UNIVERSITY OF CALIFORNIA,

IRVINE

Implementation of a Radix-512 Divider

THESIS

submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in Engineering

by

Alberto Nannarelli

Thesis Committee:
Professor Tomás Lang, Chair
Professor Fadi J. Kurdahi
Professor Nikil D. Dutt

1995

The thesis of Alberto Nannarelli is approved:

Committee Chair

University of California at Irvine
1995

To my beloved parents,

who encouraged me in my studies and made me realize that

"Fatti non foste a viver come bruti
ma per seguir virtute e canoscenza"

"You were not born to live as brutes,
but to follow virtue and knowledge"

La Divina Commedia - Inferno XXVI, 119-120

Contents

List of Figures

List of Tables

Acknowledgement

I am indebted to Professor Tomás Lang for his insight and direction during the development of the project. I am grateful for his confidence in me, and for having had the opportunity to work with him. Our work was funded by MICRO Project (grant 93-088) with the industrial support of Sun Microsystems Inc.

I also wish to thank the members of my thesis committee, Professors Nikil Dutt and Fadi Kurdahi, for their helpful comments and recommendations regarding my work.

In addition, thanks are due to Professors Milos Ercegovac and Javier Bruguera for their suggestions in regard to my design.

A special thank-you goes to Alfred Thordarson whose willingness to discuss my ideas was of great help in writing my thesis.

Abstract of the Thesis

Implementation of a Radix-512 Divider
by
Alberto Nannarelli
Master of Science in Engineering
University of California, Irvine, 1995
Professor Tomás Lang, Chair

The objective of this work is to develop reasonably accurate methods to evaluate the speed and area of arithmetic modules, mainly to compare these to different schemes. Such methods will be of use to the research community to guide its efforts towards schemes that have the potential for successful implementation.

In order to evaluate these methods, we consider a relatively complex arithmetic module, a radix-512 division unit. Previous evaluations of this unit have been done, without an actual implementation, in terms of area and delay using two measures: full-adder units and nand2 units. In this project we implement a complete design of the radix-512 divider and compare its area and delay to the results obtained in the above mentioned evaluations.

Furthermore, we compare the implementation of the radix-512 divider in two different standard cells libraries, $1.2\mu m$ and $0.6\mu m$, and we discuss the impact of sub-micron technologies in the design of these units.

Finally, we compare this radix-512 unit to a simpler one, a radix-4 unit, to see if the use of higher radixes can be effective in the realization of fast arithmetic units.

When the radix-512 divider was implemented, its performance showed that the speed-up over the radix-4 divider is more than double, and that it can be very effective if the priority is to have a fast circuit and the area is of lesser importance.

Introduction

The use of radix-512 recurrence algorithms in divisions could reduce dramatically the number of clock cycles required to calculate the quotient. The drawback is that the circuits are more complicated, and this increases the area needed on the chip and lengthens the clock cycle. An effective implementation is achieved by scaling the operands, as described in [1] and [2], and by rounding the residual to obtain the quotient-digit at each iteration [3].

The objective of this work is to develop reasonably accurate methods to evaluate the speed and area of arithmetic modules, mainly to compare these to different schemes. Such methods will be of use to the research community to guide its efforts towards schemes that have the potential for successful implementation.

In this project we implement a complete design of the radix-512 divider and compare it to the results obtained in [3] and [4], where the evaluation of this unit has been done, without an actual implementation, in terms of area and delay using two measures: full-adder units and nand2 units. We also compare the implementation of the radix-512 divider in two different standard cells libraries, $1.2\mu m$ and $0.6\mu m$, and we discuss the impact of sub-micron technologies in the design of these units. Finally, we compare this radix-512 divider unit to a radix-4 unit ([3] and [5]), to see if the use of higher radixes can be effective in the realization of fast arithmetic units.

The radix-512 divider was implemented, and its performance indicated that the speed-up over the radix-4 divider is more than double, Moreover, it can be very effective if the priority is to have a fast circuit and the area is of lesser importance.

In the first chapter, we describe the project flow and the methodology used in our design. We also present a list of the CAD tools used. In the second chapter, we briefly describe the digit-recurrence division algorithm for radix-512 with scaling and quotient-digit selection by rounding ([3] and [4]). In the third chapter, we explain the design of the divider and how it is sub-divided into blocks in the VHDL models. In the fourth chapter, we show the results of the implementation in two standard cells libraries. We also present the results of the design: delay and area. In the fifth chapter, we first compare the results of our design with those of the earlier evaluations, then the results obtained for the two implementations of the radix-512 divider in the two libraries, and finally, we compare the radix-512 divider unit to a radix-4 divider unit.

There are two appendixes to the thesis. Appendix A contains the structure of the VHDL models and the VHDL descriptions of some modules. Appendix B lists the test vectors used.

Chapter 1
Project Description

The starting point of the project is the algorithm presented in [3] and [4] and the specifications on the bit length for every variable. The project flow is depicted in Figure 1.1.

We implemented the divider using the VHDL language [6] and hierarchical design. We used two different tools in the realization of the divider: Synopsys [7] for the behavioral model and part of the structural design and Compass [8] for part of the structural and physical design. The project flow is described by the following main steps:

- A behavioral model of the divider was developed from the algorithm. Using the Synopsys simulator, some simulations were carried out on this model, choosing a set of test vectors that tested the functionality and the correctness of the results within the bounds stated in the algorithm.

- The unit was divided into functional blocks. This was done manually. Each block represents a different functionality of the system. A block could be either a combinational or a sequential circuit, and a controller was introduced in order to have the correct sequencing in the operations. Then part of these functional blocks were expanded into sub-blocks containing logic gates, adders, latches and multiplexers. To verify these sub-blocks the same set of vectors and simulator was used.

- Using the VHDL format, the divider rtl-model was imported into the Compass environment for the physical design and the layout generation.

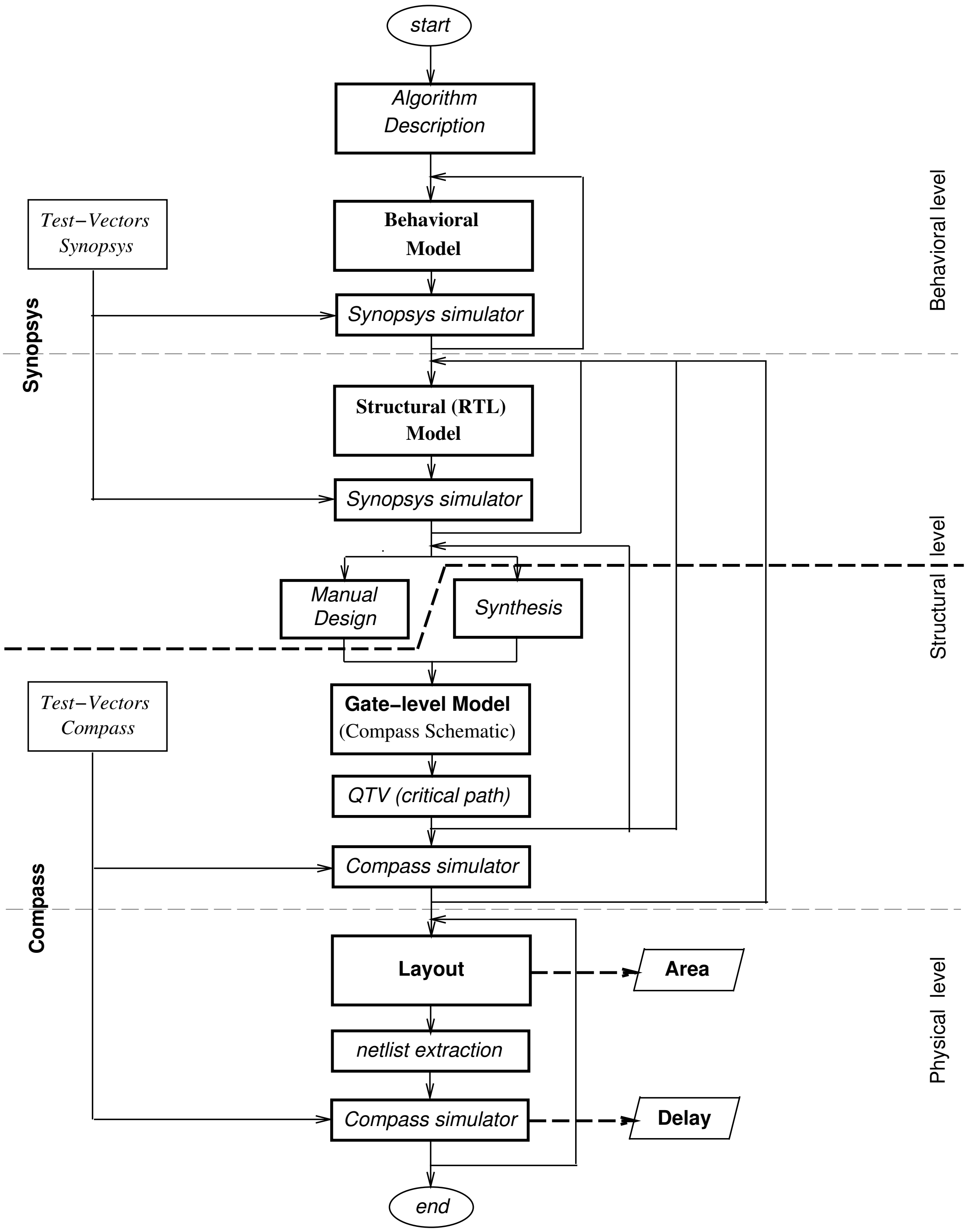

Figure 1.1: Project flow

The original test vectors were converted into new patterns suitable to the Compass mixed-mode simulator.

- Using the Compass ASIC Synthesizer, the schematics for each block were generated. The Synthesizer created the gate-level model based on the synthesis constraints (minimal area or minimal delay) and on the specified standard cell library. Not all the blocks were synthesised, some of them were manually designed using the Compass schematic editor (Compass Logic Assistant).

- QTV was run to determine the critical path. Simulations were then carried out to see if the functionality of the circuit had been maintained.

- The layout was generated in a totally automatic way and then following the netlist extraction, the extracted circuit was again simulated in order to verify the functionality and the minimum clock cycle applicable.

1.1 Synopsys Tools

The Synopsys Graphical Environment (SGE) is a collection of tools for entering design information and for accessing designs graphically during simulation. The Synopsys VHDL System Simulator provides tools for simulating and debugging VHDL circuit descriptions. SGE consists of three main tools:

Schematic Editor - A graphical editor for creating schematic diagrams. Schematics typically consist of symbols, wires, I/O markers, and text.

Symbol Editor - A graphical editor for creating and editing custom symbols, which can be added to a schematic. Symbols represent either primitive cells or functional blocks. SGE allows hierarchical designs; a symbol in one schematic can represent the entire content of another schematic.

Hierarchy Navigator - A graphical tool for navigating through the hierarchical collection of schematics that represents a design. The navigator interacts with the simulator, the debugger, and the waveform viewer; it gives graphical access to the design during the simulation.

1.2 Compass Tools

The Compass design environment provides a number of tools for designing chips. The following tools were used in our project:

ASIC Synthesizer - A design tool that transforms a high-level description written in a hardware description language into an optimized gate-level description. It reads circuit specifications written in VHDL and synthesizes combinational and sequential logic from behavioral descriptions.

Logic Assistant - An interface for high-level logic entry. The Logic Assistant can create, edit, view and plot schematic diagrams and icons. It can be used to place portable library, datapath and synthesized logic into a schematic.

QTV Timing Verifier - A tool that performs a static timing analysis on a complete circuit or subcircuit. It is used after the design has been entered with the Logic Assistant to find potential timing problems in the design. It calculates which paths are the critical paths, the delay along any specified path, and the paths with the greatest delay.

Mixed-Mode Simulator - An interactive, mixed-mode, event-driven simulator. The circuits to be simulated may contain a mixture of transistor-level, gate-level, and behavioral-level components. Circuit descriptions are obtained from the Logic Assistant, Chip Compiler, Logic Synthesizer, and Netlist Extractor. Waveforms can be displayed graphically during the simulation or plotted from history files produced by the simulation. The Mixed-Mode Simulator simulates transistor logic with timing for MOS technologies. It predicts logic levels, approximate voltages, and approximate timing of changes in the circuit nodes. Each transistor functions as a resistive switch that is either off or turned on to some degree, depending on the voltages on its terminals. Circuit nodes have one of four logic values: *HIGH*, *LOW*, *Unknown*, or *Intermediate*.

Chip Compiler - An integrated block/standard cell placement and routing system with a floorplanning stage and an automatic floorplan evaluator. Input to the system is a schematic netlist that may contain any mixture of standard cells and functional blocks.

Chapter 2
Algorithm Description

The starting point of the project is the digit-recurrence algorithm radix-512 with scaling and quotient-digit selection by rounding, presented in [3] and [4]. The algorithm performs a division between two double precision floating point numbers, x and d, that produces the quotient

$$q = \frac{x}{d} \quad .$$

In this algorithm only the mantissa is calculated since sign and exponent of the quotient are attainable simply by comparison and shifting. The ranges of the operands are:

$$1/2 \leq d < 1$$

$$1/2 \leq x < 1$$

and for the quotient we have:

$$1/2 \leq q < 1 \quad .$$

The algorithm requires that $x < d$, and in case $x > d$, we divide x by 2 and increment its exponent.

The digit-recurrence algorithm uses a radix $r = 512 = 2^9$, which means that 9 bits of the quotient are produced every iteration. To apply the quotient-digit selection by rounding, the divisor must be within a determined range. To achieve this, both operands are scaled by a quantity M so that:

$$z = Md$$

and

$$w[0] = Mx$$

and the condition to be satisfied is:

$$0.9995127 = 1 - \frac{r-2}{4r(r-1)} < z < 1 + \frac{r-2}{4r(r-1)} = 1.0004873 \quad .$$

M, truncated to its 13th fractional bit, is calculated by the following expression:

$$M = -\gamma_1 d_{15} + \gamma_2$$

where the two coefficients

$$\gamma_1 = \frac{1}{{d_6}^2 + d_6 2^{-6} + 2^{-15}}$$

$$\gamma_2 = \frac{2d_6 + 2^{-6}}{{d_6}^2 + d_6 2^{-6} + 2^{-15}}$$

are also truncated to their 13th fractional bit and in the range:

$$1 < \gamma_1 < 4 \qquad\qquad 2 < \gamma_2 < 4 \quad .$$

d_{15} and d_6 are the divisor d truncated to its 15th and 6th bit respectively.

The recurrence to be executed is:

$$w[j+1] = rw[j] - q_{j+1} z \qquad\qquad j = 0, 1, \ldots 5$$

with the quotient-digit selection:

$$q_{j+1} = \lfloor \hat{y} + 1/2 \rfloor$$

where:

$w[j]$ is the residual after iteration j.

$w[0] = Mx$.

q_{j+1} is the quotient digit generated in iteration j. $q_j = \{-511, \ldots, 0, \ldots, 511\}$

$\hat{y} = \{rw[j]\}_2$ truncated to its 2nd fractional bit.

At the end, q must be rounded according to the sign of the residual of the last iteration. If the last residual is positive we have to add 1 in the least significant position before the rounding. We do not add anything if it is negative.

To execute the recurrence two multiplications and one addition are required. The residual $w[j]$ and the quotient digit q_{j+1} are both in carry-save representation to avoid carry-propagation in the addition. Multiplying a number by $r = 512 = 2^9$ is equivalent to the shifting of its binary representation by 9 positions to the left. The other multiplication is performed by recoding one of the operands [9]. Recoding the multiplier into radix-4 representation reduces the number of partial products and makes the operation faster. The recoded operand is in Signed Digit representation and each digit can assume the values $\{-2, -1, 0, 1, 2\}$.

The algorithm can be represented by the block diagram in Figure 2.1.

Since the two operations:

$$M = -\gamma_1 d_{15} + \gamma_2$$

and

$$w[j+1] = 512w[j] - q_{j+1}z$$

can be executed in the same unit, in order to reduce the area we eliminated the multiplier-accumulator required for the computation of M and performed this operation with the main multiplier-accumulator. The resulting block diagram is shown in Figure 2.2.

The algorithm is divided into a preamble and a body. The preamble is executed once, while the body is executed as many times as required. In case of radix-512, the body, or recurrence, is executed six times. The preamble operations are: M calculation and the scaling of the two operands. A total of ten cycles is needed to perform one division. We can list the operations for each cycle:

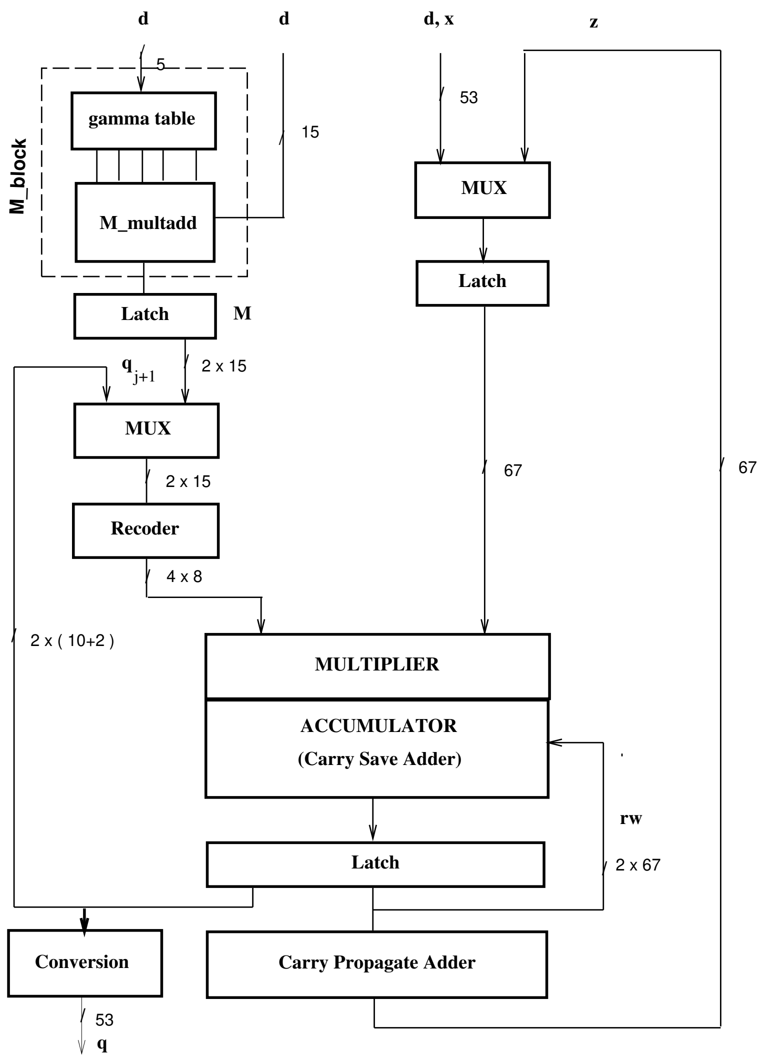

Figure 2.1: Block diagram of divider

1. M is calculated in **MultAdd**. d is stored in **Latch1** and M is stored in **Latch2**.

2. Md is calculated in **MultAdd**. x is stored in **Latch1** and Md is stored in **Latch3**.

3. Mx is calculated in **MultAdd**, Meanwhile the sum and carry parts of Md are assimilated in **Cpa**. $Mx = w[0]$ is stored in **Latch3**, $Md = z$ is stored in **Latch1**.

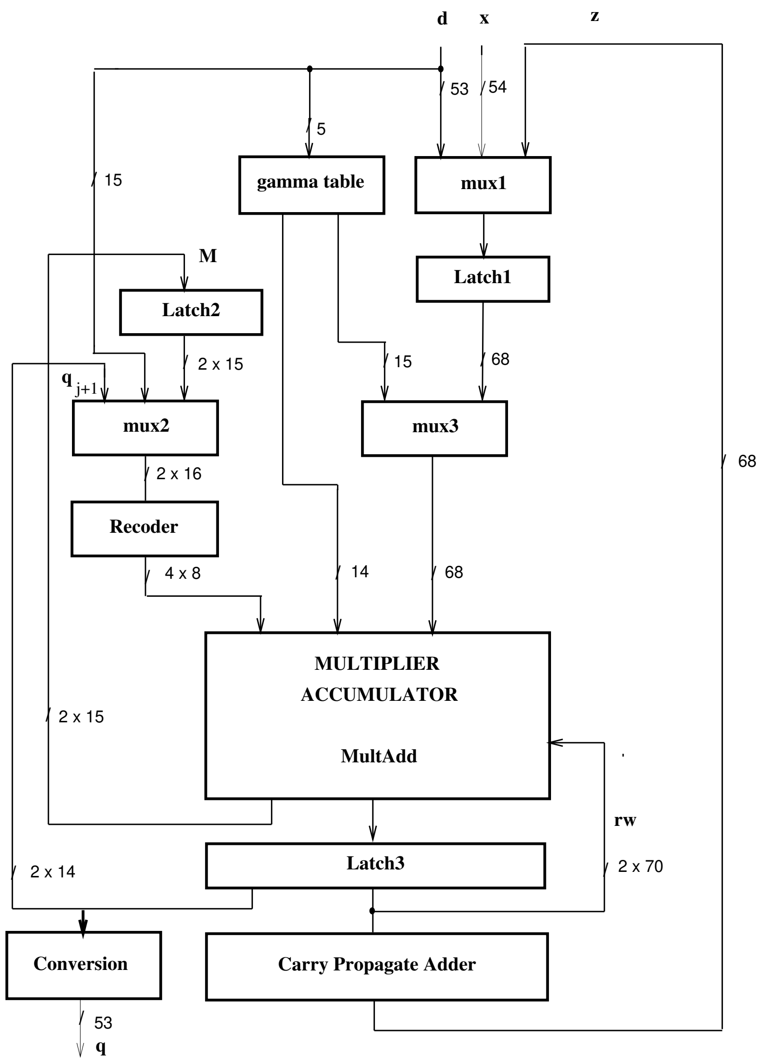

Figure 2.2: Block diagram of modified divider

4. q_1 is extracted and $w[1]$ is calculated in **MultAdd**. $w[1]$ is stored in **Latch3**.

5. q_2 is extracted and $w[2]$ is calculated in **MultAdd**. $w[2]$ is stored in **Latch3**.

6. q_3 is extracted and $w[3]$ is calculated in **MultAdd**. $w[3]$ is stored in **Latch3**.

7. q_4 is extracted and $w[4]$ is calculated in **MultAdd**. $w[4]$ is stored in **Latch3**.

8. q_5 is extracted and $w[5]$ is calculated in **MultAdd**. $w[5]$ is stored in **Latch3**.

9. q_6 is extracted and $w[6]$ is calculated in **MultAdd**. $w[6]$ is stored in **Latch3**.

10. The carry and sum parts of the last residual $w[6]$ (remainder) are assimilated in **Cpa** to determine its sign and the final rounding is done.

<table>
<tr><th>cycle</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th></tr>
<tr><td>Latch1</td><td></td><td>d</td><td>x</td><td colspan="7">z</td></tr>
<tr><td>Latch2</td><td></td><td colspan="9">M</td></tr>
<tr><td>Multadd</td><td>M</td><td>Md</td><td>w[0]</td><td>w[1]</td><td>w[2]</td><td>w[3]</td><td>w[4]</td><td>w[5]</td><td>w[6]</td><td></td></tr>
<tr><td>Latch3</td><td colspan="2"></td><td>Md</td><td>w[0]</td><td>w[1]</td><td>w[2]</td><td>w[3]</td><td>w[4]</td><td>w[5]</td><td>w[6]</td></tr>
</table>

Figure 2.3: Cycles and operations

Figure 2.3 shows that in the last cycle **MultAdd** is idle. This last cycle can then be used to calculate M from the next divisor, if there is a sequence of divisions. By overlapping M calculation to the last cycle we can increase the throughput by 10%.

