

# **Modeling Of Arithmetic Units For Early Design Space Exploration Of Algorithms Trade-Offs**

Bing Zhang

Kongens Lyngby 2008  
IMM-Master-2008

Technical University of Denmark  
Informatics and Mathematical Modelling  
Building 322, DK-2800 Kongens Lyngby, Denmark  
Phone +45 50132972  
s050985@imm.dtu.dk  
[www.imm.dtu.dk](http://www.imm.dtu.dk)

# Summary

---

This thesis work is part of a larger research project on developing a tool which would allow the exploration of arithmetic algorithms and designs in a standardized and flexible way. The tool should provide a good estimate of the performance, in terms of delay, area and power dissipation, of the implementation of a given algorithm at an early stage of the design process, and allow a standardized comparison with preexisting implementations.

This thesis is focus on the delay and area estimation and comparison based on logical effort with given algorithms. In addition, basic modules such as adders, multipliers and dividers are developed and verified against actual implementation of standard arithmetic operators. It verified that logical effort can be used for different modules' delay estimation in this tool, taking into account a number of design constraints/specifications such as: bit-width, fan-out and optimization criteria.

In the implementation part, several modules such as carry save adders (CSA), carry propagate adders (CPA), Radix 4 multiplier, Radix 2, 4 and 8 dividers are implemented with VHDL and estimated with software tools Spice and Synopsys. Standard blocks that can be used for future exploration are estimated, and nonstandard blocks are estimated separately.



# Preface

---

This thesis is prepared at Informatics Mathematical Modelling, the Technical University of Denmark in partial fulfillment of the requirements for acquiring the master degree in engineering.

This thesis work is part of a larger research project on developing a tool which would allow the exploration of arithmetic algorithms and designs in a standardized and flexible way. The main focus is on the delay and area estimation and comparison based on logical effort with given algorithms.

The thesis consists of a summary report and a collection of twelve research papers and books written during the period 1990–2007, and elsewhere published.

Lyngby, January 2008

Bing Zhang



# Acknowledgements

---

I would like to start thanking my supervisor, Alberto Nannarelli, for his support, patience and wise supervision during the preparation of this thesis. I owe him my profound admiration and respect. I would like also to acknowledge Flemming Stassen for his guidance and encouragement during the initial stages of this work.

Many thanks to my classmates Di Wang and Wei Liu, for their relevant technical questions and support, comments and suggestions for improving this thesis.

I wouldd like to conclude by thanking my parents who have patiently waited for this work to be concluded and for all their other supports.





# Contents

---

<b>Summary</b>	<b>i</b>
<b>Preface</b>	<b>iii</b>
<b>Acknowledgements</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 What Is This Thesis About . . . . .	2
1.3 Related Work . . . . .	4
<b>2 Background</b>	<b>7</b>
2.1 Performance Measurement : Delay, Area and Power Consumption	7
2.2 Logical Effort . . . . .	9
2.3 Simulation Tools . . . . .	16

<b>3</b>	<b>Division Algorithms Simulation</b>	<b>17</b>
3.1	Propagation Delay and Transition Time . . . . .	17
3.2	Delay Analysis for SPICE Modeling and Logical Effort . . . . .	22
3.3	Dividers . . . . .	30
3.4	Delay, area and power evaluation . . . . .	42
<b>4</b>	<b>Select Function</b>	<b>49</b>
4.1	CPA Modeling . . . . .	50
4.2	Select Function with Select Table . . . . .	54
4.3	Select Function with Comparison . . . . .	59
4.4	Simulation Analyzation . . . . .	62
<b>5</b>	<b>Conclusion</b>	<b>63</b>
	<b>Bibliography</b>	<b>65</b>
<b>A</b>	<b>Synopsys Report without Select Function</b>	<b>67</b>
A.1	Radix-2 Timing Report . . . . .	67
A.2	Radix-2 Area Report . . . . .	70
A.3	Radix-2 Power Report . . . . .	71
A.4	Radix-4 Timing Report . . . . .	74

---

A.5 Radix-4 Area Report . . . . .	77
A.6 Radix-4 Power Report . . . . .	78
A.7 Radix-8 Timing Report . . . . .	82
A.8 Radix-8 Area Report . . . . .	86
A.9 Radix-8 Power Report . . . . .	87
 <b>B Synopsys Report with Select Function</b>	 <b>95</b>
B.1 Radix-2 Timing Report . . . . .	95
B.2 Radix-4 Timing Report (select table) . . . . .	98
B.3 Radix-4 Timing Report (Comparison) . . . . .	101
B.4 Radix-8 Timing Report . . . . .	104
 <b>C VHDL Code</b>	 <b>111</b>



# Introduction

---

## 1.1 Motivation

This thesis work is part of a larger research project on developing a tool which would allow the exploration of arithmetic algorithms and designs in a standardized and flexible way. The tool will provide a good estimate of the performance, in terms of delay, area and power dissipation, of the implementation of a given algorithm at an early stage of the design process, and allow a standardized comparison with preexisting implementations. When designing arithmetic modules, such as adder, multiplier, divider, square root and squares sum, in gate level; multiple algorithms can be implemented. However, different algorithms can lead to multiple performances, which are mainly measured with delay time, area and power consumption. The delay time, area and power consumption can not be optimized to the smallest at the same time. Usually, a module that has shorter delay time will cost more area and higher power consuming. Besides, a higher delay time may earn a less area and lower power consuming. One of the goals of this thesis work is to find a delay calculation scheme for general modules and archive a trade-off performance optimization among these measuring parameters. In addition, this tool will provide multiple standard blocks which can be chosen

to implement different preexisting arithmetic algorithms.

This thesis is focus on the delay time and area estimation and comparison based on logical effort with given algorithms. Logical effort is a method to analyze components' delays in MOS integrated circuit level. It is a very succinct and easy way to measure delay time through critical path. By comparing delay estimates of different logic structures, the fastest candidate can be selected.[2] This method also specifies the proper number of logic stages on a path and the best transistor sizes for the logic gates.[2] It presumes that the MOS circuit can be treated as a RC module. The logical effort includes two parts, the logical effort and electrical effort; details will be presented in Chapter 2. The thesis will start from some basic modules such as adders and multipliers; these modules will be developed and verified against actual implementation of standard arithmetic operators. It first verified that logical effort can be used for different modules' delay estimation in this tool through experiments comparing with simulated data; then, some more complicated modules such as dividers with different radices will be implemented in the following step; these modules contains standard blocks such as buffer, multiplexers, and carry save adder, and nonstandard blocks such as select functions and on-the-fly converter(OTFC); adders and select functions are implemented with different algorithms; these will be analyzed in Chapter 3 and 4. When implementing and analyzing modules, we will take into account a number of design constraints or specifications such as: bit-width, fan-out and optimization criteria.

## 1.2 What Is This Thesis About

The thesis work starts from basic components such as fan out 4 inverter, NAND2 gate, NOR gate XOR gates and 3 to 1 multiplexer, all of which are basic and frequently used components in MOS circuits. Typically, INVERTER, NAND2 and NOR gate is the most basic CMOS components, all the other gates can be generated from these three components. They are also the basic components for applying logical effort. table 1.1 compares these components' logical delay with

simulation delay.

