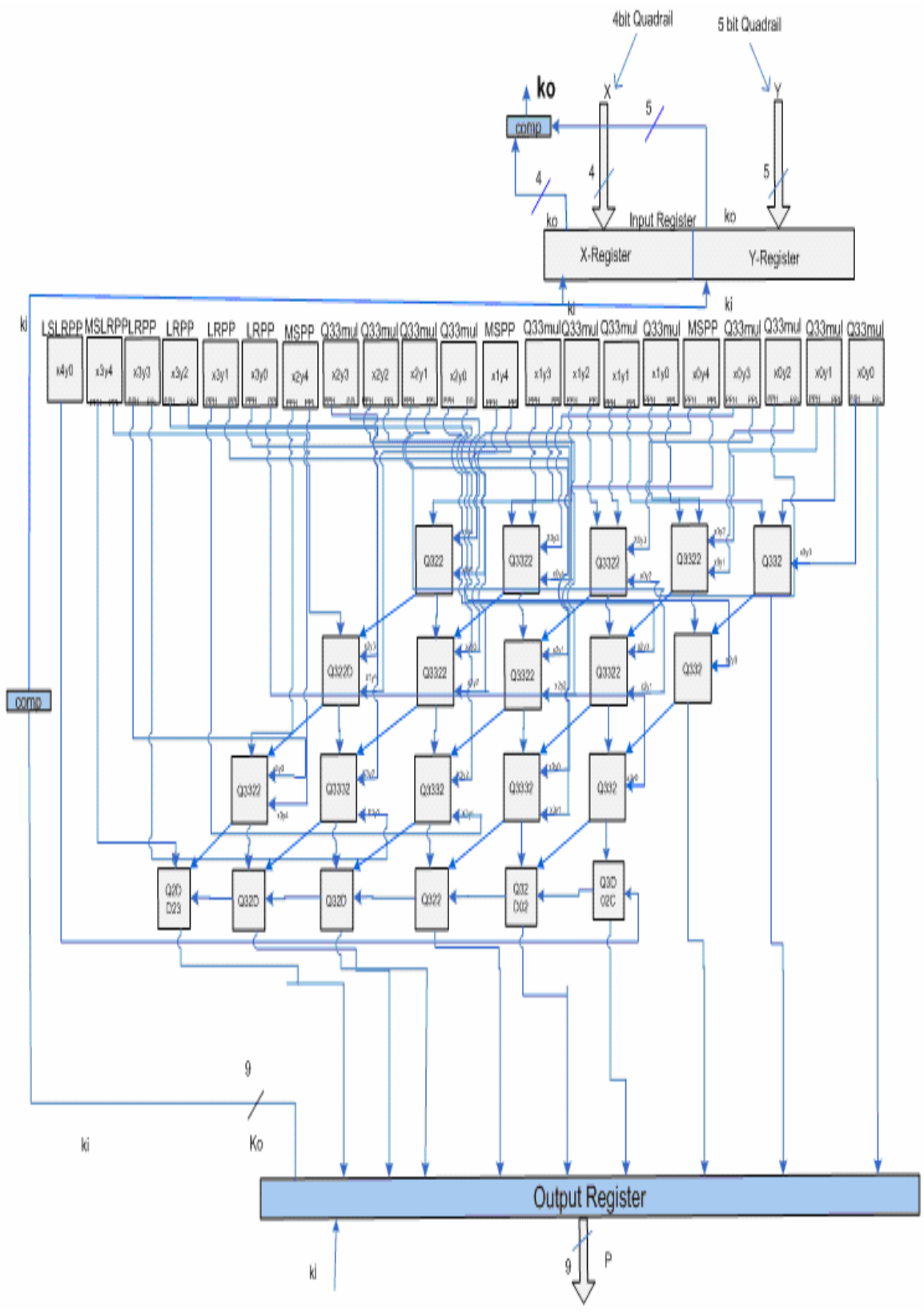


Figure 4.18. Quad-rail 2^s complement 10×8 multiplier

5. CONCLUSIONS AND FUTURE WORK

This thesis presents the design and VHDL implementation of generic versions of an NCL quad-rail unsigned Multiply and Accumulate (MAC) unit and quad-rail 2^s complement multipler. Both system level designs and each sub-component have been simulated and tested exhaustively using VHDL testbenches, showing that the circuits are functionally correct. These generic circuit implementations will be utilized in future versions of NCL synthesis tools to automatically instantiate arithmetic components from IEEE numeric_std package usage during NCL systhesis. Additional generic NCL arithmetic components (e.g., divider) also need to be designed to accommodate the full range of IEEE numeric_std package functions.

The last stage of PP summation for the unsigned MAC was designed using a RCA; however, this could have instead been combined with the accumulator feedback input using a CSA, which would have increased throughput [11], since this would only require one RCA instead of two. Both the MAC and multiplier could be pipelined to increase throughput; however, if speed is a primary consideration, the PPs should be summed using either a Wallace Tree or Dadda Tree, not array-structured CSAs, as used in this thesis.