Bit width is important to achieve the required precision after the truncations in the iteration steps and the final rounding.

The IEEE double-precision standard defines a format of 64 bits for this representation [9]. These 64 bits are then divided into three fields: a sign field (1 bit), an exponent field (11 bits) and a mantissa field (52 bits). Because numbers in this representation are normalized with the mantissa in the range $1 < x < 2$ the most significant bit is always 1 and can be omitted (hidden bit). Thus the real mantissa is composed of 52 bits plus a 1 in the most significant position, giving a total of 53 bits for mantissa representation. Since our divider works in the range $0.5 \leq d < 1$, we must divide the IEEE mantissa by 2 and consequently increment the exponent. This is equivalent to shifting the mantissa to the right one position obtaining, in this case, 53 fractional bits.

$$d \;:\quad 0\,.\,d_{[-1]}\;d_{[-2]}\;\ldots\;d_{[-53]}$$

We use the notation $a_{[n]}$ to refer to the bit with weight 2^n in the binary representation of a.

Because the algorithm also requires that $d > x$, when this condition is not satisfied, we must again divide the mantissa by 2 and increment the exponent. In this case, the right shift will bring a 0 in the most significant position and to not loose precision we have to take into account also the shifted out bit increasing the number of bits of the representation from 53 to 54.

$$x \;:\quad 0\,.\,0\;x_{[-2]}\;\ldots\;x_{[-54]}$$

According to [4] the scaling factor M is in the range:

$$0 < M < 2$$

with 13 fractional bits. A total of 15 bits is required to store M. Because the scaled operands can be greater than 1, for the z and $w[j]$ representation we need a total of $54+13+1+1 = 69$ bits: 1 sign bit, 1 integer bit and 67 fractional bits. To have the correct recoding of q_j in carry-save format, as explained below, we add an extra integer bit. The number of bits needed to store the partial remainder $w[j]$ is 70. To store z, which is always positive, we need only one integer bit and 67 fractional bits for a total of 68 bits.

$$z \;:\quad z_{[0]}\,.\,z_{[-1]}\;z_{[-2]}\;\ldots\;z_{[-67]}$$

During the design, the following modifications to the algorithm were obtained:

- In the first iteration the value of q could be 512, because it is obtained by rounding $512 \cdot w[0] = 512 \cdot Mx$ that can assume the value

$$512 \cdot 1.0004873 = 512.2495$$

in the upper bound. This requires an additional bit to represent the quotient digit, as well as the residual.

- In the carry-save representation of $-M$ if both the MSBs are 1, to have the correct sign after the recoding, they must be changed to 0. This is done by making

$$Ms_{[1]} = Ms_{[1]} + \overline{Mc_{[1]}}$$

$$Mc_{[1]} = Mc_{[1]} + \overline{Ms_{[1]}}$$

- The same problem occurs when recoding q_s and q_c, but in this case, since the sign is not known, the only solution is to extend by one bit the carry-save representation of q. This also requires the extension of w_s and w_c, from 2 to 3 integer bits.

- The inclusion of one additional fractional bit for the residual because of the shifting by one bit required to handle the case $x \geq d$.

Chapter 3
VHDL Models

In the design we followed a hierarchical modeling approach. First, a behavioral model of the divider was developed, using the VHDL language and the Synopsys simulator. Then, after having verified the behavioral model, the unit was decomposed into functional blocks.

3.1 Behavioral Model

From the algorithm we developed a behavioral model of the divider. The behavioral level is the highest level of abstraction. At this level the functionality of the system is described using the simplest VHDL syntax. The division algorithm was implemented using the arithmetic capability of the VHDL language. More in detail:

- The recoder was not implemented and the multiplication was performed using the VHDL arithmetic operation.
- The residual was represented in carry-save format.
- The rounding of the quotient-digit was done by adding 0.5.
- The conversion was done by shifting and adding the quotient-digit obtained at each iteration to the partial quotient.

We simulated the behavior of the model, using a set of input test vectors and a simulator, in order to test the functionality of this model.

The behavioral model is realized with two segments of VHDL code: the main body that implements the algorithm and a second segment that implements the functions (such as left and right shifts, carry-save additions, 2's complements) used in the main body. The VHDL code is shown in Appendix A.

The VHDL model was tested using ten specially selected test vectors and one hundred vectors generated at random. Some of these ten vectors were selected to determine errors in the calculation of the quotient, especially for easy debugging purposes, others were selected to detect errors occurring at the boundary values for dividend and divisor. These ten vectors are shown in Table 3.1. Refer to Appendix B for a description of the one hundred random vectors.

	x	d	q	comments
1	0.750000	0.875000	0.857143	
2	0.656250	0.875000	0.750000	
3	0.500000	0.600000	0.833333	
4	0.500000	$1.0 - 2^{-53}$	$0.5 + 2^{-53}$	d maximum value
5	0.250000	0.500000	0.500000	x, d minimum values
6	0.250000	$1.0 - 2^{-53}$	0.250000	x min , d max
7	$1.0 - 2^{-52} - 2^{-53}$	$1.0 - 2^{-53}$	$1.0 - 2^{-52}$	x,d $\simeq$ d upper bound
8	0.500000	$0.5 + 2^{-53}$	$1.0 - 2^{-52}$	x,d $\simeq$ d lower bound
9	$0.5 - 2^{-53}$	0.5	$1.0 - 2^{-52} - 2^{-53}$	x,d $\simeq$ d lower bound
10	$0.5 - 2^{-53}$	$0.75 - 2^{-53}$	0.666667	q periodic

Table 3.1: Ten special test vectors

3.2 Structural Model

In this section the structural model and the bit-width of all the signals are described. The structural model was obtained by manually decomposing the behavioral model into functional blocks. This model presents both blocks expanded in sub-blocks and blocks that are still behavioral. Those blocks were synthesized directly by Compass from behavioral to gates. The structure of the design and some examples of VHDL code are shown in Appendix A. The Synopsys structural model of the divider is shown in Figure 3.1. Table 3.2 shows the significant data buses or signals.

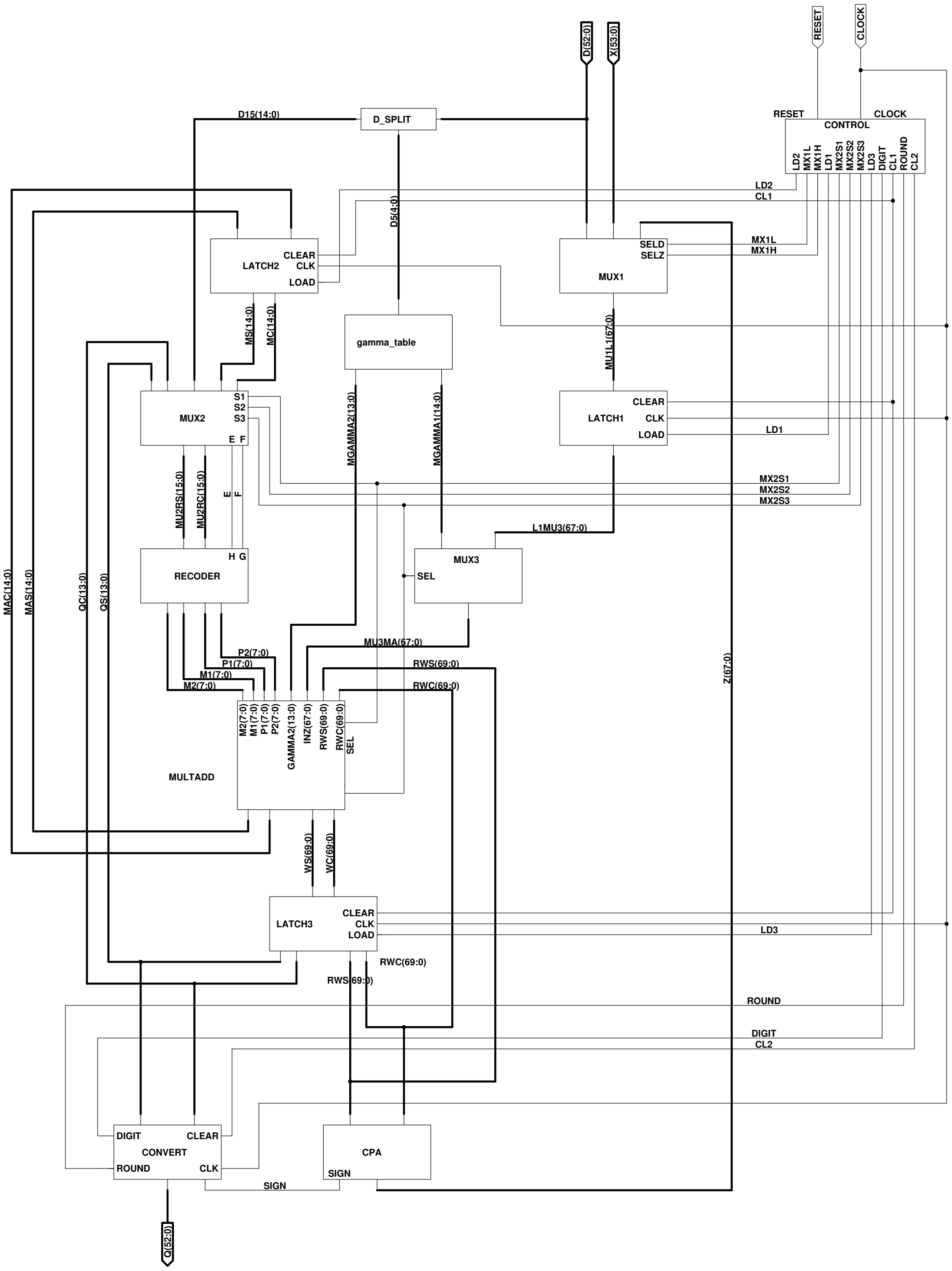

Figure 3.1: Synopsys structural model

bus	from	to	bits
D	Input	Mux1 & d_Split	53
X	Input	Mux1	54
MU1L	Mux1	Latch1	68
LUMU3	Latch1	Mux3	68
MU3MA	Mux3	MultAdd	68
D5	d_Split	gamma_Table	5
D15	d_Split	Mux2	15
MGAMMA1	gamma_Table	Mux3	15
MGAMMA2	gamma_Table	MultAdd	14
MAS	MultAdd	Latch2	15
MAC	MultAdd	Latch2	15
MS	Latch2	Mux2	15
MC	Latch2	Mux2	15
MU2RS	Mux2	Recoder	16
MU2RC	Mux2	Recoder	16
E	Mux2	Recoder	1
F	Mux2	Recoder	1
M2	Recoder	MultAdd	8
M1	Recoder	MultAdd	8
P1	Recoder	MultAdd	8
P2	Recoder	MultAdd	8
WS	MultAdd	Latch3	70
WC	MultAdd	Latch3	70
RWS	Latch3	MultAdd & Cpa	70
RWC	Latch3	MultAdd & Cpa	70
QS	Latch3	Convert & Mux2	14
QC	Latch3	Convert & Mux2	14
Z	Cpa	Mux1	68
SIGN	Cpa	Convert	1
Q	Convert	Out	53

Table 3.2: List of signal names and bus width

3.2.1 Controller

The **Control** block sends control signals to the other blocks to synchronize all the operations of the divider. The circuit is driven by a clock signal. At this point of the description the clock cycle length is still unknown, it will be set later when the critical path on the circuit is calculated. Figure 3.2 shows the timing of the signals. The states of the controller are ten, as many as the division cycles.

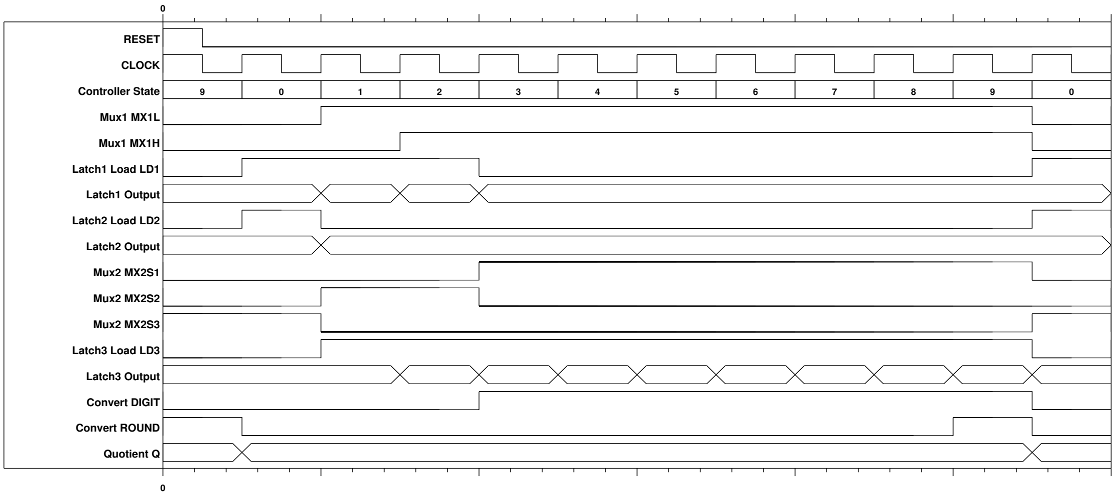

Figure 3.2: Control signals

3.2.2 MultAdd

MultAdd executes both the multiplication and the addition in the recurrence. The adder is also used to reduce the number of the partial products of the multiplication to the final carry-save representation. Because it is used for calculating M, scaling the operands and calculating the next residual, some modifications are required to have the divider working properly. The operation done by **MultAdd** is:

$$sum = A + BC$$

the four different operations to be accommodated in it are:

1. $-M = \gamma_1 d_{15} - \gamma_2$

2. $z = Md$

3. $w[0] = Mx$

4. $w[j+1] = 512w[j] - q_{j+1}z$

B is the output of the recoder and to calculate the difference in case 1 and 4, we must incorporate the $-$ sign in the recoder, recoding the negative of the input $(-R)$. Thus the combination of the **Recoder** and **MultAdd** is

$$sum = A - RC$$

(Figure 3.3). Comparing Figure 3.3 to the **MultAdd** block in Figure 3.1, we can

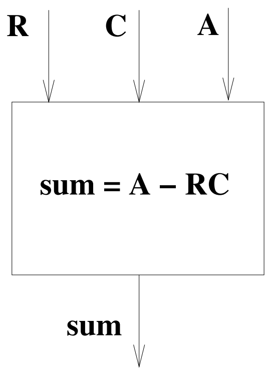

Figure 3.3: $sum = A - RC$

see that following correspondences:

A	either **GAMMA2** or the pair **RWS RWC**
R	**M1 M2 P1 P2**
C	**INZ**

In the four cases listed above we have:

1. $A = -\gamma_2$, $R = d_{15}$, $C = -\gamma_1$, $sum = -M$

$$-M = -\gamma_2 - d_{15}(-\gamma_1)$$

bit #	...	29	28	...	15	14	...	0
A	...	0	$-\gamma_{2[0]}$	...	$-\gamma_{2[-13]}$	0	...	0
C	...	1	1	...	1	$-\gamma_{1[1]}$	...	$-\gamma_{1[-13]}$
R	8 digits of d_{15}							
$-RC$	...	$-RC_{[1]}$	$-RC_{[0]}$	...	...	...	...	$-RC_{[-28]}$
sum	-	$-M_{[1]}$	$-M_{[0]}$	...	$-M_{[-13]}$	-	-	-

In this case only the thirty least significant bits of **MultAdd** are used.

2. $A = 0$, $R = -M$, $C = d$, $sum = z$

$$z = 0 - (-M)d$$

bit #	...	67	66	...	52	...	1	0
A	...	0	0	...	0	...	0	0
C	...	0	0	...	$d_{[-1]}$	...	$d_{[-53]}$	0
R	8 digits of $-M$							
$-RC$	...	$Md_{[0]}$	$Md_{[-1]}$	...	$Md_{[-14]}$	...	$Md_{[-66]}$	$Md_{[-67]}$
sum	-	$z_{[0]}$	$z_{[-1]}$	...	$z_{[-14]}$	...	$z_{[-66]}$	$z_{[-67]}$

3. $A = 0$, $R = -M$, $C = x$, $sum = w[0]$

$$w[0] = 0 - (-M)x$$

bit #	69	68	...	53	52	...	1	0
A	0	0	...	0	0	...	0	0
C	0	0	...	0	$x_{[-1]}$	...	$x_{[-53]}$	$x_{[-54]}$
R	8 digits of $-M$							
$-RC$	$Mx_{[2]}$	$Mx_{[1]}$	...	$Mx_{[-13]}$	$Mx_{[-14]}$	...	$Mx_{[-66]}$	$Mx_{[-67]}$
sum	$w_{[2]}$	$w_{[1]}$	...	$w_{[-13]}$	$w_{[-14]}$	...	$w_{[-66]}$	$w_{[-67]}$

4. $A = 512w[j]$, $R = q_{j+1}$, $C = z$, $sum = w[j+1]$

$$w[j+1] = 512w[j] - q_{j+1}z$$

bit #	69	68	67	...	9	8	...	0
A	$w_{[-7]}$	$w_{[-8]}$	$w_{[-9]}$	...	$w_{[-67]}$	0	...	0
C	-	-	$z_{[0]}$	...	$z_{[-58]}$	$z_{[-59]}$	...	$z_{[-67]}$
R	6 digits of q_{j+1}							
$-RC$	$qz_{[11]}$	$qz_{[10]}$	$qz_{[9]}$	...	$qz_{[-13]}$	$qz_{[-14]}$	...	$qz_{[-58]}$
sum	$w_{[2]}$	$w_{[1]}$	...	$w_{[-13]}$	$w_{[-14]}$	...	$w_{[-66]}$	$w_{[-67]}$

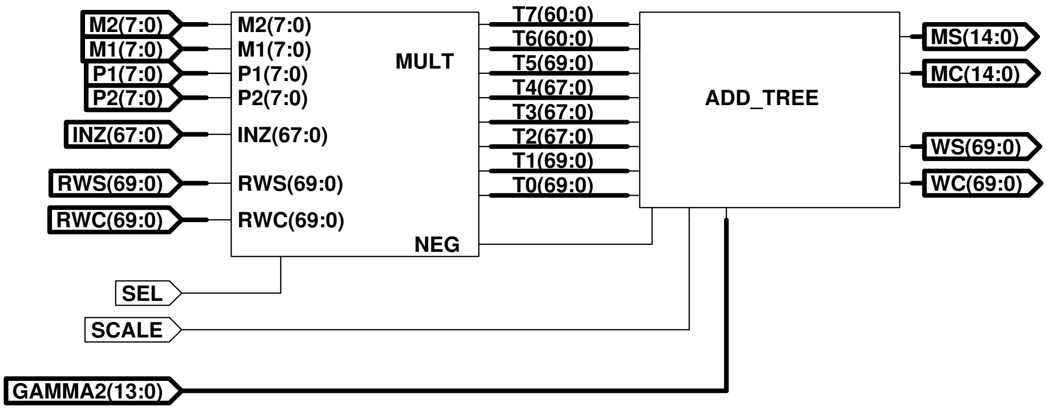

Figure 3.4: MultAdd block diagram

As shown in Figure 3.4, **MultAdd** can be split in two sub-blocks: the first one calculates the partial products and the second one adds them. The multiplier generates the partial products $t_0, t_1, \ldots, t_7$ from the recoded multiplier and a 2's complement multiplicand.

- In M calculation, the recoded multiplier is d_{15} (15 bits) that produces 8 partial products $(t_0, t_1, \ldots, t_7)$. Also $-\gamma_2$ should be added.

$$s = t_0 + t_1 + t_2 + t_3 + t_4 + t_5 + t_6 + t_7 + (-\gamma_2)$$

- In the scaling, the recoded multiplier is $-M$ (15 bits).

$$s = t_0 + t_1 + t_2 + t_3 + t_4 + t_5 + t_6 + t_7$$

- In the recurrence, the recoded multiplier is q_j that is 12 bits. In this case only 6 partial products are generated (buses t_0-t_5) and the two buses t_6 and t_7 are used to add the carry-save represented shifted residual $(rw[j])$.

$$s = t_0 + t_1 + t_2 + t_3 + t_4 + t_5 + rws + rwc$$

In the latter two cases we need a 8:2 adder, in the first a 9:2.

Signals **SEL** and **SCALE** select the different operations.

Multiplication

In **Mult** one recoded (radix-4) multiplier is multiplied by a 2's complement multiplicand. This is a standard radix-4 multiplication [9]. Every partial product is 2 position left-shifted with respect to the preceding one, because of the radix-4 representation of the digits. Sign extension is required to obtain the correct result.

```
sssssssxxxxxxxxxxxxxxxxxx
sssssxxxxxxxxxxxxxxxxxx00
sssxxxxxxxxxxxxxxxxxx0000
sxxxxxxxxxxxxxxxxxx000000
```

When a negative digit is encountered, we bit complement the corresponding partial product (before the shift) and we put a 1 in the next product in correspondence of the least significant bit of the actual partial product. If the digit is -1 we have:

```
sssssssxxxxxxxxxxxxxxxxxx
sssscccccccccccccccccccc00
sssxxxxxxxxxxxxxxxxxx0100
sxxxxxxxxxxxxxxxxxx000000
```

being $c = \overline{x}$. And if the digit is -2 we have:

```
sssssssxxxxxxxxxxxxxxxxxx
ssscccccccccccccccccccc000
sssxxxxxxxxxxxxxxxxxx1000
sxxxxxxxxxxxxxxxxxx000000
```

Putting 1 in place of 0 in the shift extension doesn't change the complexity of the circuit. These bits in the first and second shift extension are simply **M2** and **M1** of the preceding partial product respectively.

From the examples above we can see that this method of complementing is not applicable if the last partial product is negative. This will never happen when we are doing the scaling: the last partial product is generated by the most significant digit of M which is always positive.

In the recurrence the last partial product is t_5 that is generated by the MSD of q_j. This digit could be negative, and the solution above cannot be adopted. Because

$$-512 < q_j \leq 512$$

its MSD can be either 0 or 1 or -1. If it is -1 the second digit must be positive, because it is impossible to be in the range when

$$(-1) \cdot 4^5 + (-1) \cdot 4^4 = -1280$$

Knowing that, we can modify the procedure to calculate the bit complement. In this case we bit complement the shifted partial product and later in **Add_Tree** add a 1, in the least significant position.

```
ssssssxxxxxxxxxxxxxxxxxx
ssssxxxxxxxxxxxxxxxxxx00
ssxxxxxxxxxxxxxxxxxx0000
sxxxxxxxxxxxxxxxxx111111
000000000000000000000001
```

This bit, a sort of carry, is the signal **NEG**.

The block diagram is shown in Figure 3.5. It is composed of 8 sub-blocks (**mp_pp[0-7]**) that generate the partial products and a multiplexer **mp_mux** that selects either the 8 partial products, needed in the scaling, or 6 partial products and the carry-save representation of the residual in the recurrence.