Module	Logical effort	SPICE simulation	Error
Inverter	$5\tau$	$2.8394e^{-11}$	0%
Nand2	$6\tau$	$3.2483e^{-11}$	4%
Nor2	$8\tau$	$4.1660e^{-11}$	4%
Xor	$8\tau$	$5.6000e^{-11}$	4%
[3:2] adder	$15.42857\tau$	$9.9098e^{-11}$	11%
[4:2] adder	$21.80952\tau$	$11.878e^{-11}$	19%

Table 1.1: Logical effort and spice simulation for some components

Some more complicated modules like adders, multiplier and dividers are implemented based on these basic components according to the algorithm in [1]. We implemented some carry save adders(CSA), carry propagate adders(CPA).

For dividers, we implement radix 2, 4 and 8 division; calculate delay with logical effort and Synopsys simulation tool. The comparison result is shown in table 1.2. When calculating division delay, we divided the module blocks into two types, the standard blocks like multiplexer and adder, the nonstandard blocks like select function and on-the-fly converter. Since in our scheme, the on-the-fly converter is not on the critical path, we only discuss the select function in detail. For radix 2 and 8 division, the select functions are implemented with select table according to the scheme on [1] and [9]. For radix 4 division, we implemented two select function schemes: the select table and select function with comparison, by using which the iteration delay will be reduced evidently. Table 1.3 shows the comparison between two select function schemes. From the result we can see that using comparison method can save 30% of the timing delay than using select table.

division with select table	delay( $u : ns$ )
radix-2 division	1.17
radix-4 division	1.50
radix-8 division	1.90

Table 1.2: delay analyzation for division with select table

division	delay with select table( $u : ns$ )	delay with comparison( $u : ns$ )
radix 4	1.50	1.13

Table 1.3: comparison between two scheme.

### 1.3 Related Work

This thesis models and simulates modules represented in [1]. Book [1] gives a particular account of various digital arithmetic algorithms. It concentrates on a thorough presentation of alternative algorithms and implementation for addition/subtraction(of two or more than two operands), multiplication, division, and square root. These algorithms and implementations can be directly used for fixed-point applications.[1] This thesis work mainly uses 3 to 2 carry-save adder, 4 to 2 carry-save adder, radix 4 multiplication, radix 2, 4 and 8 division for verifying logical effort, which is explicitly described in [2]. The method of logical effort, a term coined by Lvan Sutherland and Robert Sproull in 1991, is a straightforward technique used to estimated delay in a CMOS circuit. Circuit topology and gate sizing are key components utilized in this method. The model describes delays caused by the capacitive load that the logic gate drives and by the topology of the logic gate.[2] One of the strengths of the method of logical effort is that it combines into one framework the effects on performance of capacitive load, of the complexity of the logic function being computed, and of the number of stages in the network.[2]

Chapter 2 introduces the performance measure standards, logical effort and simple description of simulation tools: SPICE and SYNOPSYS. Chapter 3 introduces basic components and radix 2, 4 and 8 dividers with delay measured in terms



of FO4. Chapter 4 discusses the quotient digits selection function of dividers. Chapter 5 is conclusion.



# Background

---

This chapter includes three parts: performance measuring schemes, logical effort and simple description of simulation tools: SPICE and SYNOPSIS. Delay, area and power consumption are three parameters that are mainly used for VLSI circuit performance measurement. These three factors are interacted with each other to get the final performance. Logical effort is an easy and frequently way to estimate the delay in an MOS circuit. This method is the basement of this project for computing delay performance for different kinds of modules. Finally, SPICE and SYNOPSIS are used for module simulation and performance analysis.

## 2.1 Performance Measurement : Delay, Area and Power Consumption

Delay, area and power consuming are three factors that are frequently used for module performance measurement.

Delay calculation in integrated circuit design means calculating gate delays of logical gates and the wires attached to them. In our simulation, the wire delay

is ignored; only logical gates delay is considered. There are many methods can be used for delay calculation of logical gate itself. For example, circuit simulator such as SPICE can be used, which is the most accurate method; a very simple model called the K-factor model is sometimes used. This approximates the delay as a constant plus  $k$  times the load capacitance; a more complex model called Delay Calculation Language, or DCL, calls a user-defined program whenever a delay value is required. This allows arbitrarily complex models to be represented, but raises significant software engineering issues; Logical effort provides a simple delay calculation that accounts for gate sizing and is analytically tractable. In our simulation, both of the first method and the last method are used. Logical effort, which will be explained particularly below, is used for module delay calculating, and SPICE is used for verifying the results obtained from logical effort.

Area is another factor used for VLSI circuit performance measurement. In our project, area is roughly calculated by adding the area of all logical gates together; the area of wire attached to them is ignored. The module speed increasing, which means the delay goes down, will cause area increases directly. In conventional restoring radix  $2_k$  division, area increases exponentially while the speed increases linearly.[4] As technique develops, area is much smaller than before, most leading semiconductor companies, like Intel, AMD, Infineon, Texas Instruments, IBM, and TSMC are looking into 90nm, 60nm and 45nm processes. In our project, Modules are simulated with VHDL based on a standard CMOS cell library with feature of 90nm.

Energy dissipation in CMOS includes two parts: dynamic dissipation and static dissipation. Dynamic dissipation is caused by charging or discharging of load capacitances and by the short-circuit current. Static dissipation is caused by leakage current and other current drawn continuously from the power supply. Since the increasing densities and faster speed, low power consumption is becoming more and more important now. In this project, we only present the power analysis data, but does not analyze energy consumption particularly. However, paper [5] presents the relation between division radix and energy dissipated explicitly.

These three factors are interacted with each other. Commonly, delay will decrease while area might increase; using faster clock might cause lower delay but higher energy consumption; using less number of cells can reduce both of area and power consumption sometimes, which may cause higher delay. Therefore, a trade-off should be found for lower delay, smaller area and low power consumption.

## 2.2 Logical Effort

### 2.2.1 Logical effort for single stage

Logical effort is particularly introduced in [2]. It is a method to estimate the delay in an MOS circuit. It also specifies the proper number of logic stages on a path and the best transistor sizes for the logic gates. Because the method is easy to use, it is ideal for evaluating alternatives in the early stages of a design and provides a good starting point for more intricate optimizations.

The delay incurred by a logic gate is comprised of two components, a fixed part called the parasitic delay,  $p$ , and a part that is proportional to the load on the gate's output, called the effort delay or stage effort,  $f$ . The total delay is the sum of effort delay and parasitic delay as shown in equation (2.1).

$$d = f + p \quad (2.1)$$

Effort delay,  $f$ , depends on the load and on properties of the logic gate driving the load.

$$f = g \cdot h \quad (2.2)$$

Equation (2.2) shows the two factors that define effort delay. The logical effort,

$g$ , captures properties of the logic gate; while the electrical effort,  $h$ , characterizes the load.

$$h = \frac{C_{out}}{C_{in}} \quad (2.3)$$

Electrical effort,  $h$ , is defined by equation (2.3), where  $C_{out}$  is the capacitance that loads the logic gate and  $C_{in}$  is the capacitance presented by the logic gate at one of its input terminals. Logical effort,  $g$ , captures the effect of the logic gate's topology on its ability to produce output current.