BIBLIOGRAPHY

[1] J. McCardle and D. Chester, "Measuring an Asynchronous Processor's Power and Noise," *Synopsys User Group Conference (SNUG)*, Boston, 2001.

[2] C. L. Seitz, "System Timing," in *Introduction to VLSI Systems*, Addison-Wesley, pp. 218-262, 1980.

[3] C. H. (Kees) van Berkel, M. Rem, and R. Saeijs, "VLSI Programming," *1988 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 152-156, 1998.

[4] D. E. Muller, "Asynchronous Logics and Application to Information Processing," in *Switching Theory in Space Technology*, Stanford University Press, pp. 289-297, 1963.

[5] K. M. Fant and S. A. Brandt, "NULL Convention Logic: A Complete and Consistent Logic for Asynchronous Digital Circuit Synthesis," *International Conference on Application Specific Systems, Architectures, and Processors*, pp. 261-273, 1996.

[6] T. Verhoff, "Delay-Insensitive Codes – An Overview," *Distributed Computing*, Vol. 3, pp. 1-8, 1988.

[7] G. E. Sobelman and K. M. Fant, "CMOS Circuit Design of Threshold Gates with Hysteresis," *IEEE International Symposium on Circuits and Systems* (II), pp. 61-65, 1998.

[8] A. Kondratyev, L. Neukom, O. Roig, A. Taubin, and K. Fant, "Checking Delay-insensitivity: 10^4 Gates and Beyond," *Eighth International Symposium on Asynchronous Circuits and Systems*, pp. 137-145, 2002.

[9] M. V. Joshi, S. Gosavi, V. Jegadeesan, A. Basu, S. Jaiswal, W. K. Al-Assadi, and S. C. Smith, "NCL Implementation of 8-bit Dual-Rail Modified Booth Multiplier using Static and Semi-Static Primitives," *IEEE Region 5 Technical Conference*, April 2007.

[10] S. K. Bandapati, S. C. Smith, and M. Choi, "Design and Characterization of NULL Convention Self-Timed Multipliers," *IEEE Design and Test of Computers: Special Issue on Clockless VLSI Design*, Vol. 30/6, pp. 26-36, November-December 2003.

[11] S. C. Smith, R. F. DeMara, J. S. Yuan, M. Hagedorn, and D. Ferguson, "NULL Convention Multiply and Accumulate Unit with Conditional Rounding, Scaling, and Saturation," *Elsevier's Journal of Systems Architecture*, Vol. 47/12, pp. 977-998, June 2002.

[12] S. C. Smith, "Development of a Large Word-Width High-Speed Asynchronous Multiply and Accumulate Unit," *Elsevier's Integration, the VLSI Journal*, Vol. 39/1, pp. 12-28, September 2005.

[13] Behrooz Parhami, *Computer Arithmetic Algorithms and Hardware Designs*, Oxford University Press, New York, 2000

[14] S. C. Smith, "Designing NULL Convention Combinational Circuits to Fully Utilize Gate-Level Pipelining for Maximum Throughput," *International Conference on VLSI*, pp. 407-412, June 2004.

[15] S. C. Smith, R. F. DeMara, J. S. Yuan, D. Ferguson, and D. Lamb, "Optimization of NULL Convention Self-Timed Circuits," *Elsevier's Integration, the VLSI Journal*, Vol. 37/3, pp. 135-165, August 2004.

[16] S. C. Smith, "Integrating Asynchronous Digital Design into the Undergraduate Computer Engineering Curriculum," *ASEE Midwest Section Annual Conference*, September 2006.

[17] M. L. Dertouzos, *Threshold Logic: A Synthesis Approach*, Cambridge, M. I. T. Press, 1965.

[18] Lewis & Coates, *Threshold Logic*, New York: John Wiley & Sons, Inc., 1967.

[19] C. Sheng, *Threshold Logic*, New York: Ryerson Press, 1969.

[20] S. R. Mallepalli, S. Kakarla, S. Burugapalli, S. Beerla, S. Kotla, P. K. Sunkara, W. K. Al-Assadi, and S. C Smith, "Implementation of Static and Semi-Static Versions of a Quad-Rail NCL 24+8×8 Multiply and Accumulate Unit, *IEEE Region 5 Technical Conference*, April 2007.

VITA

Samarsen Reddy Mallepalli was born on April 2, 1981 in Wanaparthy, India. He received the degree of Bachelor of Engineering in Electronics and Communications from Muffakham Jah College of Engineering and Technology, Osmania University, Hyderabad in June 2002. After obtaining his bachelor's degree, he worked as an Engineer in Hindustan Aeronautics Limited, Bangalore, India. He joined the Master of Science program in Computer Engineering at the University of Missouri-Rolla in August 2005. He received his Master of Science degree in Computer Engineering at the University of Missouri-Rolla in December 2007. His research with Dr. Scott C. Smith in the University of Missouri-Rolla has concentrated in the area of Asynchronous Delay-Insensitive Digital Design using NULL Conventional Logic.