Addition

Add_Tree is a tree of full-adders that reduces nine/eight inputs to two outputs (see Figure 3.6). When the adder is used to calculate M, 9 inputs are required instead of 8. Block **Csa_15mx** (Figure 3.7) provides the necessary multiplexing. It is only a 15 bit adder, as many bits as required to add $-\gamma_2$ to the partial products

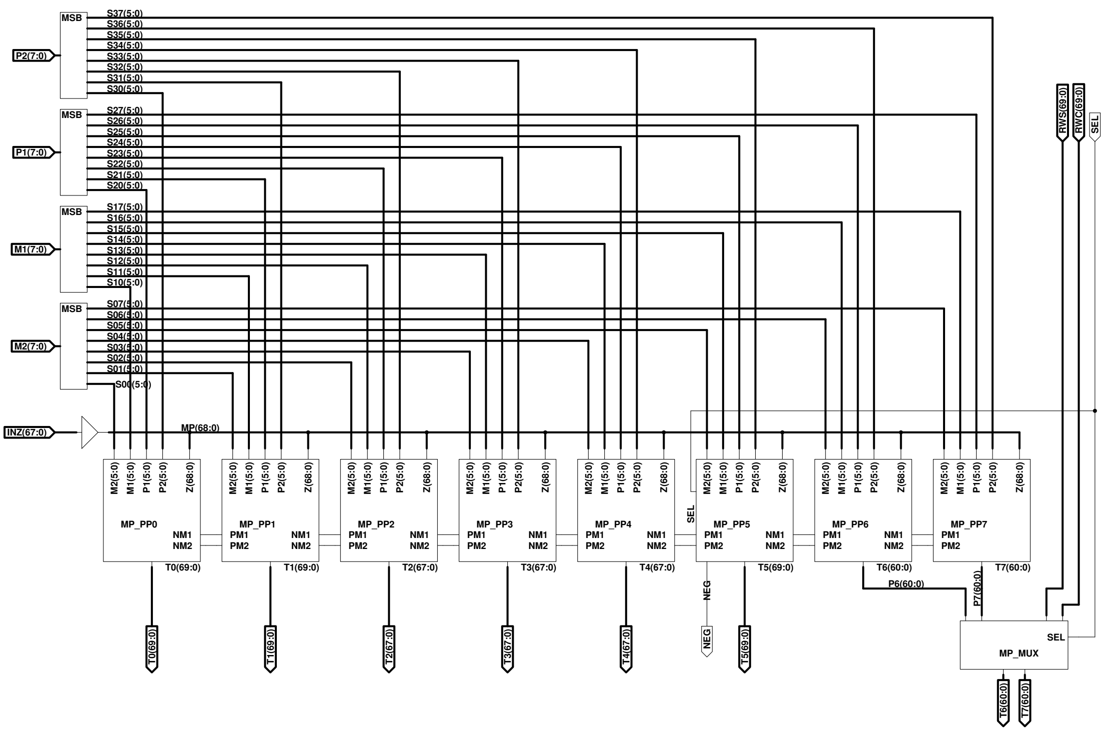

Figure 3.5: Mult block diagram

$\gamma_1 d_{15}$. Block **M_out** adjusts the bits of M for a correct recoding. In the carry-save representation of $-M$ if both the MSB are 1, they must be changed to 0. This is done by making

$$Ms_{[1]} = Ms_{[1]} + \overline{Mc_{[1]}}$$

$$Mc_{[1]} = Mc_{[1]} + \overline{Ms_{[1]}}$$

The carry-save adders are of different sizes to minimize the number of bits (i.e. area) necessary in the assimilation of the addends.

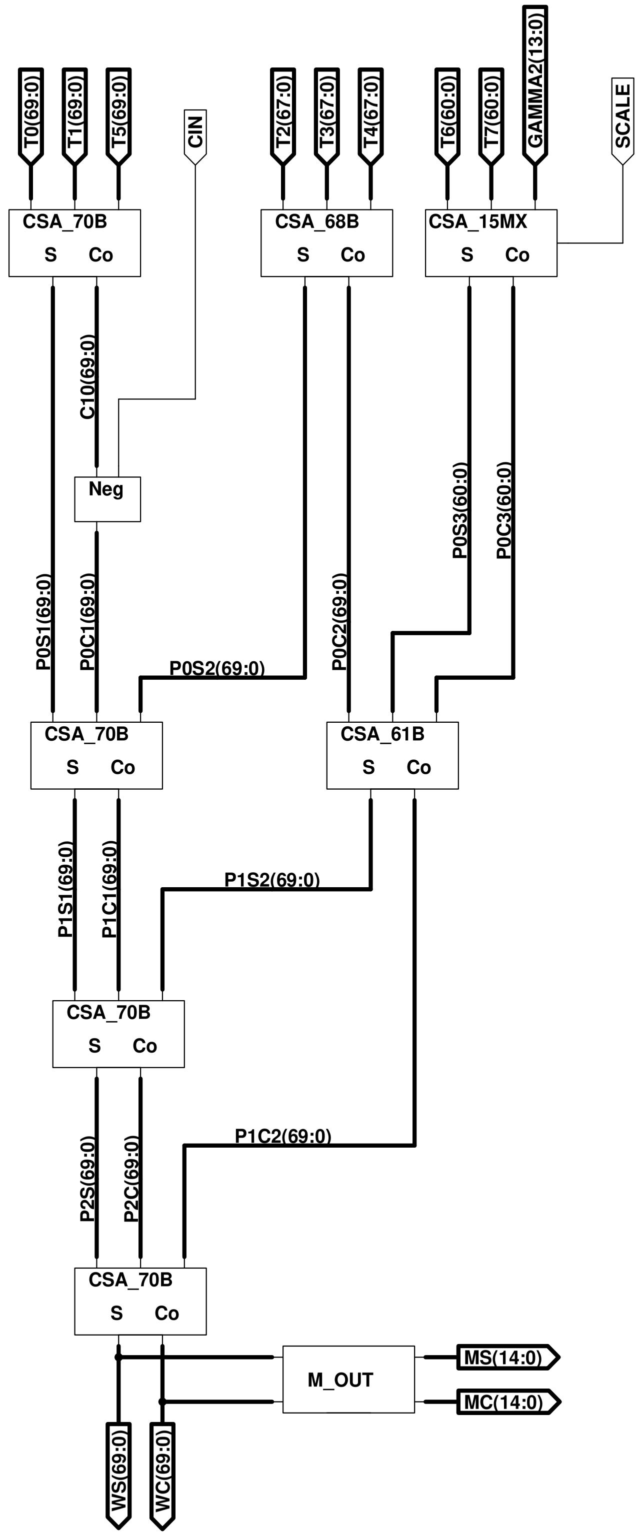

Figure 3.6: Add_Tree block diagram

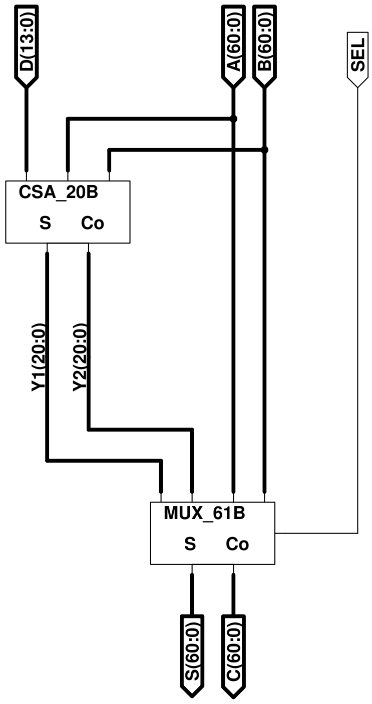

Figure 3.7: Csa_15mx block diagram

3.2.3 Gamma_Table

Gamma_Table is the block that generates $-\gamma_1$ and $-\gamma_2$ to calculate $-M$ by:

$$-M = (-\gamma_1)(-d_{15}) - \gamma_2$$

$$\gamma_1 = \frac{1}{{d_6}^2 + d_6 2^{-6} + 2^{-15}}$$

$$\gamma_2 = \frac{2d_6 + 2^{-6}}{{d_6}^2 + d_6 2^{-6} + 2^{-15}}$$

where d_{15} and d_6 are d truncated to its 15th and 6th bit respectively. Also γ_1 and γ_2 are truncated to their 13th fractional bit and in the range:

$$1 < \gamma_1 < 4 \qquad\qquad 2 < \gamma_2 < 4$$

Being d greater than 0.5 the MSB of d_6 is always 1. So we have $2^5 = 32$ different values of $-\gamma_1$ and $-\gamma_2$ (Table 3.3).

$(0.5 - d_6)2^6$	$-\gamma_1$	$-\gamma_2$
0	000001111100101	00000111110101
1	000101100110010	00010110111000
2	001000111011110	00100101000100
3	001011111111101	00110010011100
4	001110110011100	00111111000110
5	010001011001001	01001011000100
6	010011110010001	01010110011010
7	010101111111101	01100001001100
8	011000000010110	01101011011100
9	011001111100100	01110101001100
10	011011101101110	01111110011110
11	011101010111011	10000111010110
12	011110111001111	10001111110100
13	100000010110000	10010111111001
14	100001101100001	10011111101001
15	100010111101000	10100111000100
16	100100001000111	10101110001011
17	100101010000010	10110101000000
18	100110010011011	10111011100100
19	100111010010101	11000001110111
20	101000001110010	11000111111011
21	101001000110101	11001101110000
22	101001111011111	11010011011000
23	101010101110011	11011000110010
24	101011011110001	11011110000001
25	101100001011011	11100011000100
26	101100110110011	11100111111011
27	101101011111010	11101100101001
28	101110000110001	11110001001100
29	101110101011001	11110101100110
30	101111001110010	11111001110111
31	101111101111111	11111101111111

Table 3.3: $-\gamma_1$ and $-\gamma_2$ table

Being $-4 < -\gamma_2 < -2$, we have $10x.xxxx$ and to minimize the circuit we implement only 1 integer bit.

$-\gamma_1 \Rightarrow 2$ integer + 13 fractional bits = 15 bits

$-\gamma_2 \Rightarrow 1$ integer + 13 fractional bits = 14 bits

3.2.4 Latches

Latch1 stores the multiplicand selected by **Mux1**. **Latch2** stores $-M$ in carry-save format. **Latch3** stores the residual after each step $w[j]$ in carry-save format. Also the quotient digits q_j are extracted from there.

3.2.5 Multiplexers

Mux1 selects either d, x or z and aligns them for the multiplication. **Mux2** selects either d_{15} or the carry-save representation of either $-M$ or q_j. When d_{15} is selected, its value is assimilated in the sum part and the carry part is set to zero. Also are calculated e (signal **E**) and f (signal **F**) bits necessary for the correct rounding of the quotient digit.

```
S : xxxxxxxxxxxxxx.ab
C : xxxxxxxxxxxxxx.cd
```

$$e = a + c$$

$$f = bd\overline{(a \oplus c)}$$

Mux3 selects either $-\gamma_1$ or **L1MU3**. In case of $-\gamma_1$, it is aligned to the LSB and the empty positions are filled with zeros.

3.2.6 Recoder

This block is used to recode the multiplier to have a faster multiplication. Recoding this multiplier into radix-4 representation with digits in the range $(-2, \ldots, 2)$ makes the operation faster and the circuits simpler. The recoded operand is in Signed Digit representation and every digit can assume the values $\{-2,-1,0,1,2\}$. To represent these five different values four signals are used: **M2**, **M1**, **P1**, **P2**. Only one of them can be set to 1, if all of four are 0 the signed digit is 0 (Table 3.4).

Digit	**M2**	**M1**	**P1**	**P2**
-2	1	0	0	0
-1	0	1	0	0
0	0	0	0	0
1	0	0	1	0
2	0	0	0	1

Table 3.4: Correspondence between digits and signals

An implementation of this recoding is described in [3], here we utilize a variation that is described in [10]. Since the value d_k of each radix-4 digit in the carry-save representation is in the range 0 to 6, the recoding consists of three steps, as follows:

1. Obtain t_{k-1} and w_k such that

$$d_k = 4t_{k-1} + w_k$$

   where t_{k-1} is a transfer bit and $0 \leq w_k \leq 4$.

2. The transfer bit t_k generated with digit d_{k+1} is added to w_k, resulting in $w_k + t_k \in \{0,\ldots,5\}$. Moreover, to achieve $Z_k \in \{-2,-1,0,1,2\}$, a second transfer bit h_{k-1} is generated such that

$$w_k + t_k = 4h_{k-1} + v_k$$

   with $v_k \in \{-2,\ldots,1\}$.

3. Finally,

$$Z_k = v_k + h_k$$

As seen in Figure 3.8, three digits d_k, d_{k+1} and d_{k+2}, in an overlapped fashion, are used to obtain one digit Z_k since transfer bit t_{k-1}, obtained from d_{k+2}, is needed to compute h_k. The actual implementation is shown in Figure 3.9.

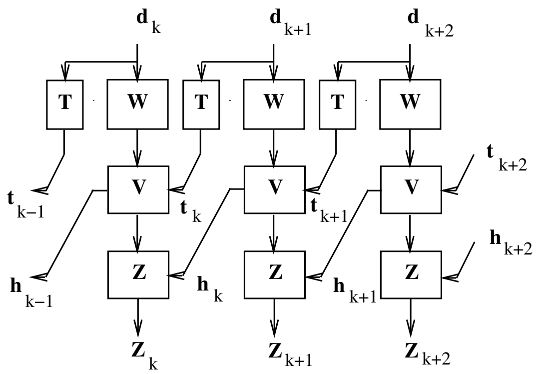

Figure 3.8: Recoder block diagram

- For step 1, we make $t_{k-1} = \lfloor (a+b)/2 \rfloor$ and $w_k = 2(a+b) \bmod 2 + (c+d)$. Being a and b the MSBs of the carry-save representation of the radix-4 digit. Moreover, we represent the digits of $w_k = 2w_1 + w_0$ by the vectors (w_{11},w_{10}) and (w_{02},w_{01},w_{00}) so that $w_{xy} = 1$ if $w_x = y$.

- For step 2, we obtain $h_{k-1} = 1$ if $w_k + t_k \geq 2$. Moreover,

$$v_k = w_k + t_k - 4h_{k-1} \quad (-2 \leq v_k \leq 1)$$

  is represented in a 1-out-of-4 code.

- Step 3 produces Z_k in a 1-out-of-5 code, which is directly used to select the multiples of the multiplicand (signals: **M2**, **M1**, **P1**, **P2**). The value 0 is not required.

Also the rounding of the quotient digit q_j can be incorporated in the recoding by the inputs $t_0 = e$ and $h_0 = f$.

The **Recoder** consists of 8 stages (16 bits), as shown in Figure 3.10. Each stage recodes one radix-4 digit of the carry-save representation.

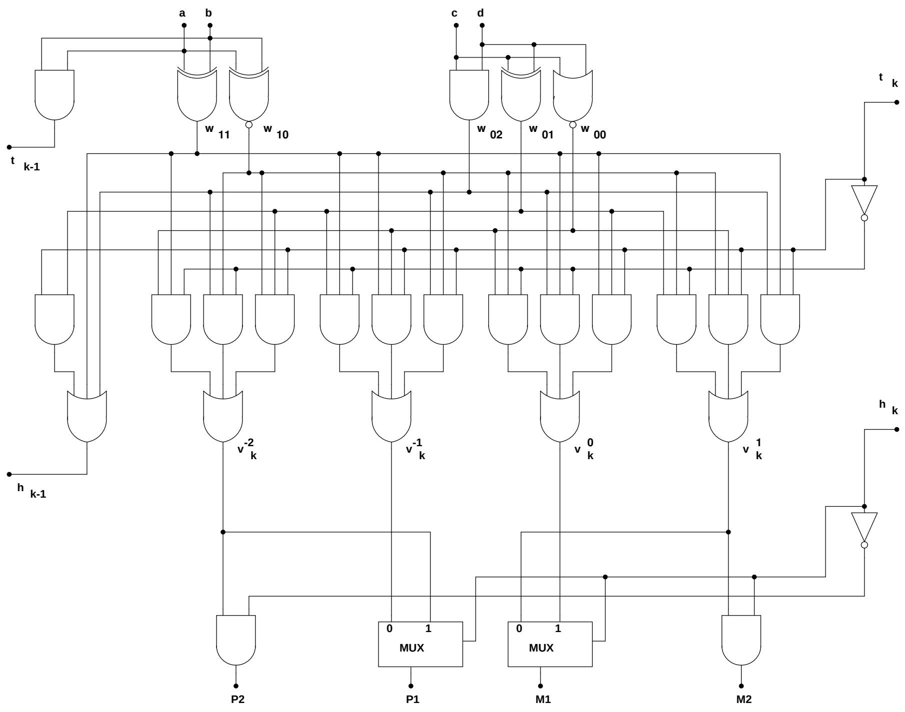

Figure 3.9: Recoder schematic

3.2.7 Convert

This block converts the quotient digits to the conventional representation in 2's complement. It also rounds the quotient. The algorithm used is the on-the-fly conversion that performs this conversion as the digits of the quotient are produced and does not require a carry-propagate adder [3].

The partial result is stored in two registers Q and QM. These two registers are updated every iteration by the following rules:

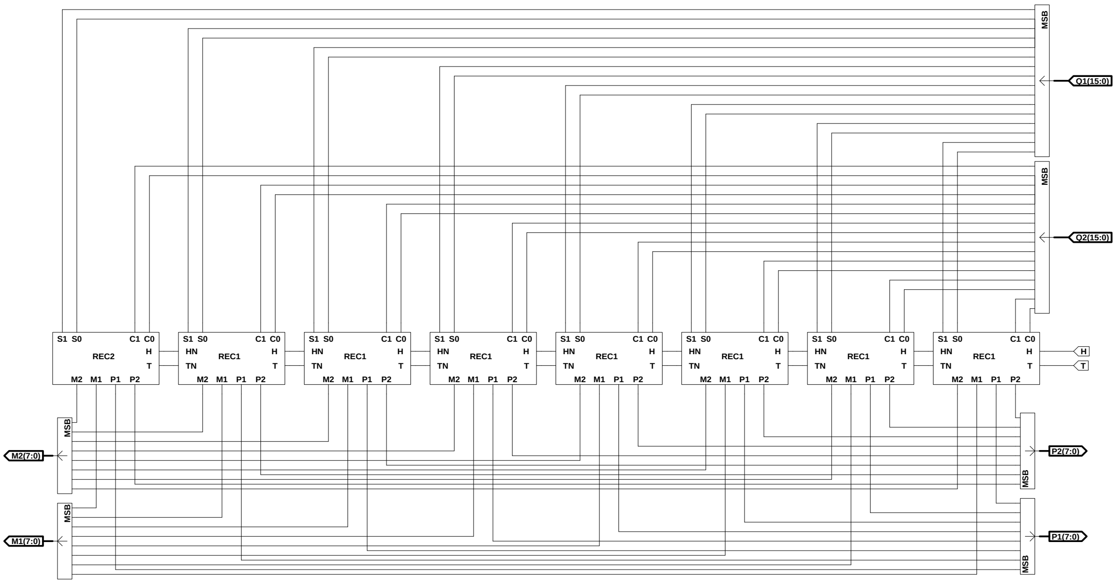

Figure 3.10: Recoder VHDL stages

$$
\begin{array}{lll}
Q[k+1] & \Leftarrow (\ Q[k]\ ,\ q_{k+1}\ ) & q_{k+1} > 0 \\
QM[k+1] & \Leftarrow (\ Q[k]\ ,\ q_{k+1} - 1\ ) & \\
 & & \\
Q[k+1] & \Leftarrow (\ Q[k]\ ,\ q_{k+1}\ ) & q_{k+1} = 0 \\
QM[k+1] & \Leftarrow (\ QM[k]\ ,\ \overline{q_{k+1}}\ ) & \\
 & & \\
Q[k+1] & \Leftarrow (\ QM[k]\ ,\ q_{k+1}\ ) & q_{k+1} < 0 \\
QM[k+1] & \Leftarrow (\ QM[k]\ ,\ \overline{q_{k+1}}\ ) &
\end{array}
$$

where $Q[k+1] \Leftarrow (\ Q[k]\ ,\ q_{k+1}\ )$ means that the register Q at iteration $(k+1)$ is loaded with the concatenation of the previous value Q and the 9 bits of q_{k+1}. $\overline{q_{k+1}}$ is the bit-complement of q_{k+1}.

The on-the-fly conversion can be summarized in the following steps:

1. the carry-save representation of q_j is assimilated and rounded.

$$q_{k+1} = \lfloor qs + qc + 0.5 \rfloor$$

```
qs :  xxxxxxxxxxxx.xx
qc :  xxxxxxxxxxxx.xx
.5 :  000000000000.10
qk :  ---xxxxxxxxx---
```

2. The new value of q_{k+1} to be loaded into the registers is calculated.

3. The two registers Q and QM are loaded and shifted according with the value q_{k+1}.

The rounding is done using the same method, and another register QP is needed. This register is updated every iteration, but its value is used only at the end (rounding step).

$$
\begin{array}{lll}
QP[k+1] & \Leftarrow (\ QP[k]\ ,\ 0\ ) & q_{k+1}+1 = 512 \\
QP[k+1] & \Leftarrow (\ Q[k]\ ,\ q_{k+1}+1\ ) & 0 \leq q_{k+1}+1 < 512 \\
QP[k+1] & \Leftarrow (\ QM[k]\ ,\ q_{k+1}+1\ ) & q_{k+1}+1 < 0
\end{array}
$$

In the last iteration, if the last residual is positive, the quotient q must be incremented by 1.

$$p = q_L + \overline{SIGN}$$

$\overline{SIGN}$ is the signal **SIGN** coming from block **Cpa**, the MSB of the reminder. The final quotient q is then:

$$
\begin{array}{lll}
q & \Leftarrow (\ QP[L-1]\ ,\ p\ ) & p = 512 \\
q & \Leftarrow (\ Q[L-1]\ ,\ p\ ) & 0 \leq p < 512 \\
q & \Leftarrow (\ QM[L-1]\ ,\ p\ ) & p < 0
\end{array}
$$

The block diagram of this converting and rounding circuit is shown in Figure 3.11.