Combining equation (2.1) and (2.2), we obtain the basic equation that models the delay through a single logic gate in (2.4), in units of  $\tau$ , which is the delay of a minimum sized inverter with fanout of four inverters (FO4 delay).

$$d = gh + p \quad (2.4)$$

Equation (2.4) shows that logical effort  $g$  and electrical effort  $h$  both contribute to the delay in the same way. It separates  $\tau$ ,  $g$ ,  $h$ , and  $p$ , the four contributions to delay. The process parameter  $\tau$  represents the speed of the basic transistors. The parasitic delay,  $p$ , expresses the intrinsic delay of the gate due to its own internal capacitance. The electrical effort,  $h$ , combines the effects of external load, which establishes  $C_{out}$ , with the sizes of the transistors in logic gate, which establishes  $C_{in}$ . The logical effort,  $g$ , expresses the effects of circuit topology on the delay free of considerations of loading or transistor size.

Let us consider about the calculating of logical effort,  $g$ . logical effort relates the input capacitance to the output drive current available. Its formulation is :

$$g = \frac{C_{in}}{C_{inv}} = \frac{W_{gp} + W_{gn}}{W_{invp} + W_{invn}} \quad (2.5)$$

$C_{in}$  is the combined input capacitance of every signal in the input group, and  $C_{inv}$  is the input capacitance of an inverter designed to have the same drive capabilities as the logic gate whose logical effort we are calculating.  $W_{gp}$  and  $W_{gn}$  are the widths of the  $p$  and  $n$  transistors of the corresponding input of the gate and  $W_{invp}$ ,  $W_{invn}$  are the widths of the inverter transistors. Concrete demonstration can be read in [2] and [6].

According equation (2.5), logical efforts of inverter, NAND2, and XOR gate are calculated in figure 2.1 below. The ratio of the  $p$  transistor width to  $n$  transistor width is 2.

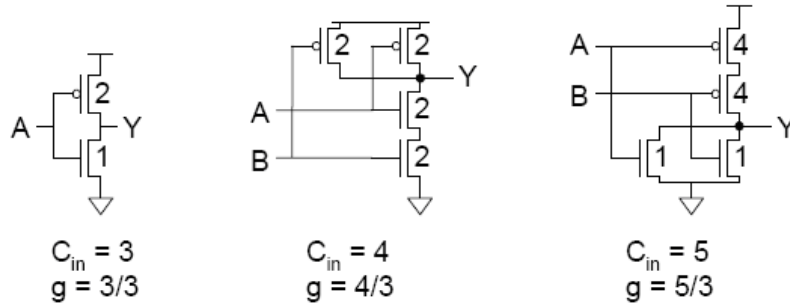


Figure 2.1: logical efforts for simple components

Figure 2.2 is cited from [2]. It contains expressions for logical effort of all the components that needed for developing our simulating modules described in chapter 3 and 4.

In figure 2.2, the expressions are applied to logic gates with an arbitrary number of inputs,  $n$ ; besides, they use a parameter for the ratio of p-type to n-type transistor widths, so as to permit calculation of logical effort for gates fabricated with various CMOS processes. Each logic gate is designed to have a pull down drive equivalent to an n-type transistor of width 1 and a pull up drive equivalent to a p-type transistor of width  $\gamma$ .  $n$  presents the number of inputs for each logic gate. Transistor structures of these components can be seen in Appendix.

<i>Gate type</i>	Logical effort	Formula	$n = 2$ $\gamma = 2$	$n = 3$ $\gamma = 2$	$n = 4$ $\gamma = 2$
NAND	total per input	$\frac{n(n+\gamma)}{1+\gamma}$ $\frac{(n+\gamma)}{1+\gamma}$	8/3 4/3	5 5/3	8 2
NOR	total per input	$\frac{n(1+n\gamma)}{1+\gamma}$ $\frac{1+n\gamma}{1+\gamma}$	10/3 5/3	7 7/3	12 3
multiplexer	total $d, s^*$	$4n$ 2,2	8 2,2	12 2,2	16 2,2
XOR, XNOR, parity (symmetric)	total per bundle	$n^2 2^{n-1}$ $n 2^{n-1}$	8 4	36 12	128 32
XOR, XNOR, parity (asymmetric)	total per bundle		8 4,4	24 6,12,6	48 8,16,16,8
majority (symmetric)	total per input			12 4	
majority (asymmetric)	total per input			10 4,4,2	
C-element	total per input	$n^2$ $n$	4 2	9 3	16 4
latch (dynamic)	total $d, \phi^*$	4 2,2			
upper bounds	total per bundle	$\frac{\gamma n^2 2^n}{1+\gamma}$ $\frac{\gamma n 2^h}{1+\gamma}$	32/3 16/3	48 16	512/3 128/3

Figure 2.2: summary of calculations of the logical effort of logic gates



Calculating parasitic delay of logic gates is not as easy as calculating logical effort. The principal contribution to the parasitic capacitance is the capacitance of the diffused regions of transistors connected to the output signal [2]. Detailed description can be found in [2]. Here we only give the equation for calculating parasitic delay and the factors attached to it.

$$p = \frac{\Sigma W_d}{1 + \gamma} \cdot P_{inv} \quad (2.6)$$

In equation (2.6),  $W_d$  is the width of transistors connected to the logic gate's output. For this estimate to apply, we assume that transistor layouts in the logic gates are similar to those in the inverter. Note that this estimate ignores other stray capacitances in a logic gate, such as contributions from wiring and from diffused regions that lie between transistors that are connected in series. Figure 2.3 shows a summary of parasitic delay of logic gates used in following simulation.

Gate type	Formula	Parasitic delay when $p_{inv} = 1.0$			
		$n = 1$	$n = 2$	$n = 3$	$n = 4$
inverter	$p_{inv}$	1			
NAND	$np_{inv}$		2	3	4
NOR	$np_{inv}$		2	3	4
multiplexer	$2np_{inv}$		4	6	8
XOR, XNOR, parity	$n2^{n-1}p_{inv}$		4	12	
majority	$6p_{inv}$			6	
C-element	$np_{inv}$		2	3	4
latch	$2p_{inv}$	2			

Figure 2.3: Estimate of the parasitic delay of logic gates.

### 2.2.2 Logical effort of multi stages

The method of logical effort is applied in two ways to design fast multi-stage logic networks. It reveals the best number of stages to use in the network and it shows

how to get least overall delay by balancing the delay among the stages [2]. In our modules, we use optimized multi-stage logical effort to analyze complicated modules such as multipliers and different kinds of dividers with fixed number of stages.