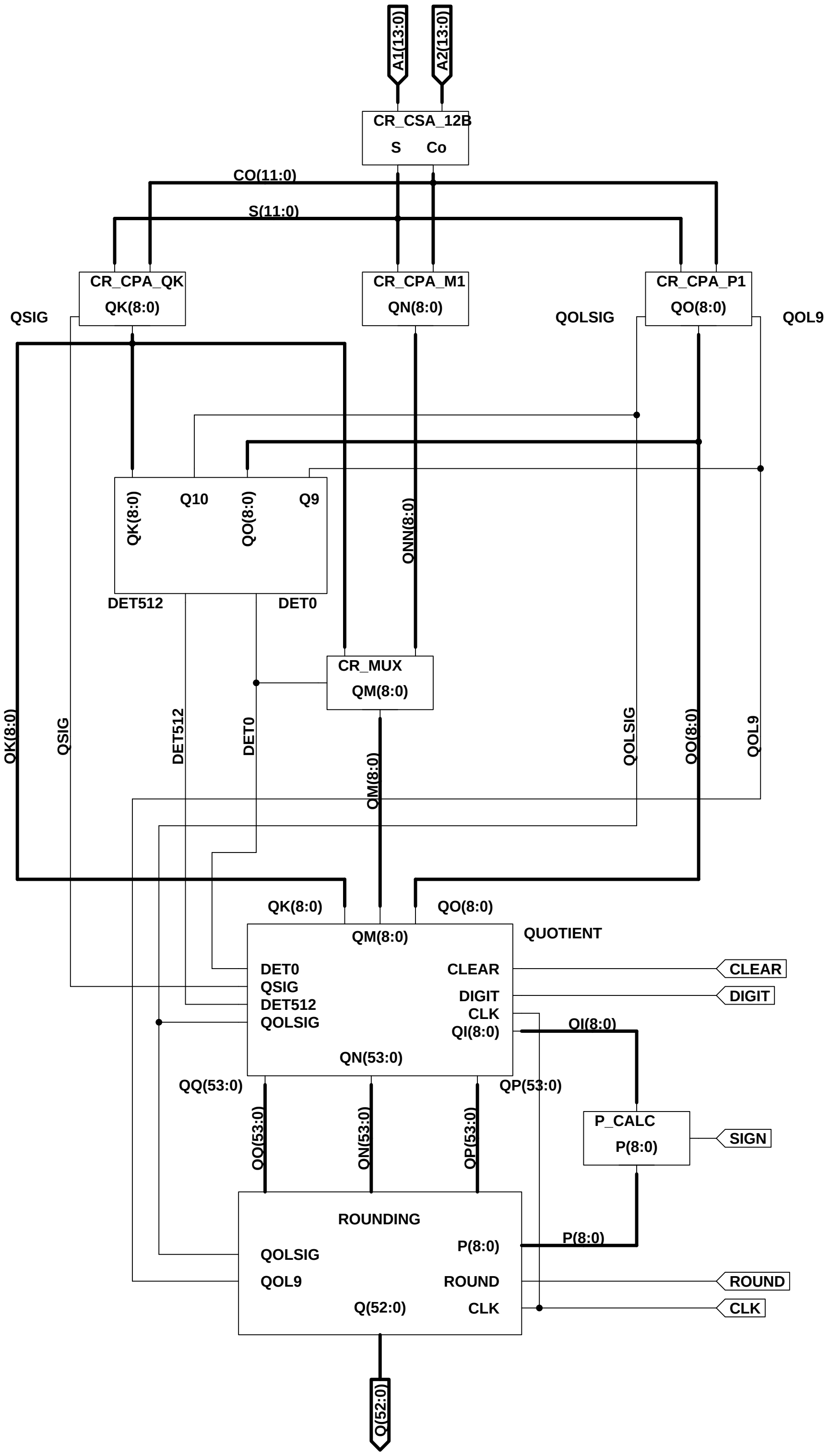

Figure 3.11: Convert block diagram

cr_csa_12b calculates the carry-save representation of the quotient digit. It reduces the addends from 3 to 2.

cr_cpa_qk calculates q_{k+1} and its sign **QSIG**.

cr_cpa_m1 decrements q_{k+1} by 1. It is needed to update QM when $q_{k+1} > 0$

$$q_{k+1} = q_{k+1} - 1$$

cr_cpa_p1 increments q_{k+1} by 1. It is needed to update QP.

$$q_{k+1} = q_{k+1} + 1$$

It also generates the two signals **QOLSIG** and **QOL9**. **QOLSIG** is the sign of $q_{k+1} + 1$, and **QOL9** is the second MSB of $q_{k+1} + 1$. It is used along with **QOLSIG** in the rounding step to determine if $q_{k+1} + 1$ is equal or greater than 512.

cr_det calculates the values of the bits used to select the registers to load and shift Q, QM, QP. The two outputs indicate whether or not $q_{k+1} = 0$ and $q_{k+1} + 1 = 512$.

cr_mux selects the digit to be loaded in QM according with the sign of q_{k+1}

$q_{k+1} > 0$	$q_{k+1} \Leftarrow q_{k+1} - 1$	from **cr_cpa_m1**
$q_{k+1} \leq 0$	$q_{k+1} \Leftarrow \overline{q_{k+1}}$	bit-complement

quotient loads and shift the three registers Q, QN and QP, doing the update of the quotient, every iteration.

p_calc increments p by 1 if the reminder is positive (**SIGN** $= 0$).

$$p = p + 1$$

It is a conditional sum incrementer.

rounding loads p in the least significant digit of q. This step is the final rounding.

3.2.8 Cpa

Cpa is a 70 bits carry-propagate adder that is used to assimilate the sum and carry parts of the result. In particular, it is used to calculate z and to determine the sign of the last residual $w[6]$ necessary for the final rounding. **Cpa** is implementented with a 3-level carry look-ahead scheme [9]. The carries are grouped by 4 and the hierarchy is 4-16-64. The six most significant bits of the result are again generated with a 6 bits carry look-ahead adder (see Figure 3.12).

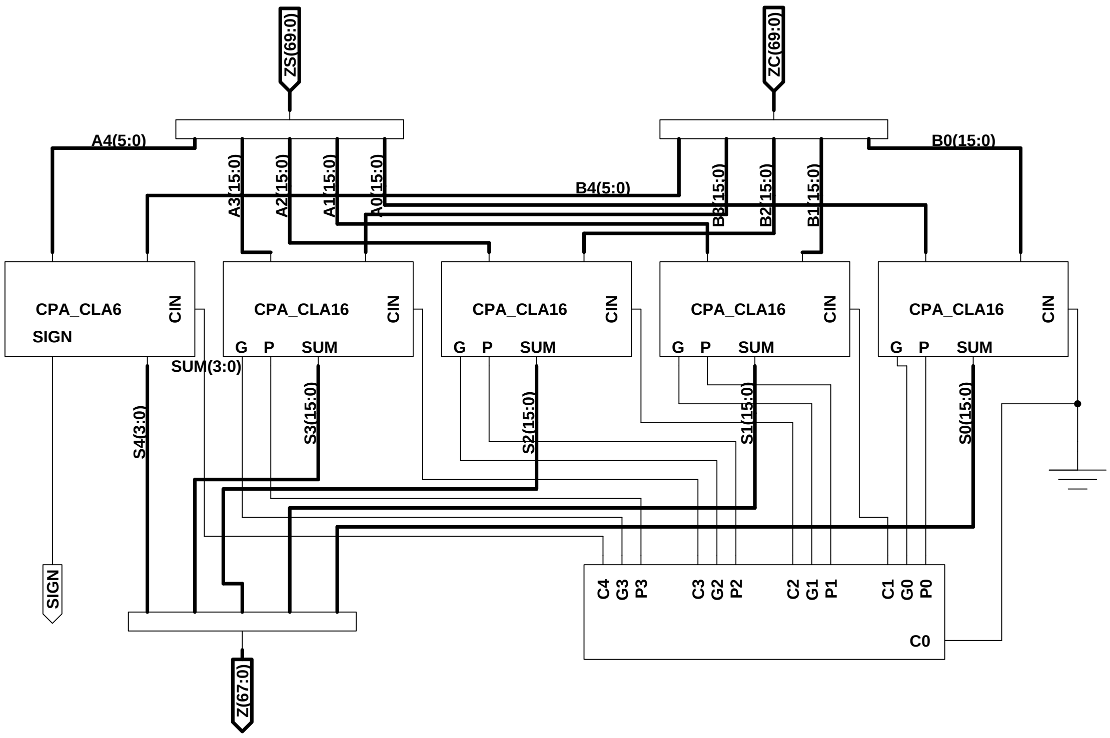

Figure 3.12: Cpa block diagram

Chapter 4
Physical Design

In this chapter we describe the physical design of the divider. From the VHDL description of the blocks we synthesize or design the different parts of the circuit and then we generate the layout.

4.1 Implementation in $1.2\mu m$ Library

This section describes the VLSI implementation using Compass and the standard cells library VTI cmn12 process $1.2\mu m$ [11].

The gate-level descriptions of some blocks, or schematics, were obtained using Compass ASIC Synthesizer [8]. This synthesizer, reading a VHDL file, is able to generate a netlist by choosing cells from the library supported. It can perform synthesis starting from a behavioral description as well as from a more specific description, such as a switching expression. In the design of modules the optimization can take into account several parameters, such as maximum capacitance and cell fan-out, frequency, maximum delay and set-up time, temperature, and VDD level. The default synthesis constraint is to obtain minimum area. An example of a circuit synthesized directly from the behavioral VHDL description is the controller block, and its schematic is shown in Figure 4.1.

Other blocks were manually designed. These blocks, such as buffers, were introduced in the schematic after the first simulations when we noticed that long delays were produced due to the high loading of some cells. An example of a manually entered schematic is the single stage of the recoder shown in Figure 4.2.

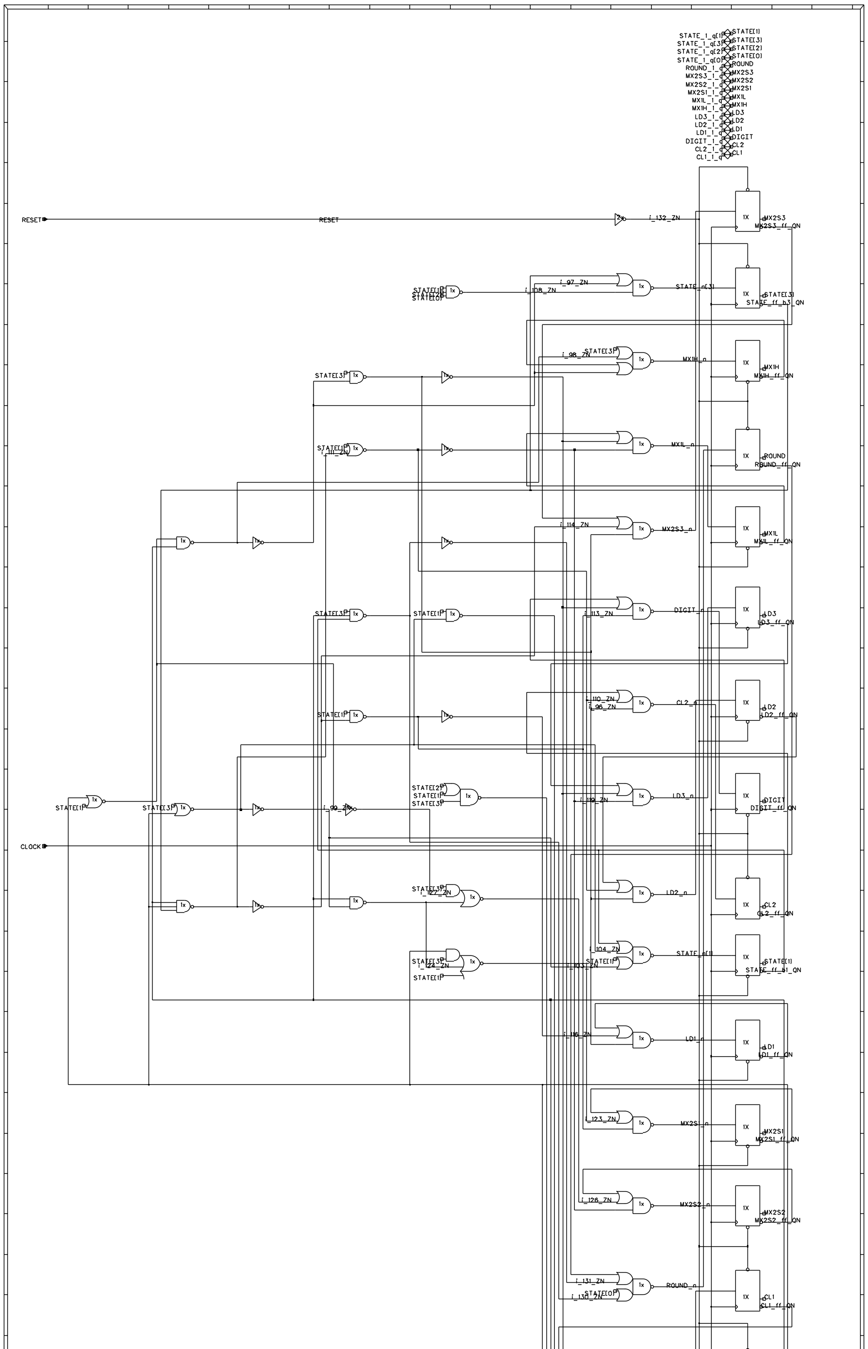

Figure 4.1: Controller Compass schematic

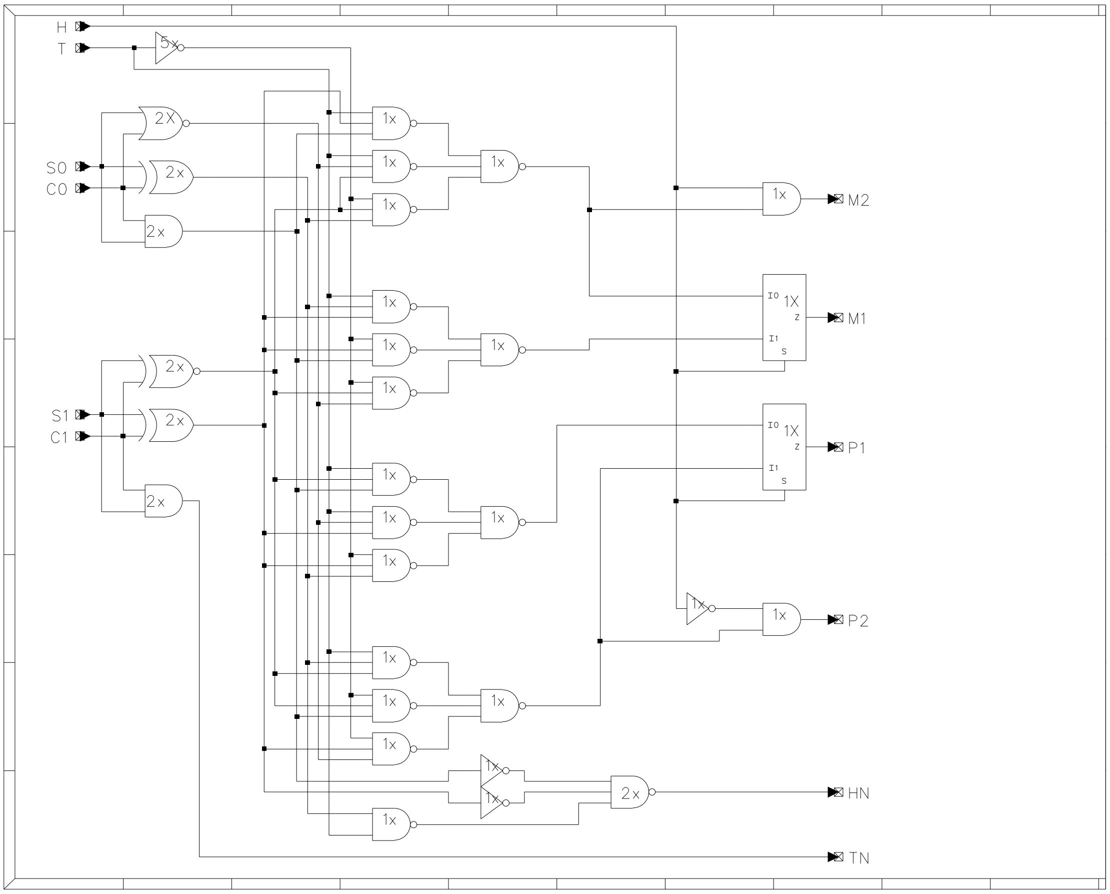

Figure 4.2: Single stage recoder Compass schematic

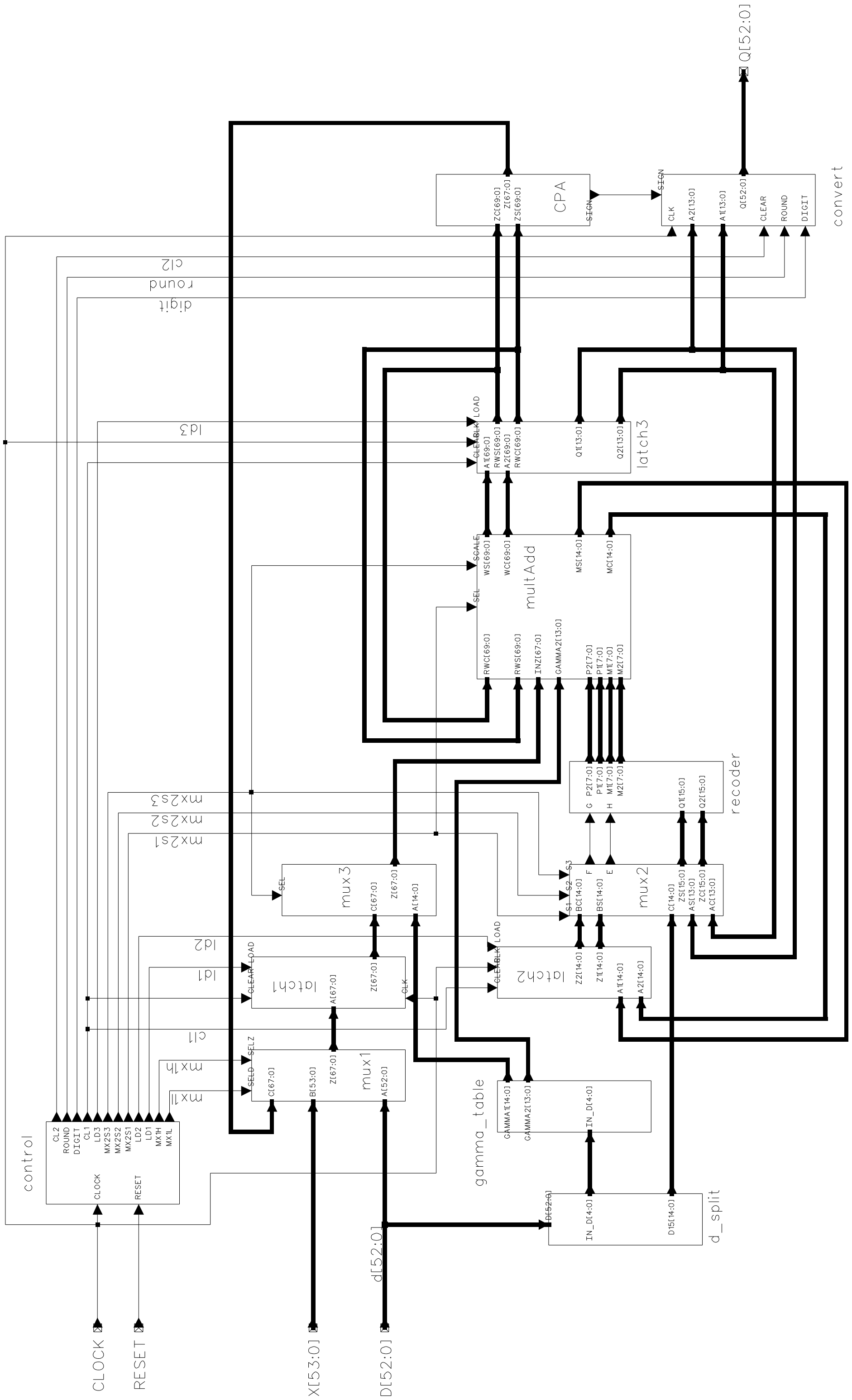

Figure 4.3: Radix-512 divider Compass schematic

All these blocks were then manually assembled into a gate-level block diagram (Figure 4.3) that was given as the input to the Compass Chip Compiler for the layout generation.

4.1.1 Area and Critical Path

To achieve a design with suitable speed and area we first designed each module using the default constraint (*minimum area*) and then optimized those modules in the critical path. We did this iteratively until the minimum achievable delay was obtained. The cells were obtained from the standard cells library VTI cmn12 $1.2\mu m$. The results of the first layout are given in Table 4.1. The dimensions are 6.6 $mm \times 5.4$ mm and the critical path is 86 ns. From the entry *Total cells* in Table 4.1 we can see that the sum of the areas of the blocks is 80 % of the area of the whole divider; the other 20 % corresponds to inter-block routing.

Block	area		no. transistors	
	mm^2	%	$\times 1000$	%
control	0.3	1.0 %	0.7	1.2 %
convert	4.6	16.5 %	11.1	19.7 %
cpa	1.6	6.0 %	3.7	6.5 %
gammaTable	0.6	2.1 %	1.0	1.8 %
latch1	0.7	2.5 %	2.2	3.9 %
latch2	0.3	1.0 %	1.0	1.7 %
latch3	1.7	6.0 %	4.5	8.0 %
multadd	16.7	59.0 %	29.0	50.9 %
mux1	0.6	2.0 %	1.0	1.7 %
mux2	0.2	1.0 %	0.4	0.7 %
mux3	0.2	1.0 %	0.5	0.9 %
recoder	0.7	2.5 %	1.6	2.9 %
Total cells	28.4	100.0 %	56.6	100.0 %
Total divider	35.6			

Table 4.1: Area of first layout

As a second step, we redesigned some blocks to reduce the critical path. Since the typical delay of a nand2 gate of the library used is 1 ns, we calculated from the structure of the circuit a total delay of 36 ns. The difference between this value and the 86 ns of the first design is due to high delays of gates driving high loads.

The second design was made by optimizing the blocks which were found along the critical path. After a block was modified, the critical path changed and sometimes it went through a different block that had been optimized as well. This optimization was done again using the ASIC Synthesizer tool giving a *minimum delay* constraint. After this redesign the critical path was reduced to 39 ns and the area changed to 7.3 $mm \times 5.8$ mm.

The layout is shown in Figure 4.4 and its characteristics are given in Table 4.2. With respect to the first design the area increased by 20% and the number of

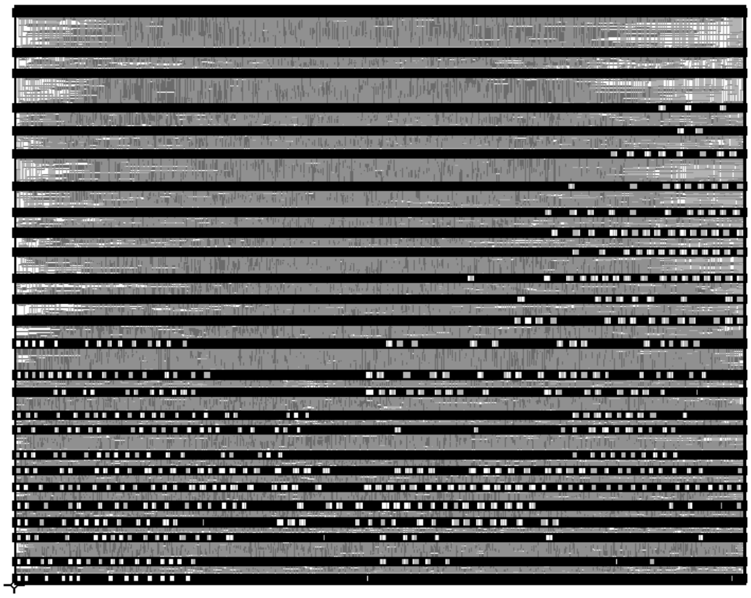

Figure 4.4: Radix-512 divider layout (1.2μm library)

transistors by 23%.

Figure 4.5 gives the delay of each of the blocks in the critical path.

Block	area		no. transistors	
	mm^2	%	$\times 1000$	%
control	0.3	1.0 %	0.9	1.3 %
convert	6.5	18.3 %	14.0	20.0 %
cpa	2.8	8.0 %	6.9	9.8 %
gammaTable	0.6	1.7 %	1.0	1.5 %
latch1	1.1	3.1 %	2.8	3.9 %
latch2	0.5	1.3 %	1.2	1.7 %
latch3	3.0	8.3 %	6.8	9.7 %
multadd	18.8	53.0 %	32.5	46.3 %
mux1	0.6	1.7 %	1.0	1.4 %
mux2	0.4	1.1 %	0.9	1.3 %
mux3	0.2	0.4 %	0.3	0.4 %
recoder	0.8	2.2 %	1.9	2.7 %
TOTAL	35.5	100.0 %	70.1	100.0 %
divider	42.8			

Table 4.2: Area of second layout

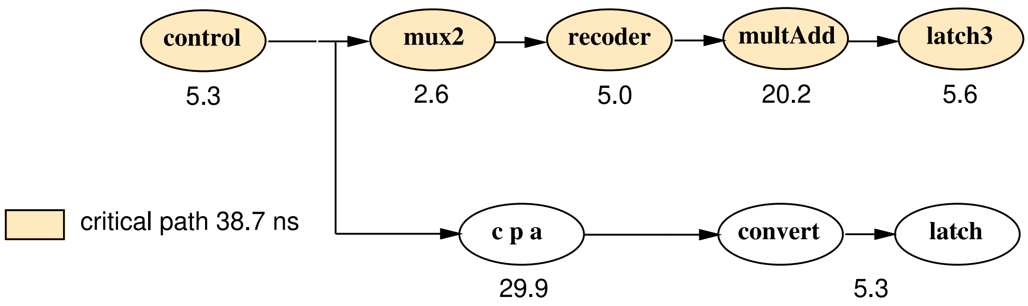

Figure 4.5: Critical path

4.1.2 Simulations

The pre-layout simulations were made choosing a clock cycle of 40 ns that is almost as long as the critical path. These simulations gave correct results (Figure 4.6).