We define the path effort with an upper-case symbol  $F$ , which can be obtained by combining path logical effort, path branch effort and path electrical effort. This is shown in equation (2.7).

$$F = G \cdot B \cdot H \quad (2.7)$$

The logical effort along a path compounds by multiplying the logical efforts of all the logic gates along the path as shown in equation (2.8). Upper-case symbol  $G$  denotes the path logical effort, and  $g$  denotes the logical effort of single stage.

$$G = \prod g_i \quad (2.8)$$

The electrical effort along a path through network is simply the ratio of the capacitance that loads the last logic gate in the path to the input capacitance of the first gate in the path. An upper-case symbol  $H$  is used to indicate the electrical effort along a path. This is shown in equation (2.9).

$$H = \frac{C_{out}}{C_{in}} \quad (2.9)$$

The branching effort along an entire path,  $B$ , is the multiplication of the branching effort at each of the stages along the path.

$$B = \prod b_i \quad (2.10)$$

While the single stage branching effort is defined as below :

$$b = \frac{C_{on-path} + C_{off-path}}{C_{on-path}} \quad (2.11)$$

Where  $C_{on-path}$  is the load capacitance along the path we are analyzing and  $C_{off-path}$  is the capacitance of connections that lead off the path. If the path has no branch, this value  $b$  is set to one.

The path delay,  $D$ , is the sum of the delays of each of the stages of logic in the path.

$$D = \sum D_i = D_F + P \quad (2.12)$$

Where the path effort delay  $D_F$  is

$$D_F = \sum g_i \cdot h_i \quad (2.13)$$

And the path parasitic delay is

$$P = \sum p_i \quad (2.14)$$

Optimizing the design of an N-stage logic network, the minimum delay is achieved when the stage effort is :

$$f = g \cdot h = F^{\frac{1}{N}} \quad (2.15)$$

The reason is demonstrated in [2]. We simply explain here: the path delay is the least when each stage in the path bears the same stage effort. Therefore, the minimum logical effort delay along a path can be expressed as :

$$D = F^{\frac{1}{N}} + P \quad (2.16)$$

In chapter 3 and chapter 4, we mainly use the equations and theory described above to analyze the module delay and compare it with the simulation results. Our results approve that logical effort can be used for cursory estimate modules' delay with number of input bits and number of stages.

## 2.3 Simulation Tools

In the implementation part, several modules such as carry save adders (CSA), carry propagate adders (CPA), Radix 4 multiplier, Radix 2, 4 and 8 dividers are implemented with VHDL and estimated with software tools Spice and Synopsys. Standard blocks that can be used for future exploration are estimated, and nonstandard blocks are estimated seperately.

Modules are simulated with VHDL based on a standard CMOS cell library with feature of 90nm. SPICE and SYNOPSYS are also put in use in the early phases for components' and complicate modules' delay and area analysis. SYNOPSYS is a well known software tool for integrated circuit design compiling; it also provides logic synthesis; timing, area and power analysis; Verilog and VHDL simulation. SPICE (Simulation Program with Integrated Circuit Emphasis) is a general purpose analog electronic circuit simulator. It is a powerful program that is used in IC and board-level design to check the integrity of circuit designs and to predict circuit behavior. In addition, both of these two softwares are on the bases of same cell library from company UNICAD. Logical effort are simulated and compared with SPICE and SYNOPSYS; however, these tools are using much more complicate arithmetics when analyzing performances.

# Division Algorithms Simulation

---

In this chapter, we first introduce transition time and its simulation method in part 1, then, in part 2, we describe FO4 module and some basic components with delay measured in terms of FO4; finally, some complicate modules like multiplier and divider are presented.

## 3.1 Propagation Delay and Transition Time

Since the principle of logical effort utilizes propagation delays to compare designs implementing the same logical statement. We will first introduce propagation delay and transition time.

Propagation delay is the measuring method that we use in this project. Propagation delay, which is also called gate delay, for single gate, is the length of time starting from when the input to a logic gate becomes stable and valid, to the time that the output of that logic gate is stable and valid; for multiple stages, is the length of time it takes for a signal to travel to its destination. In our simulation, we refer gate properties presented in [7]. In [7], for each gate, a propagation delay

calculation method is presented. Take inverter as an example, figure 3.1 is cited from [7]. It shows a propagation delay measuring equation. If we want to use this equation for delay measurement, two parameters are needed:  $C$  means output capacitance, which is already given in [7], another parameter,  $Tr$ , which is called transition time, is the one that we are going to obtain from our simulation.

Cell	Path	Event	Best 1.1V - 40C	Nominal 1V 25C	Worst 0.9V 125C
IVSVTX1	A-Z	A_Z (fall)	$-0.010 + 0.288 \cdot Tr + 1.448^\circ C$	$-0.022 + 0.338 \cdot Tr + 2.301^\circ C$	$-0.024 + 0.370 \cdot Tr + 3.427^\circ C$
IVSVTX1	A-Z	A_Z (rise)	$-0.008 + 0.364 \cdot Tr + 1.817^\circ C$	$-0.019 + 0.371 \cdot Tr + 2.832^\circ C$	$-0.021 + 0.379 \cdot Tr + 4.154^\circ C$

Figure 3.1: propagation delay of inverter as a function of  $C$  and  $Tr$

when responding to a stable input signal, in a logic circuit (a discrete-time dynamical system whose state is representable as a boolean-valued vector function of time) undergoing a change of state, it identifies the rise time or the fall time of the output voltage.[3]  $Tr$  means the transition time low-in-high;  $Tf$  means the transition time high-in-low.

Transition time can be tested using any logic gates. However, for a single gate, fall time and rise time are not stable. More logic gates we use, this simulation value will be closer to the true value. In our simulation, a 10-inverter-chain and a 15-inverter-chain are used to simulate the input transition fall time and input transition rise time. Figures are shown in 3.2. For this module, the last inverter is connected directly to ground, which means the  $C_{load}$  is empty.

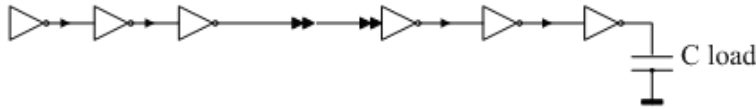


Figure 3.2: 10 or 15 inverter chain

For using SPICE, transition rise time and fall time should be set as input parameter before simulation. Before we know the normal scope of the input transition

time for 90nm library, some huge values are set to these simulation modules; since these long inverter chain can neutralize the error and by testing middle and last inverters, the range of the transition time can be detected. We cut the wave time when the voltage value is between 10 and 90 to neutralize the inaccuracy. In the following simulation, the seconde, 4th, 6th, ... even number inverters are measured to get the average transition fall time.

the $n^{th}$ inverter	90% of fall time	10% of fall time	Subtraction value
$4^{th}$	$2.71e^{-10}$	$2.53e^{-10}$	$0.18e^{-10}$
$6^{th}$	$3.00e^{-10}$	$2.83e^{-10}$	$0.17e^{-10}$
$8^{th}$	$3.29e^{-10}$	$3.12e^{-10}$	$0.17e^{-10}$
$10^{th}$	$3.60e^{-10}$	$3.41e^{-10}$	$0.19e^{-10}$

Table 3.1: 10 inverter chain with transition time set to 100ps.(u:s)

$$T_f = \frac{0.18 + 0.17 + 1.17 + 0.19}{4 \cdot 80\%} = 22.5ps \quad (3.1)$$

Table 3.1 shows a simulation with 10 inverter chain. The original transition rise and fall time are set to 100ps, transition fall time is obtain from equation (3.1).

the $n^{th}$ inverter	90% of fall time	10% of fall time	Subtraction value
$4^{th}$	$2.71e^{-10}$	$2.53e^{-10}$	$0.18e^{-10}$
$6^{th}$	$3.00e^{-10}$	$2.83e^{-10}$	$0.17e^{-10}$
$8^{th}$	$3.29e^{-10}$	$3.12e^{-10}$	$0.17e^{-10}$
$10^{th}$	$3.59e^{-10}$	$3.41e^{-10}$	$0.18e^{-10}$
$12^{th}$	$3.88e^{-10}$	$3.70e^{-10}$	$0.18e^{-10}$
$14^{th}$	$4.17e^{-10}$	$4.00e^{-10}$	$0.17e^{-10}$

Table 3.2: 15 inverter chain with transition time set to 200ps.(u:s)

$$T_f = \frac{0.18 + 0.17 + 1.17 + 0.18 + 0.18 + 0.17}{6 \cdot 80\%} = 23.125ps \quad (3.2)$$

Table 3.2 shows simulation with 15 inverter chain. The original transition rise

and fall time are set to 200ps, transition fall time is obtained from equation (3.2).

the $n^{th}$ inverter	90% of fall time	10% of fall time	Subtraction value
$4^{th}$	$2.12e^{-10}$	$1.95e^{-10}$	$0.17e^{-10}$
$6^{th}$	$2.42e^{-10}$	$2.24e^{-10}$	$0.18e^{-10}$
$8^{th}$	$2.71e^{-10}$	$2.53e^{-10}$	$0.18e^{-10}$
$10^{th}$	$3.00e^{-10}$	$2.83e^{-10}$	$0.17e^{-10}$
$12^{th}$	$3.30e^{-10}$	$3.12e^{-10}$	$0.18e^{-10}$
$14^{th}$	$3.59e^{-10}$	$3.41e^{-10}$	$0.18e^{-10}$

Table 3.3: 15 inverter chain with transition time set to 100ps.(u:s)

$$T_f = \frac{0.17 + 0.18 + 1.18 + 0.17 + 0.18 + 0.18}{6 \cdot 80\%} = 23.125ps \quad (3.3)$$

Table 3.3 shows simulation with 15 inverter chain. The original transition rise and fall time are set to 100ps, transition fall time is obtained from equation (3.3).

From the above three modules, we know that the transition time is around 20ps. In the next step, we set the transition time to 23.125ps for a single inverter with increasing fan out load, that is  $C_{load}$ , the goal of this simulation is to verify that transition time is not affected by the output load increasing.

Load	propagation delay SPICE	logical effort calculation	error (ns)
0.083 pf	0.157	0.1768	0.02
0.060 pf	0.116	0.1238	0.01
0.040 pf	0.081	0.0778	0.01
0.020 pf	0.045	0.0318	0.01

Table 3.4: Propagation delay comparison between SPICE and equation calculation.(u.ns)

Table 3.4 shows the single inverter propagation delay comparison between SPICE and equation calculation from figure 3.1. Inverter is added output load of 83fF, 60fF, 40fF and 20fF. We can see that all of the errors are with 2%, which is pretty



good results. It means that the transition time we get from simulation is quite close to the value that should be used in the data book.

In the following step, we set the input transition time to 25ps in order to get the precise transition time value through a 15 inverter chain.

```

90nm; 15 inverter-chain; load=0.083pF; Tf=25ps; Tr=25ps

***** transient analysis                      tnom= 25.000 temp= 25.000
*****
tf4= 1.7229E-11 targ= 1.6290E-10 trig= 1.4567E-10
tr5= 2.5359E-11 targ= 1.8585E-10 trig= 1.6049E-10
tf6= 1.7491E-11 targ= 1.9246E-10 trig= 1.7497E-10
tr7= 2.5258E-11 targ= 2.1512E-10 trig= 1.8987E-10
tf8= 1.7608E-11 targ= 2.2198E-10 trig= 2.0437E-10
tr9= 2.5175E-11 targ= 2.4446E-10 trig= 2.1928E-10
tf10= 1.7962E-11 targ= 2.5149E-10 trig= 2.3352E-10
tr11= 2.5205E-11 targ= 2.7371E-10 trig= 2.4850E-10
tf12= 1.7419E-11 targ= 2.8019E-10 trig= 2.6277E-10
tr13= 2.5334E-11 targ= 3.0301E-10 trig= 2.7768E-10
tf14= 1.7536E-11 targ= 3.0970E-10 trig= 2.9217E-10
tr15= 2.1009E-11 targ= 3.2820E-10 trig= 3.0719E-10
tfall= 2.1926E-11
trise= 3.0696E-11

result : tr = 30.696 ps
         tf = 21.926 ps

```

Figure 3.3: Transition rise and fall time from SPICE

Figure 3.3 shows the SPICE data from a 15-inverter-chain simulation. The original transition time is set to 25ps. As the data shows, in the following step, transition rise time will be set to 30.696ps, and transition fall time will be set to 21.926ps.

### 3.2 Delay Analysis for SPICE Modeling and Logical Effort

Once the transition rise and fall time are obtained from simulation, we can not analyze delay for simple components using propagation delay equation referenced from [7]. These propagation delays will be compared with logical effort and simulation results. As we can see from Chapter 2 that, logical effort is only related with output load capacitance, which is identical with propagation delay equation in [7]. (Transition time is an internal parameter of the simulation tool, it is not related with delay directly) Since our goal is to approve that logical effort can be used for modules delay modeling for general situation, even though the module delay calculation methods inside logical effort and simulation tool are not identical, if logical effort is changing the same way as propagation delay, we can say that logical effort can be used for general modules delay calculation.

#### 3.2.1 Component block introduction

First, we begin SPICE modeling from simple components as shown in figure 3.4 below.

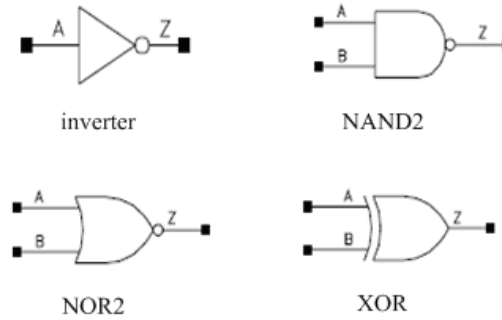


Figure 3.4: Single gate model

Figure 3.4 contains all the single gates that we use in this part's modeling. The

reason we choose inverter, NAND2 gate, NOR2 gate and XOR gate is because these four components are the basic components of CMOS circuit; any other gate can be gained from combining these basic gates together, for example, AND2 gate can be composed with a NAND2 gate and an inverter gate. However, in real CMOS circuit production, instead of combining several gates together, more complicate components have their own circuit construction to archive better delay. More complicate components such as  $[3 : 1]$  multiplexer and half adder will be introduced and used in next step. Inverter is commonly used as the gauge of other components. In our project, the logical effort will be used as module delay calculation method, which must be independent with all cell library data, with inverter delay as standard unit, our calculation can be completely applied to any other cell library standard.