After the layout generation, we extracted the chip netlist and simulated it again to see if the interconnections and the wire capacitance affected the performance of the circuit. While simulating again with a clock cycle of 40 ns we noticed a set-up violation. The result of this violation is an *Unknown* state on some of the outputs

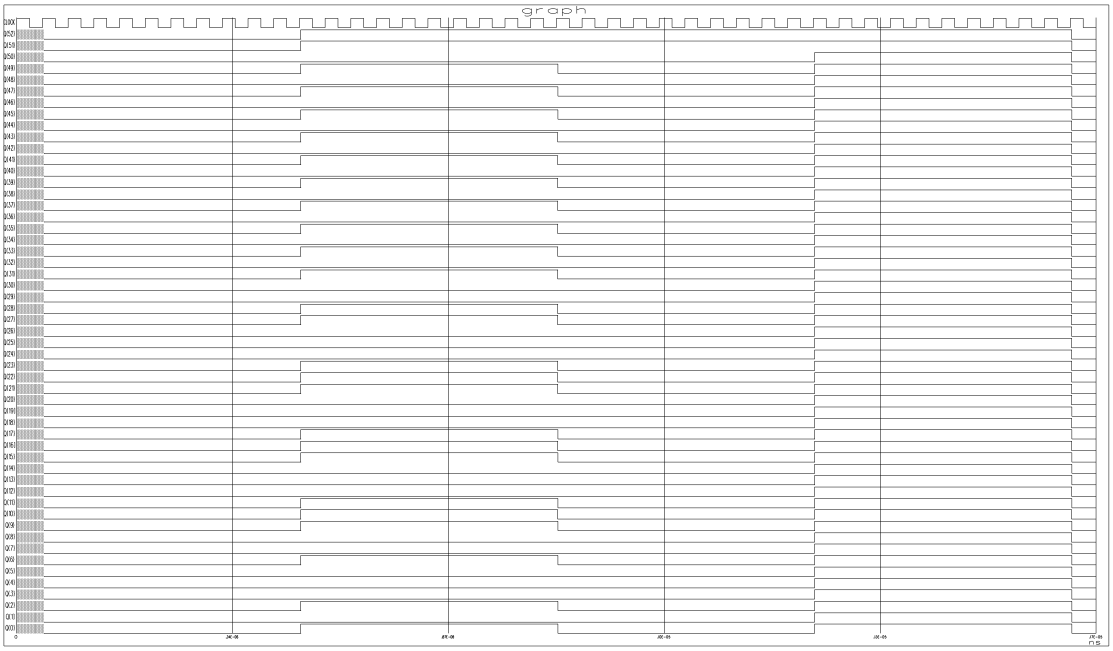

Figure 4.6: Pre-layout simulation with clock cycle 40 ns

of the circuit (Figure 4.7). By lengthening the clock cycle of 1 ns (from 40 ns to 41 ns), we obtained satisfactory results (Figure 4.8).

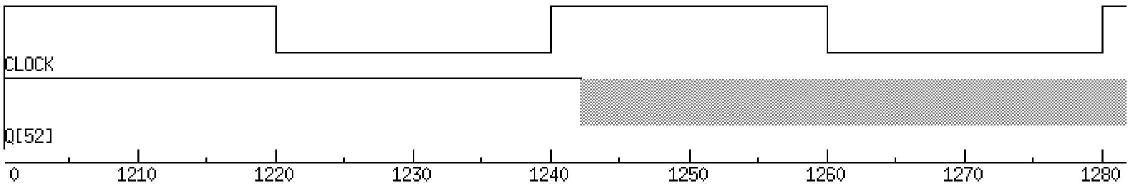

Figure 4.7: Detail of post-layout simulation with clock cycle 40 ns

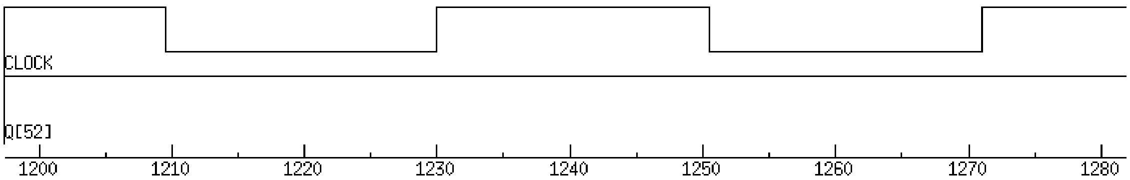

Figure 4.8: Detail of post-layout simulation with clock cycle 41 ns

4.2 Implementation in $0.6\mu m$ Library

This section describes the VLSI implementation using Compass and the standard cells library VTI cmpsc6ul2 process $0.6\mu m$.

4.2.1 Area and Critical Path

We repeated all the steps as in the implementation of the $1.2\mu m$ library. Based on the evaluation of earlier smaller designs, we expected a reduction of the delay of almost 50% and a linear shrinking factor of 0.5 in area.

The critical path of the divider designed with the *minimum area* constraint is 25 ns. By optimizing the design we reduced it to 19 ns.

Figure 4.9 shows that the critical path is not through the main recurrence loop, as in the case of the $1.2\mu m$ design, but that it passes through the carry-propagate adder (**Cpa**) and the convert and round block. Since the delay of the two paths

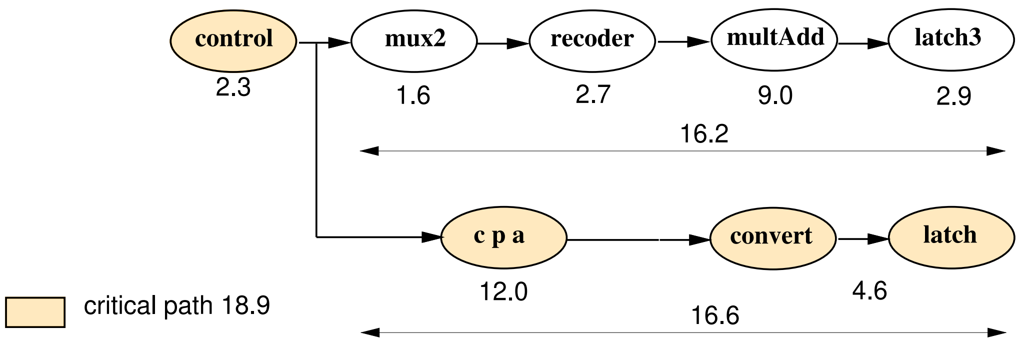

Figure 4.9: Critical path

differs only by 0.4 ns, it is not worth to optimize the **Cpa** block.

The dimensions of the layout are 3.8 $mm \times 3.4$ mm. It is shown in Figure 4.10 and its characteristics are given in Table 4.3.

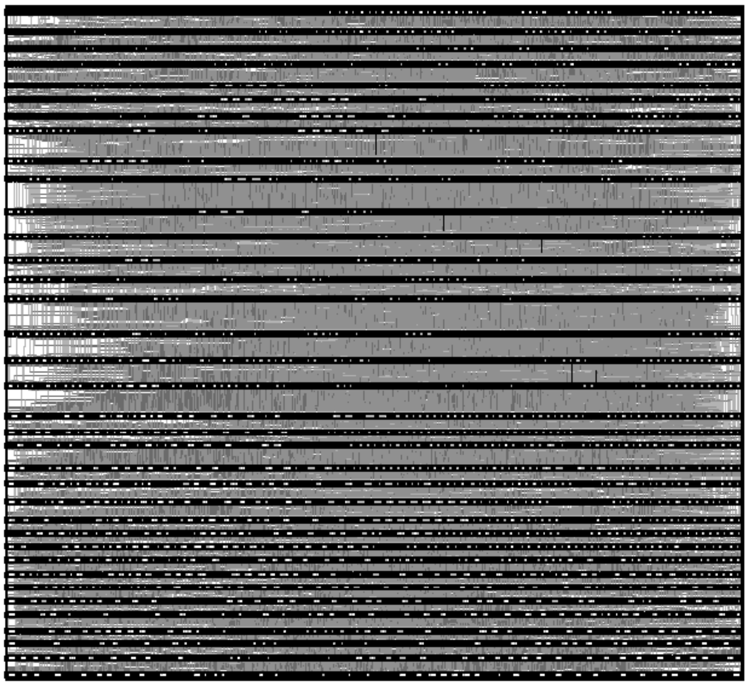

Figure 4.10: Radix-512 divider layout ($0.6\mu m$ library)

4.2.2 Simulations

To simulate the $1.2\mu m$ circuit we used the same methodology as in the $1.2\mu m$ design. In this case the pre-layout simulations were carried out successfully with a clock cycle of 20 ns. But the post-layout simulations produced set-up violations until we lengthened the clock cycle to 24 ns (Figure 4.11). This increase of 20% is explained by the RC effects due to the interconnections. While gate delay decreases linearly with technology, wiring delay does not. In a sub-micron technology the interconnection delay results in a significant contribution to the overall delay that can no longer be overlooked.

Block	area		no. transistors	
	mm^2	%	$\times 1000$	%
control	0.09	1.0 %	0.9	1.1 %
convert	1.86	20.0 %	15.9	19.7 %
cpa	0.56	6.1 %	4.9	6.1 %
gammaTable	0.20	2.2 %	1.6	2.0 %
latch1	0.26	2.8 %	2.7	3.3 %
latch2	0.10	1.1 %	1.2	1.5 %
latch3	1.10	11.8 %	9.9	12.2 %
multadd	4.60	49.4 %	39.2	48.6 %
mux1	0.17	1.8 %	1.3	1.6 %
mux2	0.07	0.8 %	0.6	0.7 %
mux3	0.04	0.4 %	0.3	0.3 %
recoder	0.25	2.7 %	2.3	2.8 %
TOTAL	9.32	100.0 %	80.6	100.0 %
divider	12.99			

Table 4.3: Area of the $0.6\mu m$ tech layout

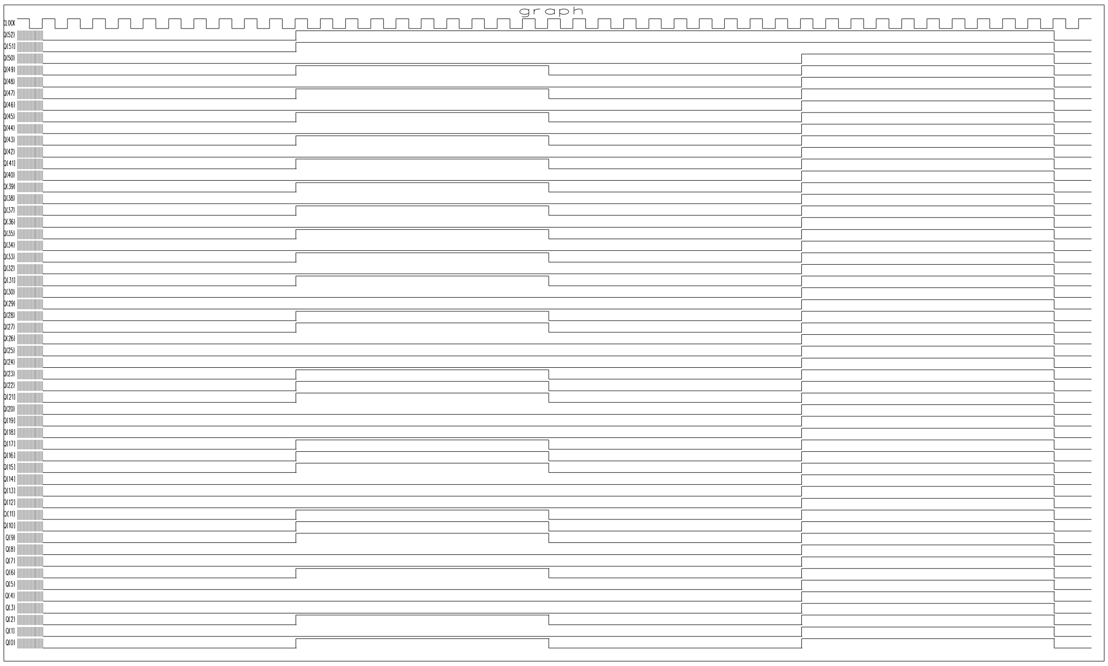

Figure 4.11: Post-layout simulation with clock cycle 24 ns

Chapter 5
Evaluation of the Design

5.1 Comparison with Previous Evaluations

In this section we perform comparisons with the evaluations of delay and area presented in [3] and [4]. As indicated before, the evaluations in [3] are given in terms of the delay and area of a 2-input NAND gate (nand2 units), whereas those of [4] are given in terms of full-adder units. We want to establish the accuracy of these measures, by considering an actual implementation.

We consider the implementation in the $1.2\mu m$ library, whose characteristics are closer to those of the library used in [3], but these evaluations can be extended to any library.

5.1.1 Delay

Table 5.1 reports the delays in terms of nand2 units for the estimations done in [3] and those obtained for the $1.2\mu m$ library implementation. We can observe that the main difference between the two total delays corresponds to the fact that the estimation in [3] did not include the delay of the control block. In addition, there was a slight increase in the delay because of the incorporation of the calculation of M in the main **MultAdd** unit (this corresponds to the increase in delay in the adder block). On the other hand, there are significant variations in the delay of the latch and of the mux2 due to different characteristics of the cell libraries used.

In [4], full-adder units are used instead of nand2 units. Reference [3] established that by making the delay of a full-adder equal to four nand2 units, the results are quite accurate.

Block	eval. [3]	actual
control	N/A	5.3
latch3	8.0	4.1
(set-up)		1.5
mux2	1.4	2.6
recoder	6.0	5.0
mult. gen.	3.6	6.6
adder	12.0	13.6
Total	31.0	38.7

Table 5.1: Comparison of critical paths (nand2 units)

5.1.2 Area

In the library we used, the area of a nand2 gate is $3025\mu m^2$. Dividing the area of each block by that number we obtain the equivalent nand2 representation of area. Table 5.2 shows the area estimated in [3] and that obtained in the actual implementation. The entry *divider(*)* includes the **Cpa** block and the routing.

The latches are faster in our implementation, but their area has doubled. Furthermore, in the evaluation made in [3] neither the controller nor the multiplexers were considered. On the other hand, in our implementation we eliminated the scaling multiplier block, which contributes more than 10% to the overall area.

5.2 Comparison between $1.2\mu m$ and $0.6\mu m$ Implementations

In this section we compare the two implementations of the radix-512 divider unit.

5.2.1 Delay

From the two pictures showing the critical path (Figure 4.5 and Figure 4.9), we can determine that the speed-up attainable by moving from the $1.2\mu m$ to the $0.6\mu m$

Block	eval. [3]	actual
control	N/A	110
convert	1900	2150
gammaTable	560	200
scaling multiplier	1280	N/A
registers(3)	650	-
latch1	-	360
latch2	-	150
latch3	-	980
mult. gen.	3000	-
adder	3100	-
multadd	-	6210
mux1	-	200
mux2	-	130
mux3	N/A	50
recoder	70	260
Total blocks	10500	10800
cpa	N/A	940
divider(*)		14000

Table 5.2: Comparison of area (in nand2 units)

library is about 2.

$$\frac{38.7}{18.9} = 2.04$$

This result was somewhat expected considering that, according to the CMOS-transistor scaling theory [12], if the devices are scaled by a constant α then the gate delay will decrease by a factor $1/\alpha$. But if we consider the clock cycle length obtained after the post-layout simulations, we can see that this speed-up is reduced.

$$\frac{41}{24} = 1.7$$

This is also in accordance with the scaling theory: while the gate delay tends to reduce, the wiring delay remains almost constant. As the level of integration increases, the average wire length on a chip tends to increase also, thereby increasing the capacitance.

5.2.2 Area

Comparing Table 4.2 to Table 4.3, we can conclude that the reduction of area when moving from $1.2\mu m$ to $0.6\mu m$ is

$$\frac{42.80}{12.99} \simeq 3.3 \quad .$$

But if we look at the entry $TOTAL$ on both tables we can see that the reduction is

$$\frac{35.5}{9.32} \simeq 3.8 \quad .$$

As noted earlier, the difference between the entry $TOTAL$ and the entry *divider* is due to the inter-block routing. We can conclude that as the device size decreases the contribution of the routing to the overall area becomes significant.

The number of transistors depends on the characteristics of the cells in the two libraries. For example, for the recoder, which was entered manually by placing the same set of gates, we have 1872 transistors in the $1.2\mu m$ implementation and 2256 in the sub-micron one.

5.3 Comparison between the Radix-512 and the Radix-4 Divider Units

In this section we evaluate the radix-512 divider comparing its performance and area to those of a radix-4 divider.

A detailed description of the radix-4 algorithm can be found in [3] and its implementation in [5].

Table 5.3 summarizes the results obtained in [5] for the radix-4 divider and those for the radix-512 divider presented here. Both units are implemented with the same library: standard cells library VTI cmn12 [11] process $1.2\mu m$.

The difference in the number of clock cycles depends on the radix. In the radix-4

unit	radix-4	radix-512
no. transistors	26500	70100
area	11.40 mm^2	42.75 mm^2
no. of clock cycles	29	10
clock cycle	30 ns	41 ns
total elapsed time	870 ns	410 ns

Table 5.3: Radix-4 and radix-512 dividers - 1.2μm summary

algorithm we obtain 2 bits of the result every iteration, while in the radix-512 one we obtain 9 bits.

According to Table 5.3, the speed-up of the radix-512 over the radix-4 unit is

$$\frac{870}{410} = 2.12 \quad .$$

The increase in the area is

$$\frac{42.75}{11.40} = 3.75 \quad .$$

By comparing these results with those presented in [3]

$$\text{speed-up} = 2.26$$

$$\text{area increase} = 3.1$$

we notice an error of 7% in speed-up and an error of 17% in area increase. However comparing the number of transistors, we see that this ratio is less than 3. Again, the routing between cells plays an important role as the number of the transistors (cells) increases.

Furthermore, if we consider a version of the radix-512 divider, where the scaling factor M of the next division is calculated during the rounding, as mentioned in Chapter 2, the speed-up over the radix-4 divider is

$$\frac{870}{41 \cdot 9} = 2.36 \quad .$$

unit	radix-4	radix-512
no. transistors	27332	80610
area	3.5 mm^2	13.0 mm^2
no. of clock cycles	29	10
clock cycle	18 ns	24 ns
total elapsed time	522 ns	240 ns

Table 5.4: Radix-4 and radix-512 dividers - 0.6μm summary

Table 5.4 summarizes the results obtained for the implementation in the standard cells library VTI cmpsc6ul2 process 0.6μm.

In this case the speed-up of the radix-512 over the radix-4 unit is

$$\frac{522}{240} = 2.18 \quad .$$

The increase in the area is

$$\frac{13.0}{3.5} = 3.71 \quad .$$

By comparing these results with those presented in [3] we obtain an error of 4% in speed-up and an error of 16% in area increase.

Chapter 6
Conclusions

The goal of this work is to implement the design of a complex arithmetic module, a radix-512 divider, to see if such a scheme can be effective in the realization of fast arithmetic units.

We have implemented the radix-512 divider and evaluated its delay and area. We then compared these values with those obtained in [3] and [4], which were done without an actual implementations. We also evaluated the impact on delay and area moving from the $1.2\mu m$ to the $0.6\mu m$ standard cells library. And finally, we compared the radix-512 divider with a radix-4 divider.

We presented the design of the radix-512 divider unit in two different libraries. The results can be summarized as follows:

$1.2\mu m$ lib.	area = $7.3\ mm \times 5.8\ mm$	total division time = 410 ns
$0.6\mu m$ lib.	area = $3.8\ mm \times 3.4\ mm$	total division time = 240 ns

The radix-512 divider is very fast when compared to units which have traditionally been used. We have shown that the speed-up over the radix-4 divider is more than 2. It can be very effective if the priority is to have a fast circuit and the area is of lesser importance.

We conclude that the evaluation methods presented in [3] and [4] are reasonable when used to compare between different schemes of dividers to determine speed-up and area increase. But those methods are not very accurate in estimating the area and the delay of the real circuit especially when the implementation of the design moves toward sub-micron technologies.

Finally, the design flow proposed in this work can be very efficient in the development of arithmetic modules, essentially to achieve a more accurate evaluation of new schemes and algorithms.

In conclusion, the VHDL language and the synthesis tools make possible a very fast implementation when the RTL-model of the unit has been verified and validated. This methodology is very rapid when transferring a design from one library to another.

Bibliography

[1] A. Svoboda. An algorithm for division. *Information Processing Machines*, 1963.

[2] J. Klir. A note on svoboda's algorithm for division. *Information Processing Machines*, 1963.

[3] M.D. Ergegovac and T. Lang. *Division and Square Root: Digit-Recurrence Algorithms and Implementations.* Kluwer Academic Publisher, 1st edition, 1994.

[4] M.D. Ergegovac, T. Lang, and P. Montuschi. Very-high radix division with prescaling and selection by rounding. *IEEE Transactions on Computers*, pages 909–918, August 1994.

[5] A. Nannarelli. Implementation of a radix-4 divider. *Technical Report*, May 1995.

[6] Douglas L. Perry. *VHDL.* McGraw-Hill, Inc., 2nd edition, 1994.

[7] Synopsys. *Synopsys User's Manual.* Synopsys Inc., 1992.

[8] Compass Design Automation. *User Manuals for COMPASS VLSI.* Compass Design Automation, Inc., 1992.

[9] Israel Koren. *Computer Arithmetic Algorithms.* Prentice-Hall, Inc. , 1st edition, 1993.

[10] J. D. Bruguera and T. Lang. Implementation of the fft butterfly with redundant arithmetic. *Technical Report*, August 1994.

[11] VLSI Technology. *1.2-Micron CMOS Portable Library.* VLSI Technology Inc., 1990.

[12] N. H. E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design.* Addison-Wesley Publishing Company, 2nd edition, 1993.

Appendix A
VHDL Descriptions

In this appendix we describe the VHDL implementation of the behavioral and RTL-model. For further details, we suggest the reader look at the following World Wide Web (WWW) URL:

http://www.eng.uci.edu/~alberto/MSthesis/

A.1 Behavioral Model

The VHDL behavioral model is split in two files:

- **radix512.vhdl** is the main block. It contains the recurrence loop and the scaling operations.
- **pack.vhdl** contains all the functions and procedures used in **radix512.vhdl** .