In the next step, we introduce two adder modules, the  $[3 : 2]$  adder and  $[4 : 2]$  adder. (in form  $[a : b]$ ,  $a$  denotes input operators number,  $b$  denotes output results)  $[3 : 2]$  adder is also called full adder, it has two input operators,  $a$ ,  $b$ , and a carry bit,  $c$ ,  $s$  and  $C_{out}$  are two output bits,  $C_{out}$  is the carry that can be used as propagate carry bit in next stage. They are shown in figure 3.5 and 3.7 below.

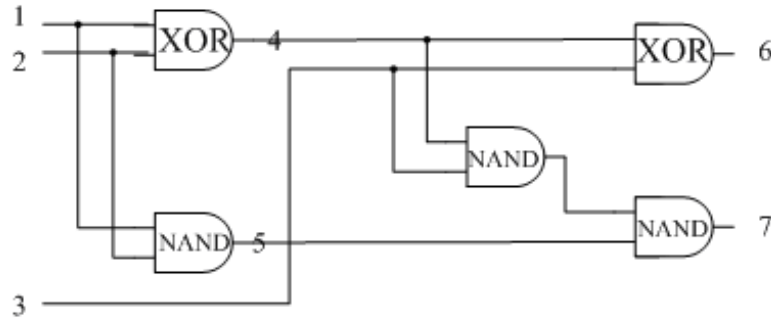


Figure 3.5:  $[3 : 2]$  full adder

Figure 3.5 shows a  $[3 : 2]$  full adder, it is composed by two half adders and a NAND2 gate. A half adder is composed with a XOR and a NAND2 gate. This adder is frequently used in our later modules as a carry save adder. 1, 2, ... 7 denotes the port number used in SPICE simulation. The critical path is shown

below.

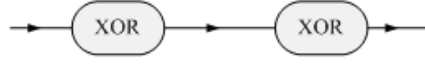


Figure 3.6: Critical path of one bit  $[3 : 2]$  adder

Figure 3.7 shows structure of  $[4 : 2]$  adder. It has four input operator bits and two output bits, one of which is the input of the next stage. Its critical path is shown in figure 3.8. Comparing with  $[3 : 2]$  adder, it has more operators but worse delay. In the next discussion, we are using  $[3 : 2]$  adder to construct carry save adder for dividers; however, this  $[4 : 2]$  module will be used in the future to replace  $[3 : 2]$  adder.

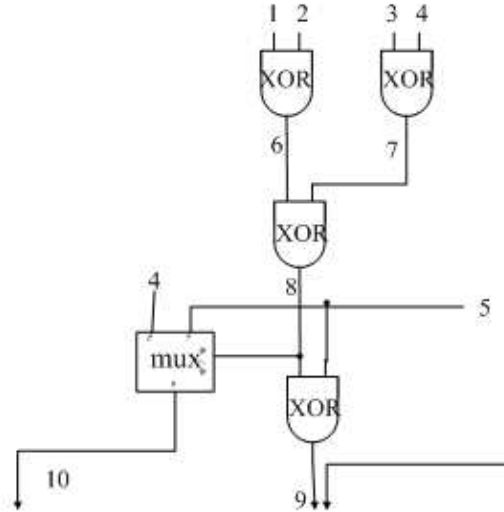


Figure 3.7:  $[4 : 2]$  adder

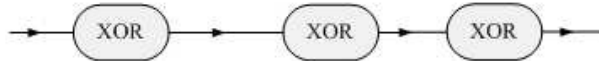
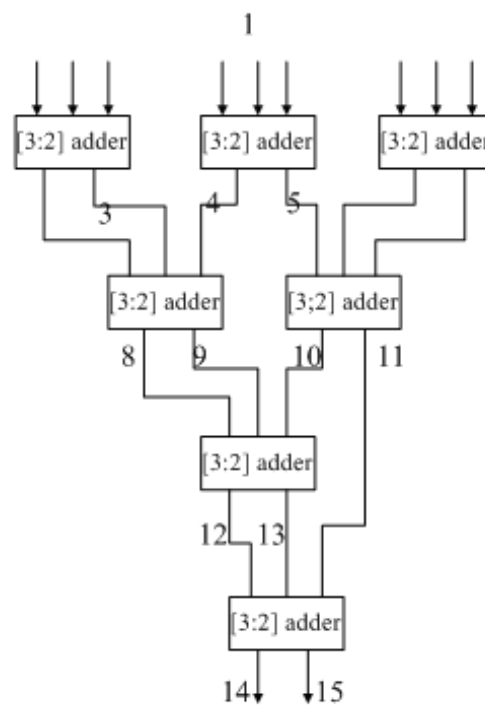


Figure 3.8: Critical path of  $[4 : 2]$  adder

Another multiple-operators adder is also involved in this modeling. Figure 3.9 shows an  $[9:2]$  adder tree.

Figure 3.9:  $[9:2]$  adder tree

### 3.2.2 Models simulation with constant output load

We can remember in Chapter 2, we introduce the logical effort method to calculate module delay, especially with equations from (2.1) to (2.6), we can get this delay with concrete properties. Capacitance useful is given in table 3.5 below. Data in table 3.5 is cited from [7].

Gate	Capacitance (u: pF)
Inverter	0.0021
Nand2	0.0021
NOR2	0.0020
XOR2	0.00505
MUX2	0.0023

Table 3.5: Capacitance for some useful gates

Now with the input transition time values obtained in last part, we get below tables including logical effort, Spice simulation result and calculation value with reference of [7]. Along with these tables, relevant charts are obtained. (Capacitances of these components below are read from [7])

Module	Logical effort	SPICE simulation	Error
Inverter	$5\tau$	$2.8394e^{-11}$	0%
Nand2	$6\tau$	$3.2483e^{-11}$	4%
Nor2	$8\tau$	$4.1660e^{-11}$	4%
Xor	$8\tau$	$5.6000e^{-11}$	4%
[3:2] adder	$15.42857\tau$	$9.9098e^{-11}$	11%
[4:2] adder	$21.80952\tau$	$11.878e^{-11}$	19%

Table 3.6: Comparison with logical effort and spice simulation for some components

Table 3.6 shows a comparison with logical effort and SPICE simulation result for components we referred above. For all the modules, output capacitance is assumed to FO4 inverter; that is  $8fF$ . ( $C_{inv} = 2fF$ ) In the logical effort column,

$\tau$  denotes one inverter delay as a standard unit. For the simulation column, standard unit is second. The error column is obtained as we consider the FO4 inverter has no inaccuracy.

When analyzing these data, we find that, for single gate components, the error scope is within 5%, which is acceptable, while for the  $[3 : 2]$  adder and the  $[4 : 2]$  adder, the error rate varies. It reveals that with the module block complication increasing, the inaccuracy grows bigger.

### 3.2.3 Model simulation with variable output load

Table 3.6 is obtained under the situation that the output load capacitance is stable. The following simulation is done with variable output load. The goal is to compare the variation rate of logical effort, propagation delay and simulation data. We only discuss  $[3 : 2]$  and  $[4 : 2]$  adders, since other single gate deviation is acceptable.

$[3:2]$ adder	Logical effort	SPICE simulation	Propagation delay
Load=2fF	$12.42857\tau$	$8.6675e^{-11}$	$12.5696e^{-11}$
Load=4fF	$13.42857\tau$	$9.0946e^{-11}$	$12.8903e^{-11}$
Load=6fF	$14.42857\tau$	$9.5084e^{-11}$	$13.2110e^{-11}$
Load=8fF	$15.42857\tau$	$9.9098e^{-11}$	$13.5317e^{-11}$

Table 3.7: Delay comparison for  $[3 : 2]$  adder

$[4:2]$ adder	Logical effort	SPICE simulation	Propagation delay
Load=2fF	$18.80952\tau$	$10.368e^{-11}$	$18.5697e^{-11}$
Load=4fF	$19.80952\tau$	$10.892e^{-11}$	$18.8905e^{-11}$
Load=6fF	$20.80952\tau$	$11.393e^{-11}$	$19.6722e^{-11}$
Load=8fF	$21.80952\tau$	$11.878e^{-11}$	$19.9929e^{-11}$

Table 3.8: Delay comparison for  $[4:2]$  adder

In table 3.7 and 3.8, we got logical effort, SPICE simulation data and propagation

delay referred from [7]. The output load capacitance is varied from 2fF to 8fF. With these two tables, we got charts in figure 3.10 and figure 3.11 below.

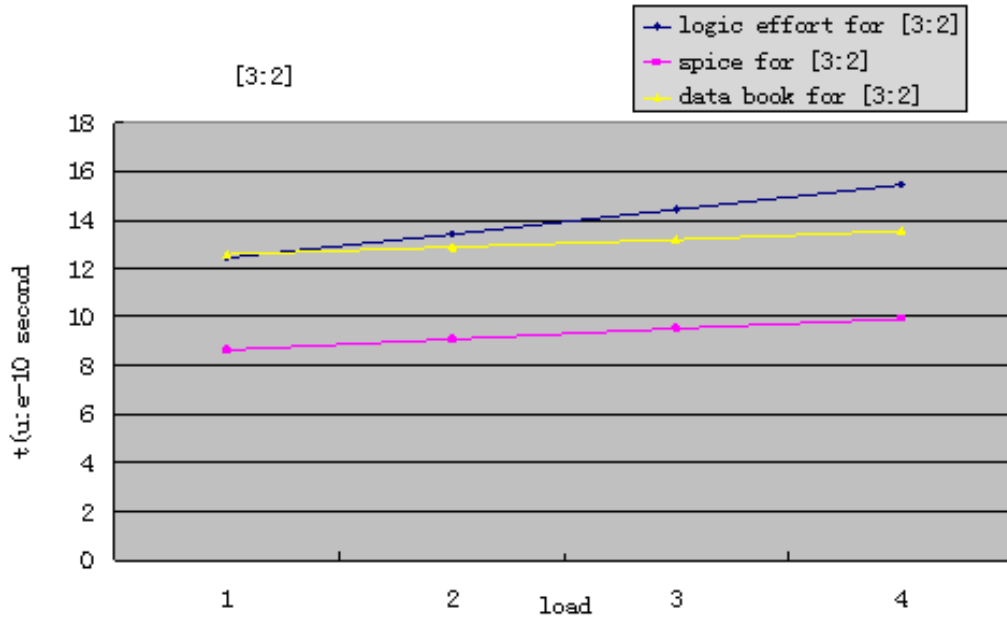


Figure 3.10: comparison chart for [3 : 2] adder

From the above figures, we can see that the SPICE line and the data book line are almost parallel with a stable difference. The logical effort line is much closer to and has a small stable difference with the data book line, but the slope is a tiny different, which means with the load increasing, the difference is growing as well.

From above simulations, we can get the conclusion that logical effort is suitable for general basic components delay analyzing.



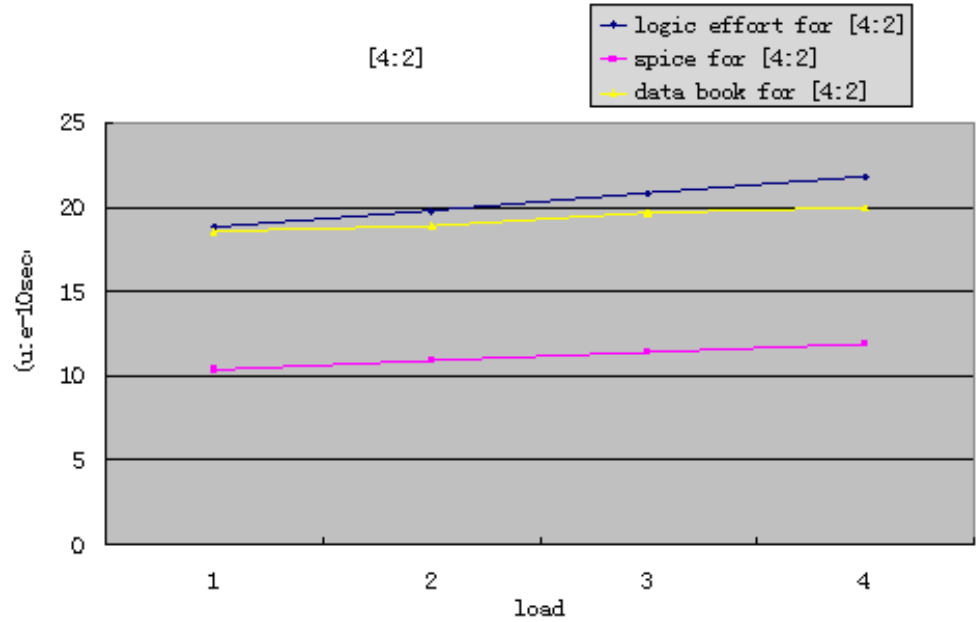


Figure 3.11: comparison chart for  $[4 : 2]$  adder

### 3.3 Dividers

#### 3.3.1 Algorithm of division

In this part, we apply logical effort on more modules with complex functions based on components and gates we discussed above. All the modeling and simulations in this part are done by Synopsys. In last part, we discussed about different adders with SPICE simulation data. However, there are differences between SPICE and Synopsys tools, which make the simulation data inaccuracy with each other. To avoid this, we start this modeling with re-doing these adders with Synopsys. Details will be discussed later. The division algorithms are mainly referenced from [1].