A.1.1 radix512.vhdl

```
use work.DIVIDER.all;

entity radix512 is
  generic( simulation_delay : time := 10 ns );
  port( x: in bit_vector (53 downto 0);
        d: in bit_vector (52 downto 0);
        q: out bit_vector (52 downto 0));
end radix512;

architecture radix512 of radix512 is

begin
```

```
process
  variable    one, zero, onehalf : bit_vector (69 downto 0);
                       -- constants 0,1, 0.5
  variable    ws, wc   : bit_vector (69 downto 0);
                       -- residual (carry-save format)
  variable    rws, rwc : bit_vector (69 downto 0);
                       -- shifted residual (carry-save format)
  variable    di : bit_vector (14 downto 0); -- 15 MSBs of d
  variable    m  : bit_vector (13 downto 0); -- M
  variable    z  : bit_vector (69 downto 0); -- z = Md
  variable    qu : bit_vector (69 downto 0); -- quotient (extended)
  variable    qi : bit_vector (69 downto 0); -- quotient-digit (extended)
  variable    qz : bit_vector (69 downto 0); -- q*z
  variable    zp : bit_vector (66 downto 0); -- temporary
  variable    zu : bit_vector (67 downto 0); -- temporary
  variable    wr : bit_vector (69 downto 0); -- final residual

  begin

      -- Initalization
      zero := conv_bit_vector(0, 70);
      one  := conv_bit_vector(1, 70);
      onehalf := left_shift(one, 70-13);

      wait on d,x;

      qu := zero;
      q  <= put_result(qu);

      -- 15 MSBs of d are taken to choose the scaling factor M
      for i in 0 to 14 loop
           di(i) := d(38+i) ;
      end loop;
      -- M is calculated
      m  := find_m(di);

      -- Scaling of d and x, Mx = w is stored in carry-save format
      zp  := m*d;
      for i in 0 to 66 loop
           z(i+1) := zp(i) ;
      end loop;
      z(69) := '0'; z(68) := '0'; z(0) := '0';
      zu := m*x;
```

```
        for i in 0 to 67 loop
             rws(i) := zu(i) ;
        end loop;
        rws(69) := '0'; rws(68) := '0';
        rwc := zero;

        -- 6 iterations
        for i in 1 to 6 loop

             -- 9 bits of the result are calculated every iteration
             qi := right_shift(( rws + rwc + onehalf ),70-(9+3),i) ;
             qu := qu + left_shift(qi,(6-i)*9);

             -- the new residue is CS-format is calculated
             qz := fit(two_complement(qi * z)) ;
             ws := csa_sum(rws,rwc,qz) ;
             wc := csa_carry(rws,rwc,qz) ;

             wait for simulation_delay;

             rws := left_shift(ws,9);
             rwc := left_shift(wc,9);

        end loop;

        -- final rounding
        -- if the residue is positive add 1 to the result
        wr := ws + wc ;
        if ( wr(69) = '0' ) then
                qu := qu + 1 ;
        end if;

        q  <= put_result(qu);

  end process;

end radix512;
```

A.1.2 pack.vhdl

```
PACKAGE divider IS

  function csa_sum(a,b,c : bit_vector (69 downto 0))
           return bit_vector ;

  function csa_carry(a,b,c : bit_vector (69 downto 0))
           return bit_vector ;

  function left_shift(a : bit_vector (69 downto 0); n : integer)
           return bit_vector ;

  function right_shift(a : bit_vector (69 downto 0); n : integer;
                        m : integer)
           return bit_vector ;

  function two_complement(a : bit_vector ((69*2+1) downto 0))
           return bit_vector ;

  function fit(a : bit_vector ((69*2+1) downto 0))
           return bit_vector ;

  function find_m(a : bit_vector (14 downto 0))
           return bit_vector ;

  function put_result(a : bit_vector (69 downto 0))
           return bit_vector ;

END divider;

PACKAGE BODY divider IS
------------------------------------------------------------------------
  -- sum calculated in Carry-Save form
  function csa_sum(a,b,c : bit_vector (69 downto 0))
   return bit_vector is
   variable s: bit_vector (69 downto 0);
  begin
        for i in 0 to (69) loop
           s(i) := a(i) XOR b(i) XOR c(i) ;
        end loop;
        return s;
  end csa_sum;
------------------------------------------------------------------------
```

```
-- carry calculated in Carry-Save form
function csa_carry(a,b,c : bit_vector (69 downto 0))
 return bit_vector is
 variable s: bit_vector (69 downto 0);
begin
      s(0):= '0' ;
      for i in 0 to (69-1) loop
         s(i+1) := ( a(i) AND b(i) ) OR ( a(i) AND c(i) )
                                     OR ( c(i) AND b(i) ) ;
      end loop;
      return s;
end csa_carry;
------------------------------------------------------------------------
-- left shift
function left_shift(a : bit_vector (69 downto 0); n : integer)
 return bit_vector is
 variable s: bit_vector (69 downto 0);
begin
      for i in 0 to (69-n) loop
          s(69-i) := a(69-n-i);
      end loop;
      for i in 0 to (n-1) loop
          s(i) := '0';
      end loop;
      return s;
end left_shift;
------------------------------------------------------------------------
-- right shift (if m=1 logical shift, otherwise arithmetic shift)
function right_shift(a : bit_vector (69 downto 0); n : integer;
                     m : integer)
 return bit_vector is
 variable s: bit_vector (69 downto 0);
begin
      for i in n to 69 loop
          s(i-n) := a(i);
      end loop;
      for i in (69-n+1) to 69 loop
        if ( m = 1 ) then
          s(i) := '0';
        else
          s(i) := a(69);
        end if;
      end loop;
```

```
      return s;
  end right_shift;
----------------------------------------------------------------------
  -- 2's complement (double word)
  function two_complement(a : bit_vector ((69*2+1) downto 0))
   return bit_vector is
   variable s: bit_vector ((69*2+1) downto 0);
  begin
        for i in 0 to (69*2+1) loop
            s(i) := NOT a(i);
        end loop;
        s := s + 1;
        return s;
  end two_complement;
----------------------------------------------------------------------
  -- fit the result of a multiplication in a shortest variable
  function fit(a : bit_vector ((69*2+1) downto 0))
   return bit_vector is
   variable s: bit_vector (69 downto 0);
  begin
        for i in 0 to 69 loop
            s(i) := a(i+9);
        end loop;
        return s;
  end fit;
----------------------------------------------------------------------
  -- calculates the scaling factor M from the divider d
  function find_m(a : bit_vector (14 downto 0))
   return bit_vector is
   variable s: bit_vector (13 downto 0);
   variable key: bit_vector (4 downto 0);
   variable gamma1, gamma2, agamma1, p : bit_vector (14 downto 0);
   variable double_m : bit_vector ((14*2+1) downto 0);
  begin

    for i in 0 to 4 loop
         key(i) := a(i+9);
    end loop;

    case key is

--       d(i) = 00       gamma1 = 3.878329     gamma2 = 3.938928
         when "00000" => gamma1 := "111110000011011" ;
```

```
                  gamma2 := "111111000001011" ;

--      d(i) = 01       gamma1 = 3.650217     gamma2 = 3.821321
        when "00001" => gamma1 := "111010011001110" ;
                        gamma2 := "111101001001000" ;

--      d(i) = 02       gamma1 = 3.441655     gamma2 = 3.710535
        when "00010" => gamma1 := "110111000100010" ;
                        gamma2 := "111011010111100" ;

--      d(i) = 03       gamma1 = 3.250471     gamma2 = 3.605991
        when "00011" => gamma1 := "110100000000011" ;
                        gamma2 := "111001101100100" ;

--      d(i) = 04       gamma1 = 3.074787     gamma2 = 3.507178
        when "00100" => gamma1 := "110001001100100" ;
                        gamma2 := "111000000111010" ;

--      d(i) = 05       gamma1 = 2.912970     gamma2 = 3.413637
        when "00101" => gamma1 := "101110100110111" ;
                        gamma2 := "110110100111100" ;

--      d(i) = 06       gamma1 = 2.763600     gamma2 = 3.324956
        when "00110" => gamma1 := "101100001101111" ;
                        gamma2 := "110101001100110" ;

--      d(i) = 07       gamma1 = 2.625431     gamma2 = 3.240766
        when "00111" => gamma1 := "101010000000011" ;
                        gamma2 := "110011110110100" ;

--      d(i) = 08       gamma1 = 2.497371     gamma2 = 3.160735
        when "01000" => gamma1 := "100111111101010" ;
                        gamma2 := "110010100100100" ;

--      d(i) = 09       gamma1 = 2.378457     gamma2 = 3.084561
        when "01001" => gamma1 := "100110000011100" ;
                        gamma2 := "110001010110100" ;

--      d(i) = 10       gamma1 = 2.267839     gamma2 = 3.011973
        when "01010" => gamma1 := "100100010010010" ;
                        gamma2 := "110000001100010" ;

--      d(i) = 11       gamma1 = 2.164762     gamma2 = 2.942723
```

```
        when "01011" => gamma1 := "100010101000101" ;
                        gamma2 := "101111000101010" ;

--      d(i) = 12       gamma1 = 2.068556     gamma2 = 2.876586
        when "01100" => gamma1 := "100001000110001" ;
                        gamma2 := "101110000001100" ;

--      d(i) = 13       gamma1 = 1.978624     gamma2 = 2.813357
        when "01101" => gamma1 := "011111101010000" ;
                        gamma2 := "101101000000111" ;

--      d(i) = 14       gamma1 = 1.894433     gamma2 = 2.752847
        when "01110" => gamma1 := "011110010011111" ;
                        gamma2 := "101100000010111" ;

--      d(i) = 15       gamma1 = 1.815502     gamma2 = 2.694886
        when "01111" => gamma1 := "011101000011000" ;
                        gamma2 := "101011000111100" ;

--      d(i) = 16       gamma1 = 1.741404     gamma2 = 2.639316
        when "10000" => gamma1 := "011011110111001" ;
                        gamma2 := "101010001110101" ;

--      d(i) = 17       gamma1 = 1.671751     gamma2 = 2.585991
        when "10001" => gamma1 := "011010101111110" ;
                        gamma2 := "101001011000000" ;

--      d(i) = 18       gamma1 = 1.606196     gamma2 = 2.534778
        when "10010" => gamma1 := "011001101100101" ;
                        gamma2 := "101000100011100" ;

--      d(i) = 19       gamma1 = 1.544422     gamma2 = 2.485554
        when "10011" => gamma1 := "011000101101011" ;
                        gamma2 := "100111110001001" ;

--      d(i) = 20       gamma1 = 1.486144     gamma2 = 2.438206
        when "10100" => gamma1 := "010111110001110" ;
                        gamma2 := "100111000000101" ;

--      d(i) = 21       gamma1 = 1.431105     gamma2 = 2.392628
        when "10101" => gamma1 := "010110111001011" ;
                        gamma2 := "100110010010000" ;
```

```
--      d(i) = 22       gamma1 = 1.379067     gamma2 = 2.348723
        when "10110" => gamma1 := "010110000100001" ;
                        gamma2 := "100101100101000" ;

--      d(i) = 23       gamma1 = 1.329816     gamma2 = 2.306400
        when "10111" => gamma1 := "010101010001101" ;
                        gamma2 := "100100111001110" ;

--      d(i) = 24       gamma1 = 1.283158     gamma2 = 2.265575
        when "11000" => gamma1 := "010100100001111" ;
                        gamma2 := "100100001111111" ;

--      d(i) = 25       gamma1 = 1.238913     gamma2 = 2.226171
        when "11001" => gamma1 := "010011110100101" ;
                        gamma2 := "100011100111100" ;

--      d(i) = 26       gamma1 = 1.196917     gamma2 = 2.188114
        when "11010" => gamma1 := "010011001001101" ;
                        gamma2 := "100011000000101" ;

--      d(i) = 27       gamma1 = 1.157021     gamma2 = 2.151336
        when "11011" => gamma1 := "010010100000110" ;
                        gamma2 := "100010011010111" ;

--      d(i) = 28       gamma1 = 1.119087     gamma2 = 2.115775
        when "11100" => gamma1 := "010001111001111" ;
                        gamma2 := "100001110110100" ;

--      d(i) = 29       gamma1 = 1.082989     gamma2 = 2.081370
        when "11101" => gamma1 := "010001010100111" ;
                        gamma2 := "100001010011010" ;

--      d(i) = 30       gamma1 = 1.048610     gamma2 = 2.048066
        when "11110" => gamma1 := "010000110001110" ;
                        gamma2 := "100000110001001" ;

--      d(i) = 31       gamma1 = 1.015842     gamma2 = 2.015811
        when "11111" => gamma1 := "010000010000001" ;
                        gamma2 := "100000010000001" ;

        when others => null;
    end case;
```

```
    double_m := gamma1*a ;

    for i in 0 to 14 loop
        agamma1(i) := double_m(i+15);
    end loop;

    p := gamma2 - agamma1 ;
    for i in 0 to 13 loop
        s(i) := p(i);
    end loop;

    return s;
  end find_m;
-------------------------------------------------------------------------
  -- load the value of the final result to the output
  function put_result(a : bit_vector (69 downto 0))
   return bit_vector is
   variable s: bit_vector (52 downto 0);
  begin
        for i in 0 to 52 loop
            s(i) := a(i+1);
        end loop;
        return s;
  end put_result;
-------------------------------------------------------------------------

END divider;
```

A.2 RTL Model

The block diagram of the VHDL rtl-model is shown in Figure 3.1. There are 13 blocks in the schematic. **d_split** is simply used to split a bus into two buses of different widths. Four of these blocks are exploded in sub-blocks:

- **Recoder** is composed of 8 stages (Figure 3.10) of the same VHDL description **rec1**.
- **MultAdd** is decomposed as in Figure 3.4. The basic unit of the multiplier is given in **mp_pp2**, while the basic component of the tree of adders (Figure 3.6), an array of full-adders without ripple-carry, is given in **csa_70b**.
- **Convert** is as in Figure 3.11. The main block is the **quotient** block which updates the three registers QP,QQ,QN by shifting or loading them.
- **Cpa** is a carry look-ahead adder. It is composed by the two sub-blocks **cpa_cla16** a 16-bit carry look-ahead adder, and the **cpa_gen**, the group carry generator.

The rest still maintains a VHDL behavioral description. We present here one sample for a latch and for a multiplexer and the VHDL description of the blocks **control** and **gamma_Table**.

A.2.1 radix512

```
entity RADIX512 is
      Port (   CLOCK : In    bit;
                   D : In    bit_vector (52 downto 0);
               RESET : In    bit;
                   X : In    bit_vector (53 downto 0);
                   Q : Out   bit_vector (52 downto 0) );
end RADIX512;
```

```
architecture SCHEMATIC of RADIX512 is

   signal      RWC : bit_vector(69 downto 0);
   signal       M2 : bit_vector(7 downto 0);
   signal       M1 : bit_vector(7 downto 0);
   signal      RWS : bit_vector(69 downto 0);
   signal       P1 : bit_vector(7 downto 0);
   signal       P2 : bit_vector(7 downto 0);
   signal      D15 : bit_vector(14 downto 0);
   signal       D5 : bit_vector(4 downto 0);
   signal    MU1L1 : bit_vector(67 downto 0);
   signal    L1MU3 : bit_vector(67 downto 0);
   signal    MU3MA : bit_vector(67 downto 0);
   signal       MS : bit_vector(14 downto 0);
   signal       MC : bit_vector(14 downto 0);
   signal      MAS : bit_vector(14 downto 0);
   signal      MAC : bit_vector(14 downto 0);
   signal       QS : bit_vector(13 downto 0);
   signal       QC : bit_vector(13 downto 0);
   signal    MU2RS : bit_vector(15 downto 0);
   signal    MU2RC : bit_vector(15 downto 0);
   signal        Z : bit_vector(67 downto 0);
   signal       WS : bit_vector(69 downto 0);
   signal       WC : bit_vector(69 downto 0);
   signal  MGAMMA1 : bit_vector(14 downto 0);
   signal  MGAMMA2 : bit_vector(13 downto 0);
   signal    MX2S3 : bit;
   signal    MX2S2 : bit;
   signal    MX2S1 : bit;
   signal        F : bit;
   signal        E : bit;
   signal     SIGN : bit;
   signal    ROUND : bit;
   signal      CL2 : bit;
   signal    DIGIT : bit;
   signal      LD3 : bit;
   signal      LD1 : bit;
   signal      CL1 : bit;
   signal     MX1H : bit;
   signal     MX1L : bit;
   signal      LD2 : bit;

   component MUX3
```

```
   Port (       A : In    bit_vector (14 downto 0);
                C : In    bit_vector (67 downto 0);
              SEL : In    bit;
                Z : Out   bit_vector (67 downto 0) );
end component;

component D_SPLIT
   Port (       D : In    bit_vector (52 downto 0);
              D15 : Out   bit_vector (14 downto 0);
             IN_D : Out   bit_vector (4 downto 0) );
end component;

component CONTROL
   Port (   CLOCK : In    bit;
            RESET : In    bit;
              CL1 : Out   bit;
              CL2 : Out   bit;
            DIGIT : Out   bit;
              LD1 : Out   bit;
              LD2 : Out   bit;
              LD3 : Out   bit;
             MX1H : Out   bit;
             MX1L : Out   bit;
            MX2S1 : Out   bit;
            MX2S2 : Out   bit;
            MX2S3 : Out   bit;
            ROUND : Out   bit );
end component;

component CONVERT
   Port (      A1 : In    bit_vector (13 downto 0);
               A2 : In    bit_vector (13 downto 0);
            CLEAR : In    bit;
              CLK : In    bit;
            DIGIT : In    bit;
            ROUND : In    bit;
             SIGN : In    bit;
                Q : Out   bit_vector (52 downto 0) );
end component;

component LATCH1
   Port (       A : In    bit_vector (67 downto 0);
            CLEAR : In    bit;
```

```
             CLK : In    bit;
            LOAD : In    bit;
               Z : Out   bit_vector (67 downto 0) );
end component;

component MUX1
   Port (       A : In    bit_vector (52 downto 0);
                B : In    bit_vector (53 downto 0);
                C : In    bit_vector (67 downto 0);
             SELD : In    bit;
             SELZ : In    bit;
                Z : Out   bit_vector (67 downto 0) );
end component;

component CPA
   Port (       ZC : In    bit_vector (69 downto 0);
                ZS : In    bit_vector (69 downto 0);
              SIGN : Out   bit;
                 Z : Out   bit_vector (67 downto 0) );
end component;

component LATCH3
   Port (       A1 : In    bit_vector (69 downto 0);
                A2 : In    bit_vector (69 downto 0);
             CLEAR : In    bit;
               CLK : In    bit;
              LOAD : In    bit;
                Q1 : Out   bit_vector (13 downto 0);
                Q2 : Out   bit_vector (13 downto 0);
               RWC : Out   bit_vector (69 downto 0);
               RWS : Out   bit_vector (69 downto 0) );
end component;

component RECODER
   Port (        G : In    bit;
                 H : In    bit;
                Q1 : In    bit_vector (15 downto 0);
                Q2 : In    bit_vector (15 downto 0);
                M1 : Out   bit_vector (7 downto 0);
                M2 : Out   bit_vector (7 downto 0);
                P1 : Out   bit_vector (7 downto 0);
                P2 : Out   bit_vector (7 downto 0) );
end component;
```

```
component MULTADD
   Port (  GAMMA2 : In    bit_vector (13 downto 0);
              INZ : In    bit_vector (67 downto 0);
               M1 : In    bit_vector (7 downto 0);
               M2 : In    bit_vector (7 downto 0);
               P1 : In    bit_vector (7 downto 0);
               P2 : In    bit_vector (7 downto 0);
              RWC : In    bit_vector (69 downto 0);
              RWS : In    bit_vector (69 downto 0);
            SCALE : In    bit;
              SEL : In    bit;
               MC : Out   bit_vector (14 downto 0);
               MS : Out   bit_vector (14 downto 0);
               WC : Out   bit_vector (69 downto 0);
               WS : Out   bit_vector (69 downto 0) );
end component;

component MUX2
   Port (      AC : In    bit_vector (13 downto 0);
               AS : In    bit_vector (13 downto 0);
               BC : In    bit_vector (14 downto 0);
               BS : In    bit_vector (14 downto 0);
                C : In    bit_vector (14 downto 0);
               S1 : In    bit;
               S2 : In    bit;
               S3 : In    bit;
                E : Out   bit;
                F : Out   bit;
               ZC : Out   bit_vector (15 downto 0);
               ZS : Out   bit_vector (15 downto 0) );
end component;

component LATCH2
   Port (      A1 : In    bit_vector (14 downto 0);
               A2 : In    bit_vector (14 downto 0);
            CLEAR : In    bit;
              CLK : In    bit;
             LOAD : In    bit;
               Z1 : Out   bit_vector (14 downto 0);
               Z2 : Out   bit_vector (14 downto 0) );
end component;
```

```
   component GAMMA_TABLE
      Port (    IN_D : In    bit_vector (4 downto 0);
              GAMMA1 : Out   bit_vector (14 downto 0);
              GAMMA2 : Out   bit_vector (13 downto 0) );
   end component;

begin

   I_16 : MUX3
      Port Map ( A(14 downto 0)=>MGAMMA1(14 downto 0),
                 C(67 downto 0)=>L1MU3(67 downto 0), SEL=>MX2S3,
                 Z(67 downto 0)=>MU3MA(67 downto 0) );
   I_15 : D_SPLIT
      Port Map ( D(52 downto 0)=>D(52 downto 0),
                 D15(14 downto 0)=>D15(14 downto 0),
                 IN_D(4 downto 0)=>D5(4 downto 0) );
   I_12 : CONTROL
      Port Map ( CLOCK=>CLOCK, RESET=>RESET, CL1=>CL1, CL2=>CL2,
                 DIGIT=>DIGIT, LD1=>LD1, LD2=>LD2, LD3=>LD3, MX1H=>MX1H,
                 MX1L=>MX1L, MX2S1=>MX2S1, MX2S2=>MX2S2, MX2S3=>MX2S3,
                 ROUND=>ROUND );
   I_1 : CONVERT
      Port Map ( A1(13 downto 0)=>QS(13 downto 0),
                 A2(13 downto 0)=>QC(13 downto 0), CLEAR=>CL2,
                 CLK=>CLOCK, DIGIT=>DIGIT, ROUND=>ROUND, SIGN=>SIGN,
                 Q(52 downto 0)=>Q(52 downto 0) );
   I_2 : LATCH1
      Port Map ( A(67 downto 0)=>MU1L1(67 downto 0), CLEAR=>CL1,
                 CLK=>CLOCK, LOAD=>LD1,
                 Z(67 downto 0)=>L1MU3(67 downto 0) );
   I_3 : MUX1
      Port Map ( A(52 downto 0)=>D(52 downto 0),
                 B(53 downto 0)=>X(53 downto 0),
                 C(67 downto 0)=>Z(67 downto 0), SELD=>MX1L, SELZ=>MX1H,
                 Z(67 downto 0)=>MU1L1(67 downto 0) );
   I_4 : CPA
      Port Map ( ZC(69 downto 0)=>RWC(69 downto 0),
                 ZS(69 downto 0)=>RWS(69 downto 0), SIGN=>SIGN,
                 Z(67 downto 0)=>Z(67 downto 0) );
   I_5 : LATCH3
      Port Map ( A1(69 downto 0)=>WS(69 downto 0),
                 A2(69 downto 0)=>WC(69 downto 0), CLEAR=>CL1,
                 CLK=>CLOCK, LOAD=>LD3, Q1(13 downto 0)=>QS(13 downto 0),
```

```
                  Q2(13 downto 0)=>QC(13 downto 0),
                  RWC(69 downto 0)=>RWC(69 downto 0),
                  RWS(69 downto 0)=>RWS(69 downto 0) );
    I_6 : RECODER
       Port Map ( G=>F, H=>E, Q1(15 downto 0)=>MU2RS(15 downto 0),
                  Q2(15 downto 0)=>MU2RC(15 downto 0),
                  M1(7 downto 0)=>M1(7 downto 0),
                  M2(7 downto 0)=>M2(7 downto 0),
                  P1(7 downto 0)=>P1(7 downto 0),
                  P2(7 downto 0)=>P2(7 downto 0) );
    I_7 : MULTADD
       Port Map ( GAMMA2(13 downto 0)=>MGAMMA2(13 downto 0),
                  INZ(67 downto 0)=>MU3MA(67 downto 0),
                  M1(7 downto 0)=>M1(7 downto 0),
                  M2(7 downto 0)=>M2(7 downto 0),
                  P1(7 downto 0)=>P1(7 downto 0),
                  P2(7 downto 0)=>P2(7 downto 0),
                  RWC(69 downto 0)=>RWC(69 downto 0),
                  RWS(69 downto 0)=>RWS(69 downto 0), SCALE=>MX2S3,
                  SEL=>MX2S1, MC(14 downto 0)=>MAC(14 downto 0),
                  MS(14 downto 0)=>MAS(14 downto 0),
                  WC(69 downto 0)=>WC(69 downto 0),
                  WS(69 downto 0)=>WS(69 downto 0) );
    I_8 : MUX2
       Port Map ( AC(13 downto 0)=>QC(13 downto 0),
                  AS(13 downto 0)=>QS(13 downto 0),
                  BC(14 downto 0)=>MC(14 downto 0),
                  BS(14 downto 0)=>MS(14 downto 0),
                  C(14 downto 0)=>D15(14 downto 0), S1=>MX2S1, S2=>MX2S2,
                  S3=>MX2S3, E=>E, F=>F,
                  ZC(15 downto 0)=>MU2RC(15 downto 0),
                  ZS(15 downto 0)=>MU2RS(15 downto 0) );
    I_9 : LATCH2
       Port Map ( A1(14 downto 0)=>MAC(14 downto 0),
                  A2(14 downto 0)=>MAS(14 downto 0), CLEAR=>CL1,
                  CLK=>CLOCK, LOAD=>LD2, Z1(14 downto 0)=>MC(14 downto 0),
                  Z2(14 downto 0)=>MS(14 downto 0) );
    I_11 : GAMMA_TABLE
       Port Map ( IN_D(4 downto 0)=>D5(4 downto 0),
                  GAMMA1(14 downto 0)=>MGAMMA1(14 downto 0),
                  GAMMA2(13 downto 0)=>MGAMMA2(13 downto 0) );

end SCHEMATIC;
```

A.2.2 d_split

```
entity D_SPLIT is
      Port (       D : In    bit_vector (52 downto 0);
                 D15 : Out   bit_vector (14 downto 0);
                IN_D : Out   bit_vector (4 downto 0) );
end D_SPLIT;

architecture BEHAVIORAL of D_SPLIT is

  begin

   process(D)
    variable j : integer ;
    begin

        for j in 0 to 14 loop
         D15(j) <= D(j+38);
        end loop;

        for j in 0 to 4 loop
         IN_D(j) <= D(j+47);
        end loop;

   end process;

end BEHAVIORAL;
```

A.2.3 rec1

```
entity REC1 is
      Port (      C0 : In    bit;
                  C1 : In    bit;
                   H : In    bit;
                  S0 : In    bit;
                  S1 : In    bit;
                   T : In    bit;
                  HN : Out   bit;
                  M1 : Out   bit;
                  M2 : Out   bit;
                  P1 : Out   bit;
                  P2 : Out   bit;
                  TN : Out   bit );
end REC1;
```

```
architecture BEHAVIORAL of REC1 is

   begin

     process(C0,C1,S0,S1,T,H)
        variable w00,w01,w02,w10,w11,nt : bit;
        variable vp1,v00,vm1,vm2 : bit;
       begin

        w02 := S0 AND C0 ;
        w01 := S0 XOR C0 ;
        w00 := NOT(S0 OR C0) ;

        w11 := S1 XOR C1 ;
        w10 := NOT(S1 XOR C1) ;
        nt  := NOT T ;
        TN  <= S1 AND C1 ;

        vp1 := ( T AND w11 AND w02) OR ( T AND w00 AND w10) OR
               (nt AND w01 AND w10) ;
        v00 := ( T AND w11 AND w01) OR (nt AND w02 AND w11) OR
               (nt AND w00 AND w10) ;
        vm1 := ( T AND w10 AND w02) OR ( T AND w00 AND w11) OR
               (nt AND w11 AND w01) ;
        vm2 := ( T AND w10 AND w01) OR (nt AND w02 AND w10) OR
               (nt AND w00 AND w11) ;
        HN  <= (T AND w01) OR w11 OR w02;

        P2  <= (NOT H) AND vm2 ;
        M2  <= H AND vp1 ;

        if ( H = '1' ) then
           M1 <= v00 ;
           P1 <= vm2 ;
         else
           M1 <= vp1 ;
           P1 <= vm1 ;
        end if;

       end process;

end BEHAVIORAL;
```

A.2.4 mp_pp2

```
entity MP_PP2 is
      Port (      M1 : In    bit_vector (5 downto 0);
                  M2 : In    bit_vector (5 downto 0);
                  P1 : In    bit_vector (5 downto 0);
                  P2 : In    bit_vector (5 downto 0);
                 PM1 : In    bit;
                 PM2 : In    bit;
                   Z : In    bit_vector (68 downto 0);
                 NM1 : Out   bit;
                 NM2 : Out   bit;
                  T2 : Out   bit_vector (67 downto 0) );
end MP_PP2;

architecture BEHAVIORAL of MP_PP2 is

-- This block generates one of the partial products multipling
-- Z by (M1,M2,0,P1,P2). The signals M1,M2,P1,P2 are given in
-- bus format because they are driven by a tree of buffers
-- [i.e. M1(0) = M1(1) = ..... = M1(5) ]

   begin

    process (M1, M2, P1, P2, Z, PM1, PM2)
     variable    pd : bit;

      begin

        t2(0) <= PM1 ;
        t2(1) <= PM2 ;
        t2(2) <= ( M1(0) AND NOT(Z(0)) ) OR ( P1(0) AND Z(0) );
        pd := Z(0);
        for i in 1 to 5 loop
            t2(i+2) <= ( M2(0) AND NOT(pd) ) OR ( M1(0) AND NOT(Z(i)) ) OR
                       ( P1(0) AND Z(i) )    OR ( P2(0) AND pd ) ;
            pd := Z(i);
        end loop;
        for i in 6 to 17 loop
            t2(i+2) <= ( M2(1) AND NOT(pd) ) OR ( M1(1) AND NOT(Z(i)) ) OR
                       ( P1(1) AND Z(i) )    OR ( P2(1) AND pd ) ;
            pd := Z(i);
        end loop;
```

```
  for i in 18 to 29 loop
      t2(i+2) <= ( M2(2) AND NOT(pd) ) OR ( M1(2) AND NOT(Z(i)) ) OR
                 ( P1(2) AND Z(i) )    OR ( P2(2) AND pd ) ;
      pd := Z(i);
  end loop;
  for i in 30 to 41 loop
      t2(i+2) <= ( M2(3) AND NOT(pd) ) OR ( M1(3) AND NOT(Z(i)) ) OR
                 ( P1(3) AND Z(i) )    OR ( P2(3) AND pd ) ;
      pd := Z(i);
  end loop;
  for i in 42 to 53 loop
      t2(i+2) <= ( M2(4) AND NOT(pd) ) OR ( M1(4) AND NOT(Z(i)) ) OR
                 ( P1(4) AND Z(i) )    OR ( P2(4) AND pd ) ;
      pd := Z(i);
  end loop;
  for i in 54 to 65 loop
      t2(i+2) <= ( M2(5) AND NOT(pd) ) OR ( M1(5) AND NOT(Z(i)) ) OR
                 ( P1(5) AND Z(i) )    OR ( P2(5) AND pd ) ;
      pd := Z(i);
  end loop;

  NM1 <= M1(0) ;
  NM2 <= M2(0) ;

 end process;

end BEHAVIORAL;
```

A.2.5 csa_70b

```
entity CSA_70B is
      Port (       A : In    bit_vector (69 downto 0);
                   B : In    bit_vector (69 downto 0);
                   D : In    bit_vector (69 downto 0);
                   C : Out   bit_vector (69 downto 0);
                   S : Out   bit_vector (69 downto 0) );
end CSA_70B;

architecture BEHAVIORAL of CSA_70B is

   begin

    process(A,B,D)

    begin

      for i in 0 to 69 loop
        S(i) <= A(i) XOR B(i) XOR D(i) ;
      end loop;

      for i in 0 to 68 loop
        C(i+1) <= ( A(i) AND B(i) ) OR ( A(i) AND D(i) ) OR ( D(i) AND B(i) ) ;
      end loop;

      C(0) <= '0' ;

    end process;

end BEHAVIORAL;
```

A.2.6 quotient

```
entity QUOTIENT is
      Port (   CLEAR : In    bit;
                 CLK : In    bit;
                DET0 : In    bit;
              DET512 : In    bit;
               DIGIT : In    bit;
                  QK : In    bit_vector (8 downto 0);
                  QM : In    bit_vector (8 downto 0);
                  QO : In    bit_vector (8 downto 0);
              QOLSIG : In    bit;
                QSIG : In    bit;
                  QN : InOut bit_vector (53 downto 0);
                  QP : InOut bit_vector (53 downto 0);
                  QQ : InOut bit_vector (53 downto 0);
                  QI : Out   bit_vector (8 downto 0) );
end QUOTIENT;

architecture BEHAVIORAL of QUOTIENT is

   begin
    process(clear,clk)
     begin

       if ( clear = '1' ) then
          for i in 0 to 53 loop
              qq(i) <= '0';
              qn(i) <= '0';
              qp(i) <= '0';
          end loop;
          qi <= "000000000" ;

       elsif ((clk = '1') AND (clk'EVENT)) then
          if ( digit = '1' ) then
                qi <= qk ;
                if ( det0 = '1' ) then
                        for i in 53 downto 9 loop
                          qq(i) <= qq(i-9);
                          qn(i) <= qn(i-9);
                        end loop;
                elsif ( qsig = '0') then
                        for i in 53 downto 9 loop
                          qq(i) <= qq(i-9);
```

```
                    qn(i) <= qq(i-9);
                  end loop;
          elsif ( qsig = '1') then
                  for i in 53 downto 9 loop
                    qq(i) <= qn(i-9);
                    qn(i) <= qn(i-9);
                  end loop;
          end if;
          for i in 0 to 8 loop
            qq(i) <= qk(i);
            qn(i) <= qm(i);
          end loop;
          if ( det512 = '1' ) then
                  for i in 53 downto 9 loop
                    qp(i) <= qp(i-9);
                  end loop;
          elsif ( qolsig = '0') then
                  for i in 53 downto 9 loop
                    qp(i) <= qq(i-9);
                  end loop;
          elsif ( qolsig = '1') then
                  for i in 53 downto 9 loop
                    qp(i) <= qn(i-9);
                  end loop;
          end if;
          for i in 0 to 8 loop
            qp(i) <= qo(i);
          end loop;
    end if;
  end if;

end process;

end BEHAVIORAL;
```

A.2.7 cpa_cla16

```
entity CPA_CLA16 is
      Port (     CIN : In    bit;
                 INC : In    bit_vector (15 downto 0);
                 INS : In    bit_vector (15 downto 0);
                 GGG : Out   bit;
                 PPP : Out   bit;
                 SUM : Out   bit_vector (15 downto 0) );
end CPA_CLA16;

architecture BEHAVIORAL of CPA_CLA16 is

 begin

  process(INS,INC,CIN)
        variable g0,p0 : bit_vector (3 downto 0) ;
        variable g1,p1 : bit_vector (3 downto 0) ;
        variable g2,p2 : bit_vector (3 downto 0) ;
        variable g3,p3 : bit_vector (3 downto 0) ;
        variable gg,pp : bit_vector (3 downto 0) ;
        variable c    : bit_vector (4 downto 0) ;
        variable cc   : bit_vector (4 downto 0) ;
        variable i,j,k,l : integer;
   begin

    cc(0) := CIN;

    for k in 0 to 3 loop
        --------------------------- 1st level -----------------------
        j := k*4 ;
        g0(k) := INS(0+j) AND INC(0+j); p0(k) := INS(0+j) OR INC(0+j);
        g1(k) := INS(1+j) AND INC(1+j); p1(k) := INS(1+j) OR INC(1+j);
        g2(k) := INS(2+j) AND INC(2+j); p2(k) := INS(2+j) OR INC(2+j);
        g3(k) := INS(3+j) AND INC(3+j); p3(k) := INS(3+j) OR INC(3+j);
        --------------------------- 2nd level -----------------------
        gg(k) := g3(k) OR (g2(k) AND p3(k)) OR (g1(k) AND p2(k) AND p3(k))
                       OR (g0(k) AND p1(k) AND p2(k) AND p3(k)) ;
        pp(k) := p0(k) AND p1(k) AND p2(k) AND p3(k) ;
    end loop;

    -- 2nd level carry generation ----------------------------------
    cc(1) := gg(0) OR (cc(0) AND pp(0));
    cc(2) := gg(1) OR (gg(0) AND pp(1)) OR (cc(0) AND pp(0) AND pp(1));
```

```
    cc(3) := gg(2) OR (gg(1) AND pp(2)) OR (gg(0) AND pp(1) AND pp(2))
                   OR (cc(0) AND pp(0) AND pp(1) AND pp(2));

    -- outgoing group-carry generator ------------------------------
    GGG <= gg(3) OR (gg(2) AND pp(3))
                 OR (gg(1) AND pp(2) AND pp(3))
                 OR (gg(0) AND pp(1) AND pp(2) AND pp(3));
    -- outgoing group-carry propagation ----------------------------
    PPP <= pp(0) AND pp(1) AND pp(2) AND pp(3) ;

    for k in 0 to 3 loop
        -- 1st level carry generation ------------------------------
        c(0) := cc(k);
        c(1) := g0(k) OR (c(0) AND p0(k));
        c(2) := g1(k) OR (g0(k) AND p1(k)) OR (c(0) AND p0(k) AND p1(k));
        c(3) := g2(k) OR (g1(k) AND p2(k)) OR (g0(k) AND p1(k) AND p2(k))
                      OR (c(0) AND p0(k) AND p1(k) AND p2(k));
        j := k*4 ;

        for i in 0 to 3 loop
          SUM(i+j)  <= INS(i+j) XOR INC(i+j) XOR c(i) ;
        end loop;

    end loop;

  end process;

end BEHAVIORAL;
```

A.2.8 cpa_gen

```
entity CPA_GEN is
      Port (      C0 : In    bit;
                  G0 : In    bit;
                  G1 : In    bit;
                  G2 : In    bit;
                  G3 : In    bit;
                  P0 : In    bit;
                  P1 : In    bit;
                  P2 : In    bit;
                  P3 : In    bit;
                  C1 : Out   bit;
                  C2 : Out   bit;
                  C3 : Out   bit;
                  C4 : Out   bit );
end CPA_GEN;

architecture BEHAVIORAL of CPA_GEN is

   begin

    C1  <= G0 OR (C0 AND P0);

    C2  <= G1 OR (G0 AND P1)
              OR (C0 AND P0 AND P1);

    C3  <= G2 OR (G1 AND P2)
              OR (G0 AND P1 AND P2)
              OR (C0 AND P0 AND P1 AND P2);

    C4  <= G3 OR (G2 AND P3)
              OR (G1 AND P2 AND P3)
              OR (G0 AND P1 AND P2 AND P3)
              OR (C0 AND P0 AND P1 AND P2 AND P3);

end BEHAVIORAL;
```

A.2.9 latch1

```
entity LATCH1 is
      Port (       A : In    bit_vector (67 downto 0);
               CLEAR : In    bit;
                 CLK : In    bit;
                LOAD : In    bit;
                   Z : Out   bit_vector (67 downto 0) );
end LATCH1;

architecture BEHAVIORAL of LATCH1 is

   begin

     process(clear,clk)

      begin

       if ( clear = '1' ) then

          for i in 0 to 67 loop
              Z(i) <= '0';
          end loop;

       elsif ((clk'EVENT) AND ( clk = '1' )) then

          if ( load = '1' ) then
              Z <= A;
          end if;

       end if;

     end process;

end BEHAVIORAL;
```

A.2.10 mux2

```
entity MUX2 is
      Port (      AC : In    bit_vector (13 downto 0);
                  AS : In    bit_vector (13 downto 0);
                  BC : In    bit_vector (14 downto 0);
                  BS : In    bit_vector (14 downto 0);
                   C : In    bit_vector (14 downto 0);
                  S1 : In    bit;
                  S2 : In    bit;
                  S3 : In    bit;
                   E : Out   bit;
                   F : Out   bit;
                  ZC : Out   bit_vector (15 downto 0);
                  ZS : Out   bit_vector (15 downto 0) );
end MUX2;

architecture BEHAVIORAL of MUX2 is

begin
 process (C, AS, AC, BS, BC, S1, S2, S3)
  begin

-- S1 = 1 -> q goes into the recoder
-- S2 = 1 -> M goes into the recoder
-- S3 = 1 -> D15 goes into the recoder

   for i in 0 to 11 loop
      ZS(i) <= (AS(i+2) and S1) or (BS(i) and S2) or (C(i) and S3);
      ZC(i) <= (AC(i+2) and S1) or (BC(i) and S2);
   end loop;

   for i in 12 to 14 loop
      ZS(i) <= (AS(13) and S1) or (BS(i) and S2) or (C(i) and S3);
      ZC(i) <= (AC(13) and S1) or (BC(i) and S2);
   end loop;

   ZS(15) <= (AS(13) and S1) or (BS(14) and S2) ;
   ZC(15) <= (AC(13) and S1) or (BC(14) and S2) ;

   -- e & f calculation
   E <= (AS(1) OR AC(1)) AND S1;
   F <= (AS(0) AND AC(0) AND (NOT (AS(1) XOR AC(1))) ) AND S1;
```

```
  end process;

end BEHAVIORAL;
```

A.2.11 control

```
entity CONTROL is
      Port (   CLOCK : In    bit;
               RESET : In    bit;
                 CL1 : Out   bit;
                 CL2 : Out   bit;
               DIGIT : Out   bit;
                 LD1 : Out   bit;
                 LD2 : Out   bit;
                 LD3 : Out   bit;
                MX1H : Out   bit;
                MX1L : Out   bit;
               MX2S1 : Out   bit;
               MX2S2 : Out   bit;
               MX2S3 : Out   bit;
               ROUND : Out   bit );
end CONTROL;

architecture BEHAVIORAL of CONTROL is

 begin
  process(reset,clock)
    variable state : integer range 0 to 9;

    begin

        if ( reset = '1' ) then
          CL1 <= '1';
          CL2 <= '1';
          LD1 <= '0';
          LD2 <= '0';
          LD3 <= '0';
          DIGIT <= '0' ;
          ROUND <= '1' ;
          MX1L <= '0' ;
          MX1H <= '0' ;
          MX2S1 <= '0' ;
          MX2S2 <= '0' ;
          MX2S3 <= '1' ;
          state := 9;

        elsif ((clock'EVENT) AND ( clock = '1' )) then
```

```
if( 9 = state ) then
        MX2S1 <= '0' ;
        MX2S2 <= '0' ;
        MX2S3 <= '1' ;
        MX1L <= '0' ;
        MX1H <= '0' ;
        LD3 <= '0' ;
        ROUND <= '0' ;
        DIGIT <= '0' ;
        CL1 <= '0';
        LD1 <= '1' ;
        LD2 <= '1' ;
        CL2 <= '1';
        state  := 0 ;

elsif( 0 = state ) then
        MX2S1 <= '0' ;
        MX2S2 <= '1' ;
        MX2S3 <= '0' ;
        MX1L <= '1' ;
        CL2 <= '0';
        LD2 <= '0' ;
        LD3 <= '1' ;
        state  := 1 ;

elsif( 1 = state ) then
        MX1H <= '1' ;
        state  := 2 ;

elsif( 2 = state ) then
        MX2S1 <= '1' ;
        MX2S2 <= '0' ;
        MX2S3 <= '0' ;
        LD1 <= '0' ;
        DIGIT <= '1' ;
        state  := 3 ;

elsif( 3 = state ) then
        state  := 4 ;

elsif( 4 = state ) then
        state  := 5 ;
```

```
                elsif( 5 = state ) then
                        state  := 6 ;

                elsif( 6 = state ) then
                        state  := 7 ;

                elsif( 7 = state ) then
                        state  := 8 ;

                elsif( 8 = state ) then
                        ROUND <= '1' ;
                        state  := 9 ;
                end if;
        end if;

    end process;

end BEHAVIORAL;
```

A.2.12 gamma_table

```
entity GAMMA_TABLE is
      Port (    IN_D : In    bit_vector (4 downto 0);
              GAMMA1 : Out   bit_vector (14 downto 0);
              GAMMA2 : Out   bit_vector (13 downto 0) );
end GAMMA_TABLE;

architecture BEHAVIORAL of GAMMA_TABLE is

  begin
   process(IN_D)
    begin
     case IN_D is

--       d(i) = 00       gamma1 = 3.878329     gamma2 = 3.938928
         when "00000" => gamma2 <= "00000111110101" ;
                         gamma1 <= "000001111100101" ;

--       d(i) = 01       gamma1 = 3.650217     gamma2 = 3.821321
         when "00001" => gamma2 <= "00010110111000" ;
                         gamma1 <= "000101100110010" ;

--       d(i) = 02       gamma1 = 3.441655     gamma2 = 3.710535
         when "00010" => gamma2 <= "00100101000100" ;
                         gamma1 <= "001000111011110" ;

--       d(i) = 03       gamma1 = 3.250471     gamma2 = 3.605991
         when "00011" => gamma2 <= "00110010011100" ;
                         gamma1 <= "001011111111101" ;

--       d(i) = 04       gamma1 = 3.074787     gamma2 = 3.507178
         when "00100" => gamma2 <= "00111111000110" ;
                         gamma1 <= "001110110011100" ;

--       d(i) = 05       gamma1 = 2.912970     gamma2 = 3.413637
         when "00101" => gamma2 <= "01001011000100" ;
                         gamma1 <= "010001011001001" ;

--       d(i) = 06       gamma1 = 2.763600     gamma2 = 3.324956
         when "00110" => gamma2 <= "01010110011010" ;
                         gamma1 <= "010011110010001" ;

--       d(i) = 07       gamma1 = 2.625431     gamma2 = 3.240766
```

```
        when "00111" => gamma2 <= "01100001001100" ;
                        gamma1 <= "010101111111101" ;

--      d(i) = 08       gamma1 = 2.497371     gamma2 = 3.160735
        when "01000" => gamma2 <= "01101011011100" ;
                        gamma1 <= "011000000010110" ;

--      d(i) = 09       gamma1 = 2.378457     gamma2 = 3.084561
        when "01001" => gamma2 <= "01110101001100" ;
                        gamma1 <= "011001111100100" ;

--      d(i) = 10       gamma1 = 2.267839     gamma2 = 3.011973
        when "01010" => gamma2 <= "01111110011110" ;
                        gamma1 <= "011011101101110" ;

--      d(i) = 11       gamma1 = 2.164762     gamma2 = 2.942723
        when "01011" => gamma2 <= "10000111010110" ;
                        gamma1 <= "011101010111011" ;

--      d(i) = 12       gamma1 = 2.068556     gamma2 = 2.876586
        when "01100" => gamma2 <= "10001111110100" ;
                        gamma1 <= "011110111001111" ;

--      d(i) = 13       gamma1 = 1.978624     gamma2 = 2.813357
        when "01101" => gamma2 <= "10010111111001" ;
                        gamma1 <= "100000010110000" ;

--      d(i) = 14       gamma1 = 1.894433     gamma2 = 2.752847
        when "01110" => gamma2 <= "10011111101001" ;
                        gamma1 <= "100001101100001" ;

--      d(i) = 15       gamma1 = 1.815502     gamma2 = 2.694886
        when "01111" => gamma2 <= "10100111000100" ;
                        gamma1 <= "100010111101000" ;

--      d(i) = 16       gamma1 = 1.741404     gamma2 = 2.639316
        when "10000" => gamma2 <= "10101110001011" ;
                        gamma1 <= "100100001000111" ;

--      d(i) = 17       gamma1 = 1.671751     gamma2 = 2.585991
        when "10001" => gamma2 <= "10110101000000" ;
                        gamma1 <= "100101010000010" ;
```

```
--        d(i) = 18        gamma1 = 1.606196     gamma2 = 2.534778
          when "10010" => gamma2 <= "10111011100100" ;
                          gamma1 <= "100110010011011" ;

--        d(i) = 19        gamma1 = 1.544422     gamma2 = 2.485554
          when "10011" => gamma2 <= "11000001110111" ;
                          gamma1 <= "100111010010101" ;

--        d(i) = 20        gamma1 = 1.486144     gamma2 = 2.438206
          when "10100" => gamma2 <= "11000111111011" ;
                          gamma1 <= "101000001110010" ;

--        d(i) = 21        gamma1 = 1.431105     gamma2 = 2.392628
          when "10101" => gamma2 <= "11001101110000" ;
                          gamma1 <= "101001000110101" ;

--        d(i) = 22        gamma1 = 1.379067     gamma2 = 2.348723
          when "10110" => gamma2 <= "11010011011000" ;
                          gamma1 <= "101001111011111" ;

--        d(i) = 23        gamma1 = 1.329816     gamma2 = 2.306400
          when "10111" => gamma2 <= "11011000110010" ;
                          gamma1 <= "101010101110011" ;

--        d(i) = 24        gamma1 = 1.283158     gamma2 = 2.265575
          when "11000" => gamma2 <= "11011110000001" ;
                          gamma1 <= "101011011110001" ;

--        d(i) = 25        gamma1 = 1.238913     gamma2 = 2.226171
          when "11001" => gamma2 <= "11100011000100" ;
                          gamma1 <= "101100001011011" ;

--        d(i) = 26        gamma1 = 1.196917     gamma2 = 2.188114
          when "11010" => gamma2 <= "11100111111011" ;
                          gamma1 <= "101100110110011" ;

--        d(i) = 27        gamma1 = 1.157021     gamma2 = 2.151336
          when "11011" => gamma2 <= "11101100101001" ;
                          gamma1 <= "101101011111010" ;

--        d(i) = 28        gamma1 = 1.119087     gamma2 = 2.115775
          when "11100" => gamma2 <= "11110001001100" ;
                          gamma1 <= "101110000110001" ;
```

```
--      d(i) = 29       gamma1 = 1.082989     gamma2 = 2.081370
        when "11101" => gamma2 <= "11110101100110" ;
                        gamma1 <= "101110101011001" ;

--      d(i) = 30       gamma1 = 1.048610     gamma2 = 2.048066
        when "11110" => gamma2 <= "11111001110111" ;
                        gamma1 <= "101111001110010" ;

--      d(i) = 31       gamma1 = 1.015842     gamma2 = 2.015811
        when "11111" => gamma2 <= "11111101111111" ;
                        gamma1 <= "101111101111111" ;

       when others => null;

    end case;

  end process;

end BEHAVIORAL;
```

Appendix B
Random Generated Test Vectors

```
-- 0 ------------------------------------------------------------
x = 0.515964404221607608747035556007176637649536132812500000
d = 0.966634690280367969705821451498195528984069824218750000
q = 0.533773936948150007353319779213052242994308471679687500
-- 1 ------------------------------------------------------------
x = 0.380429726620404862202917684044223278760910034179687500
d = 0.545232787982203426224714348791167140007019042968750000
q = 0.697738168000311520522416230960516259074211120605468750
-- 2 ------------------------------------------------------------
x = 0.412076905445231567171049391617998480796813964843750000
d = 0.533733170029582981719329382030991837382316589355468750
q = 0.772065385073203436228084228787338361144065856933593750
-- 3 ------------------------------------------------------------
x = 0.672782088011867340782146129640750586986541748046875000
d = 0.989464876469906773337470440310426056385040283203125000
q = 0.679945396760456954865503576002083718776702880859375000
-- 4 ------------------------------------------------------------
x = 0.743897568548050447390096451272256672382354736328125000
d = 0.898990534664592999547494400758296251296997070312500000
q = 0.827480979903301605560272946604527533054351806640625000
-- 5 ------------------------------------------------------------
x = 0.530011483482090595131808186124544590711593627929687500
d = 0.897958864643219278889318957226350903511047363281250000
q = 0.590240270853250037497161883948137983679771423339843750
-- 6 ------------------------------------------------------------
x = 0.405292767870841896815647942275973036885261535644531250
d = 0.603443030083292697440811025444418191909790039062500000
q = 0.671633853845158723672170708596240729093551635742187500
-- 7 ------------------------------------------------------------
x = 0.855899606065777862617949267587391659617424011230468750
d = 0.866421636364619107695261845947243273258209228515625000
q = 0.987855762301839357419908083102200180292129516601562500
-- 8 ------------------------------------------------------------
x = 0.821588699855650195225109655439155176281929016113281250
d = 0.935813227172853978608202396571869030594825744628906250
q = 0.877940892476714918046809543739072978496551513671875000
```

```
-- 9 ------------------------------------------------------------
x = 0.32417555983372758898752863387926481664180755615234375
d = 0.63129313505780570991987588058691471815109252929687500
q = 0.51351035173864789662445673457114025950431823730468750
-- 10 ------------------------------------------------------------
x = 0.29936971412942264825218785517790820449590682983398438
d = 0.54238456443063198797460700006922706961631774902343750
q = 0.55195102103188709019576663195039145648479461669921875
-- 11 ------------------------------------------------------------
x = 0.37215459317534910210767407079401891678571701049804688
d = 0.61897130944718203870991146686719730496406555175781250
q = 0.60124691967989474239431046953541226685047149658203125
-- 12 ------------------------------------------------------------
x = 0.38590464747320146754461234195332508534193038940429688
d = 0.76446899551175018228121871288749389350414276123046875
q = 0.50480091375696600763944843492936342954635620117187500
-- 13 ------------------------------------------------------------
x = 0.42511243637423612451442522797151468694210052490234375
d = 0.69372030449738741886278603487880900502204895019531250
q = 0.61280091359332145017901893879752606153488159179687500
-- 14 ------------------------------------------------------------
x = 0.48165480349289940820156630252313334494829177856445312
d = 0.75090731785209263726699191465741023421287536621093750
q = 0.64143042961764251685963245108723640441894531250000000
-- 15 ------------------------------------------------------------
x = 0.89020625589890700624096098181325942277908325195312500
d = 0.97927401144023706880403778995969332754611968994140625
q = 0.90904715687253212497154208904248662292957305908203125
-- 16 ------------------------------------------------------------
x = 0.35877100406623030348640668307780288159847259521484375
d = 0.65106570890688608699292672099545598030090332031250000
q = 0.55105191251524032747255432695965282618999481201171875
-- 17 ------------------------------------------------------------
x = 0.52450679918960985137488250984461046755313873291015625
d = 0.54169581879009298663874005796969868242740631103515625
q = 0.96826813313996817900175528848194517195224761962890625
-- 18 ------------------------------------------------------------
x = 0.68479887916929960134382326941704377532005310058593750
d = 0.69728887434307946833200730907265096902847290039062500
q = 0.98208775660979530375271906450507231056690216064453125
```

```
-- 19 ------------------------------------------------------------
x = 0.53116069525999987099851296079577878117561340332031250
d = 0.92869644795018080341009181211120449006557464599609375
q = 0.57194220612383950275159349985187873244285583496093750
-- 20 ------------------------------------------------------------
x = 0.29813971104945041767919633457495365291833877563476562
d = 0.56117217897492088773958585079526528716087341308593750
q = 0.53128027763966267915662911036633886396884918212890625
-- 21 ------------------------------------------------------------
x = 0.82665531259340019332171323185320943593978881835937500
d = 0.90686495807341527264355818260810337960720062255859375
q = 0.91155282297993289741810940540744923055171966552734375
-- 22 ------------------------------------------------------------
x = 0.66461520929104422883426650514593347907066345214843750
d = 0.68255491865917805593966249944060109555721282958984375
q = 0.97371682647401491816907537213410250842571258544921875
-- 23 ------------------------------------------------------------
x = 0.77328659420520373668495039964909665286540985107421875
d = 0.98620390914669442405937616058508865535259246826171875
q = 0.78410416652503855949873923236737027764320373535156250
-- 24 ------------------------------------------------------------
x = 0.61836814583203203454786489601247012615203857421875000
d = 0.92163771410548955831387729631387628614902496337890625
q = 0.67094492376779446551893215655582025647163391113281250
-- 25 ------------------------------------------------------------
x = 0.61749704420450002295694957865634933114051818847656250
d = 0.71710757432370797470611023527453653514385223388671875
q = 0.86109402035928994667557390130241401493549346923828125
-- 26 ------------------------------------------------------------
x = 0.48201113938447603946002573138684965670108795166015625
d = 0.86180623055519822717229772024438716471195220947265625
q = 0.55930338200728924036297939892392605543136596679687500
-- 27 ------------------------------------------------------------
x = 0.41803944200186032853494566552399192005395889282226562
d = 0.73583157371535512503157860919600352644920349121093750
q = 0.56811838052979812818676919050631113350391387939453125
-- 28 ------------------------------------------------------------
x = 0.62627522606694840945351643313188105821609497070312500
d = 0.68630375651936215142256969556910917162895202636718750
q = 0.91253358315150034929530420413357205688953399658203125
```

```
-- 29 ---------------------------------------------------------
x = 0.46477593910637121643603109077957924455404281616210938
d = 0.58958483305274733687895150069380179047584533691406250
q = 0.78831054167363556750558473140699788928031921386718750
-- 30 ---------------------------------------------------------
x = 0.46860553730214271617171561956638470292091369628906250
d = 0.81975813411164943911302316337241791188716888427734375
q = 0.57163877710095345463514604489319026470184326171875000
-- 31 ---------------------------------------------------------
x = 0.56885884426015376202911966174724511802196502685546875
d = 0.65475308250391539566237497638212516903877258300781250
q = 0.86881430490516553533097976469434797763824462890625000
-- 32 ---------------------------------------------------------
x = 0.48541192162568302936875852537923492491245269775390625
d = 0.59336564368259425705787180049810558557510375976562500
q = 0.81806543198739978262068461845046840608119964599609375
-- 33 ---------------------------------------------------------
x = 0.34822445076341945746634110037120990455150604248046875
d = 0.65562272242066577110364278269116766750812530517578125
q = 0.53113542111188294381207697369973175227642059326171875
-- 34 ---------------------------------------------------------
x = 0.39532726567486642377247108015581034123897552490234375
d = 0.72760959701966942958506479044448590278625488281250000
q = 0.54332332516523906829064571866183541715145111083984375
-- 35 ---------------------------------------------------------
x = 0.58431917013801593085986496589612215757369995117187500
d = 0.88693395344863368290333482946152798831462860107421875
q = 0.65880798436684995245116169826360419392585754394531250
-- 36 ---------------------------------------------------------
x = 0.78878177599459031732465064123971387743949890136718750
d = 0.91097448296424676783544782665558159351348876953125000
q = 0.86586593888771723470654251286759972572326660156250000
-- 37 ---------------------------------------------------------
x = 0.79379891152204895554689301206963136792182922363281250
d = 0.95339698551846530083508923780755139887332916259765625
q = 0.83260060979778993672795195379876531660556793212890625
-- 38 ---------------------------------------------------------
x = 0.29676470069529703454946911733713932335376739501953125
d = 0.56708550572725269223184341171872802078723907470703125
q = 0.52331561589590325223042555080610327422618865966796875
```

```
-- 39 ------------------------------------------------------------
x = 0.46980044733257986244723269919632002711296081542968750
d = 0.71189754745545674730067275959299877285957336425781250
q = 0.65992704850830397056427045754389837384223937988281250
-- 40 ------------------------------------------------------------
x = 0.49436161003278644709979516846942715346813201904296875
d = 0.55709793863682910419754534814273938536643981933593750
q = 0.88738725410193930454028077292605303227901458740234375
-- 41 ------------------------------------------------------------
x = 0.92900512177916483302908545738318935036659240722656250
d = 0.95274549860169432946577217080630362033843994140625000
q = 0.97508214223276601373413541296031326055526733398437500
-- 42 ------------------------------------------------------------
x = 0.45945208471242898751185634864668827503919601440429688
d = 0.76508400578288549009897678843117319047451019287109375
q = 0.60052501586709639003203164975275285542011260986328125
-- 43 ------------------------------------------------------------
x = 0.34428853615852472724867539000115357339382171630859375
d = 0.54517939549180649549953159294091165065765380859375000
q = 0.63151421166227661441183727220050059258937835693359375
-- 44 ------------------------------------------------------------
x = 0.47569388126753919809885928771109320223331451416015625
d = 0.61812895052979177634711049904581159353256225585937500
q = 0.76957062253729902057131084802676923573017120361328125
-- 45 ------------------------------------------------------------
x = 0.63476422854455383237848309363471344113349914550781250
d = 0.88859883690653318488728018564870581030845642089843750
q = 0.71434285324337098987967920038499869406223297119140625
-- 46 ------------------------------------------------------------
x = 0.46894354232072066324121806246694177389144897460937500
d = 0.70362307303753823806147238428820855915546417236328125
q = 0.66646981926884962810220258688786998391151428222656250
-- 47 ------------------------------------------------------------
x = 0.54335191941044846952735269951517693698406219482421875
d = 0.90871092789280738521995317569235339760780334472656250
q = 0.59793703666623332360074982716469094157218933105468750
-- 48 ------------------------------------------------------------
x = 0.39849435847648156938660690684628207236528396606445312
d = 0.73980082117011813913620699167950078845024108886718750
q = 0.53865087341508544049872853065608069300651550292968750
```

```
-- 49 -------------------------------------------------------------
x = 0.56433365008064251266972632947727106511592864990234375
d = 0.58764324830269587529585351148853078484535217285156250
q = 0.96033375982897950517980234508286230266094207763671875
-- 50 -------------------------------------------------------------
x = 0.48370520921130905067641947425727266818284988403320312
d = 0.64865282045148908718346092427964322268962860107421875
q = 0.74570740149504055604978702831431291997432708740234375
-- 51 -------------------------------------------------------------
x = 0.48728860099208010092652898492815438657999038696289062
d = 0.75619219441720852969979205226991325616836547851562500
q = 0.64439781921794325736385644631809554994106292724609375
-- 52 -------------------------------------------------------------
x = 0.55962730341573585501890875093522481620311737060546875
d = 0.76837611327337851374608135301969014108180999755859375
q = 0.72832470159913409446517107426188886165618896484375000
-- 53 -------------------------------------------------------------
x = 0.35479458985142153792935459932778030633926391601562500
d = 0.65315670503916067879401907703140750527381896972656250
q = 0.54319979740566159964743064847425557672977447509765625
-- 54 -------------------------------------------------------------
x = 0.41773080961673092481589719682233408093452453613281250
d = 0.64919007436800280075317459704820066690444946289062500
q = 0.64346456624956682723848189198179170489311218261718750
-- 55 -------------------------------------------------------------
x = 0.43252712624730871304734591831220313906669616699218750
d = 0.82418483906620410017751510167727246880531311035156250
q = 0.52479383961655867718576473635039292275905609130859375
-- 56 -------------------------------------------------------------
x = 0.70628801323766254860458957409719005227088928222656250
d = 0.79405937427378225912377729400759562849998474121093750
q = 0.88946498979828525488500190476770512759685516357421875
-- 57 -------------------------------------------------------------
x = 0.38846516883394921482164363624178804457187652587890625
d = 0.62519218266252063465060473390622064471244812011718750
q = 0.62135320883185629625700130418408662080764770507812500
-- 58 -------------------------------------------------------------
x = 0.55914338005666786024505654495442286133766174316406250
d = 0.96550741021777852779450768139213323593139648437500000
q = 0.57911868323263127056321764030144549906253814697265625
```

```
-- 59 -----------------------------------------------------------
x = 0.335185789077163565075068163423566147685050964355546875
d = 0.510531142358915501766603028954705223441123962402343750
q = 0.656543276730256697426568734954344108700752258300781250
-- 60 -----------------------------------------------------------
x = 0.583636360514739660487748551531694829463958740234375000
d = 0.805135806931711606182489049388095736503601074218750000
q = 0.724891820100408645899392467981670051813125610351562500
-- 61 -----------------------------------------------------------
x = 0.449564989749139609642725190497003495693206787109375000
d = 0.521523445156180986970184676465578377246856689453125000
q = 0.862022587718003885370876560045871883630752563476562500
-- 62 -----------------------------------------------------------
x = 0.508758879969249955266263896191958338022232055664062500
d = 0.942481899141558443488975171931087970733642578125000000
q = 0.539807587214824247112687771732453256845474243164062500
-- 63 -----------------------------------------------------------
x = 0.465117186640909507922003740532090887427330017089843750
d = 0.805747597155043626671044876275118440389633178710937500
q = 0.577249238202085179771927414549281820654869079589843750
-- 64 -----------------------------------------------------------
x = 0.682282720311676471602879701094934716820716857910156250
d = 0.994568023595292172167603439447702839970588684082031250
q = 0.686009105586637857321363753726473078131675720214843750
-- 65 -----------------------------------------------------------
x = 0.446695422845285128321535239592776633799076080322265620
d = 0.649693138734294572955718649609480053186416625976562500
q = 0.687548315063814197145575235481373965740203857421875000
-- 66 -----------------------------------------------------------
x = 0.643220844046781259351064363727346062660217285156250000
d = 0.867968047441899814842258820135612040758132934570312500
q = 0.741065118632534991505167454306501895189285278320312500
-- 67 -----------------------------------------------------------
x = 0.452942666575751551327755350939696654677391052246093750
d = 0.702848147462517114369973114662570878863334655761718750
q = 0.644438870915437611586185084888711571693420410156250000
-- 68 -----------------------------------------------------------
x = 0.318172080357639164294170086577651090919971466064453120
d = 0.615474512854346289536522363050607964396476745605468750
q = 0.516954112172855251294834033615188673138618469238281250
```

```
-- 69 ---------------------------------------------------------
x = 0.856004852734508325795559358084574341773986816406250000
d = 0.971805780181570821874004195706220343708992004394531250
q = 0.880839433342919142511107111204182729125022888183593750
-- 70 ---------------------------------------------------------
x = 0.382332293727589922482934525760356336832046508789062500
d = 0.721059105229125862912553657224634662270545959472656250
q = 0.530237106715542916290928587841335684061050415039062500
-- 71 ---------------------------------------------------------
x = 0.795990619247774922051519297383492812514305114746093750
d = 0.970952600692842393570458625617902725934982299804687500
q = 0.819803787208335466019093473732937127351760864257812500
-- 72 ---------------------------------------------------------
x = 0.515118479502908233058633413747884333133697509765625000
d = 0.572920956915673240672504107351414859294891357421875000
q = 0.899109158575826405979114497313275933265686035156250000
-- 73 ---------------------------------------------------------
x = 0.298072391677681514110531679762061685323715209960937500
d = 0.574261859559575982281387496186653152108192443847656250
q = 0.519053088265839179804572722787270322442054748535156250
-- 74 ---------------------------------------------------------
x = 0.538428366900621124813142159837298095226287841796875000
d = 0.766516361742520913047371777793159708380699157714843750
q = 0.702435582296785487876888964819954708218574523925781250
-- 75 ---------------------------------------------------------
x = 0.584793002151322127701860154047608375549316406250000000
d = 0.622064727648191428954760340275242924690246582031250000
q = 0.940083846840536030420309998589800670742988586425781250
-- 76 ---------------------------------------------------------
x = 0.571652168674232519229860827181255444884300231933593750
d = 0.983922981882431990641180163947865366935729980468750000
q = 0.580992800453296709406458830926567316055297851562500000
-- 77 ---------------------------------------------------------
x = 0.321794086518601529789407322823535650968551635742187500
d = 0.580411048876313007127691889763809740543365478515625000
q = 0.554424467179942759287314402172341942787170410156250000
-- 78 ---------------------------------------------------------
x = 0.463202440511995217065077667939476668834686279296875000
d = 0.573822546319022075422822126711253076791763305664062500
q = 0.807222447921161712081072892033262178301811218261718750
```

```
-- 79 -------------------------------------------------------------
x = 0.44307932313209369423745442873041611164808273315429688
d = 0.60868760133566690573303503697388805449008941650390625
q = 0.72792565867914427180096481606597080826759338378906250
-- 80 -------------------------------------------------------------
x = 0.56839056968148360393655593725270591676235198974609375
d = 0.77954949172192700146410970774013549089431762695312500
q = 0.72912698387626440066355826274957507848739624023437500
-- 81 -------------------------------------------------------------
x = 0.37919037015139606117131165774480905383825302124023438
d = 0.71161141396109539591918746737064793705940246582031250
q = 0.53286156280248597738591342931613326072692871093750000
-- 82 -------------------------------------------------------------
x = 0.64751753916382681630636852787574753165245056152343750
d = 0.66426607345429533602043647988466545939445495605468750
q = 0.97478640719467524178298845072276890277862548828125000
-- 83 -------------------------------------------------------------
x = 0.45722978082822157697151510546973440796136856079101562
d = 0.78386170011193567752627586742164567112922668457031250
q = 0.58330414761038718030050631568883545696735382080078125
-- 84 -------------------------------------------------------------
x = 0.38987029327073613460541423592076171189546585083007812
d = 0.77046441415812094710702240263344720005989074707031250
q = 0.50601985777207325600812737320666201412677764892578125
-- 85 -------------------------------------------------------------
x = 0.37783374014675324970014003156393300741910934448242188
d = 0.54440517376382147052282789445598609745502471923828125
q = 0.69403039933391286187003288432606495916843414306640625
-- 86 -------------------------------------------------------------
x = 0.49576175981003872683672284438216593116521835327148438
d = 0.55165809930845077779792973160510882735252380371093750
q = 0.89867575665347299018748117305221967399120330810546875
-- 87 -------------------------------------------------------------
x = 0.25767888722833193204664326003694441169500350952148438
d = 0.50664199889015493205590701109031215310096740722656250
q = 0.50860151308577028483881576903513632714748382568359375
-- 88 -------------------------------------------------------------
x = 0.31228952822847733106215173393138684332370758056640625
d = 0.61150255804485764699052197101991623640060424804687500
q = 0.51069210442381973713565912476042285561561584472656250
```

```
-- 89 ------------------------------------------------------------
x = 0.58090385868256166901346659869886934757232666015625000
d = 0.66300742359040643059131525660632178187370300292968750
q = 0.87616493875253076417664033215260133147239685058593750
-- 90 ------------------------------------------------------------
x = 0.43900945989368922450779564314871095120906829833984375
d = 0.66569686106671432934689391913707368075847625732421875
q = 0.65947353152637577800021517759887501597404479980468750
-- 91 ------------------------------------------------------------
x = 0.78507215147142861422224768830346874892711639404296875
d = 0.94967108869444161189932174238492734730243682861328125
q = 0.82667795283807676831600019795587286353111267089843750
-- 92 ------------------------------------------------------------
x = 0.64961984294914631998807408308493904769420623779296875
d = 0.92866032474146242847723442537244409322738647460937500
q = 0.69952363166801534433147935487795621156692504882812500
-- 93 ------------------------------------------------------------
x = 0.53008213733792408639544646575814113020896911621093750
d = 0.57602472397313675411822941896389238536357879638671875
q = 0.92024198836757709063505217272904701530933380126953125
-- 94 ------------------------------------------------------------
x = 0.50248287082765386024618692317744717001914978027343750
d = 0.91624078383494200750192248960956931114196777343750000
q = 0.54841792648052944958436683009495027363300323486328125
-- 95 ------------------------------------------------------------
x = 0.34235616277081715175256704242201521992683410644531250
d = 0.57087344074196810783661248933640308678150177001953125
q = 0.59970588634471155842220468912273645401000976562500000
-- 96 ------------------------------------------------------------
x = 0.69579027555686900896603219734970480203628540039062500
d = 0.94309306607725706950162702923989854753017425537109375
q = 0.73777477598363627731004044107976369559764862060546875
-- 97 ------------------------------------------------------------
x = 0.78248485493589414740966958561330102384090423583984375
d = 0.84330781495352635790396789161604829132556915283203125
q = 0.92787573061802575136169934921781532466411590576171875
-- 98 ------------------------------------------------------------
x = 0.60735913929872176186819388021831400692462921142578125
d = 0.69694441635950676872113263016217388212680816650390625
q = 0.87145994004983318159673899572226218879222869873046875
```

```
-- 99 ------------------------------------------------------------
x = 0.62716951506546214645254622155334800481796264648437500
d = 0.88709972607302467473289198096608743071556091308593750
q = 0.70698873715336318923618819098919630050659179687500000
```