The division is done using the digit recurrence algorithm. Division operation is defined by the following expressions:

$$x = q \cdot d + rem \quad (3.4)$$

$$|rem| < |d| \cdot ulp \quad (3.5)$$

$$sign(rem) = sign(x) \quad (3.6)$$

Where the dividend  $x$  and the divisor  $d$  are the operands and the results are the quotient  $q$  and, optionally, the remainder  $rem$ . The unit in the last position ( $ulp$ ) defines the granularity of the quotient. The two most typical cases produce a fractional quotient or an integer quotient. Correspondingly, two types of division operation are defined: fractional division and integer division. In our project, we only discuss fractional division. Therefore, for fractional case, the divisor and dividend are required to be normalized to range as below:

$$\frac{1}{2} \leq x < d < 1 \quad (3.7)$$

The digit recurrence algorithm consists of  $n$  iterations of a recurrence, in which each iteration produces one digit of the quotient. This is preceded by an initialization step and followed by a termination step. The recurrence steps are done according to following equation:

$$q[j] = q[0] + q_j \cdot r e^{-j} \quad (3.8)$$

The quotient digit set plays a crucial role in the characteristics of the algorithm. The most direct choice is to use the canonical digit set such that  $0 \leq q_j \leq r - 1$ . This leads to the basic restoring division, which is not convenient because of an expensive quotient digit selection. Therefore, a redundant digit set is used to obtain a simpler selection function (the selection function will be discussed in Chapter 4). In particular, we use the symmetric signed-digit set of consecutive integers.

$$q_j = \{-a, -a + 1, \dots, -1, 0, 1, \dots, a - 1, a\} \quad (3.9)$$

Since for redundant representation, more than  $r$  consecutive integer values including 0 are needed,  $a$  has to satisfy (3.10).

$$a \geq \left\lceil \frac{r}{2} \right\rceil \quad (3.10)$$

The redundancy factor  $p$  is defined as

$$p = \frac{a}{r - 1} \quad \frac{1}{2} < p \leq 1 \quad (3.11)$$

The value of the redundancy factor influences the complexity of the quotient digit selection and of the generation of the divisor multiples in an opposite manner: a higher  $p$  reduces complexity of the selection function but increases complexity of the generation of the divisor multiples. Consequently, the choice of  $p$  is an important design decision.

From the definition, a correct division algorithm must produce a quotient  $q$  with a positive error of less than one  $ulp$ , which respect to the infinite precision value. More detail will be discussed in [1]. Here we only give the recurrence.

$$W[j + 1] = r \cdot W[j] + d \cdot q[j + 1] \quad (3.12)$$

Equation (3.12) is the basic recurrence on which the division algorithms are based. To follow the equation, the initial value can be obtained:

$$W[0] = r \cdot \left\lfloor \frac{x}{r} \right\rfloor + d \cdot q[0] \quad (3.13)$$

The equations above are accomplished by selecting a suitable value of  $q_{j+1}$  by means of the quotient digit selection function. Actually, the use of a redundant digit set for  $q_j$  allows that the selection function uses a truncated  $r \cdot w[j]$ , and a truncated  $\hat{d}$  as below:

$$q[j + 1] = SEL(\widehat{r \cdot w[j]}, \hat{d}) \quad (3.14)$$

1. one digit arithmetic left shift of  $w[j]$  to produce  $r \cdot w[j]$
2. determination of the quotient digit  $q_{j+1}$  by the quotient digit selection function
3. generation of the divisor multiple  $d \cdot q_{j+1}$

4. subtraction of  $d \cdot q_{j+1}$  from  $r \cdot w[j]$
5. update of the quotient  $q[j]$  to  $q[j + 1]$  by the on-the-fly conversion.

Above five steps give sub-computations for each iteration of the recurrences [1]. This general description of the recurrence step can result in different specific versions depending on several interrelated factors. We will discuss some dividers in next part, radix 2, radix 4 and radix 8 divider. For the same quotient precision, the number of iterations of the algorithm is reduced by  $r$  when  $r$  increasing. However, this increase in radix produces a more complex implementation because of the quotient digit selection and the generation of the divisor multiples. This additional complexity increases the time of each iteration [1].

$[3 : 2]$  and  $[4 : 2]$  redundant carry save adders are used in the following division scheme. These two adders are simulated following the structures we discussed in figure 3.5 and figure 3.7. Delay and area consumption will be analyzed together with dividers and multiplier.

### 3.3.2 Radix 2 divider implementation

We start divider discussion with the simplest radix 2 divider. the module is cited from [1]. The algorithm is summarized in figure 3.12.

Figure 3.12 shows the radix 2 division scheme implementation. Input operators are dividend and divisor, output result is  $q$  from the on-the-fly converter component. Four bits out of  $WS$  and  $WC$  are used for quotient digit selection in  $q_{SEL}$  component. The quotient digit is used in the next step as the selection bit of multiplexer. Since we use redundant carry save adder, registers  $WS$  and  $WC$  are needed to store two output operators. We have discussed about the  $[3 : 2]$  carry save adder in the last section. The  $[3 : 1]$  multiplexer is as figure 3.13.

Three input operators are  $d$ , constant 0 and  $-d$ .  $q[j + 1]$  is input as the

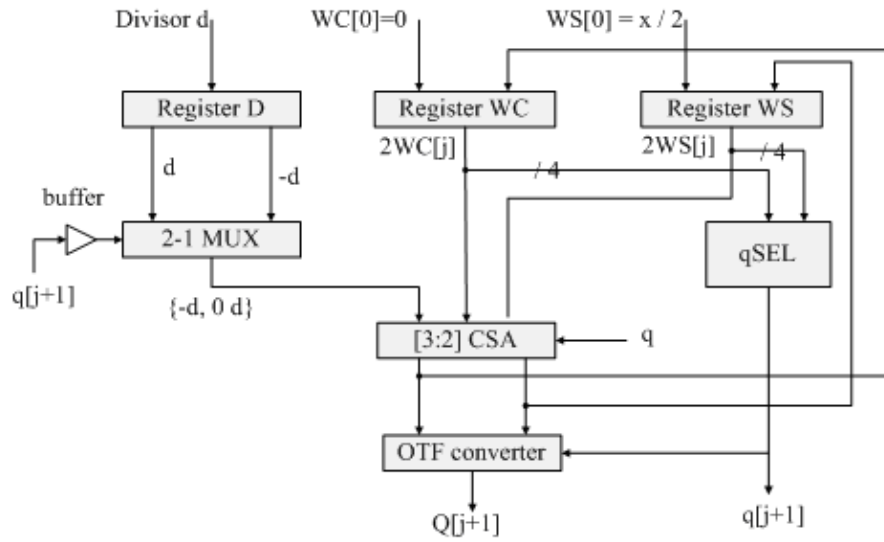


Figure 3.12: Implementation of radix 2 scheme.

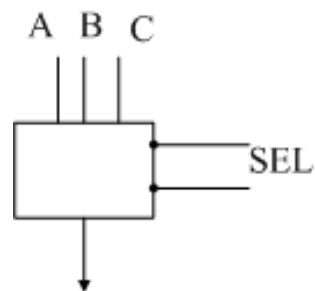


Figure 3.13: 3-1 multiplexer.

selection bits. Here we simply describe the quotient digits of radix 2. Details will be discussed in Chapter 4. Since there are three input operators, two selection bits are needed. In radix 2 division,  $q[j]$  from the selection function is indicated with 2 bits.  $\{10, 00, 01\}$  is used to represent  $\{-d, 0, d\}$ . Multiplexer's output is connected with one of carry save adder's input operator. As referred in [7],  $[3 : 1]$  multiplexer is considered as a single component for evaluating delay and area. However, when analyzing with Synopsys, this multiplexer is composed with several basic components such as XOR gates and NAND gates. Even though the function is identical, the delay of the later scheme is much worse than the former scheme. This will cause a bigger error between the logical effort calculation and simulation evaluation. Since this  $[3 : 1]$  multiplexer is possible to be implemented with CMOS circuit, we choose to consider it as an integrated component but to be composed from other basic gates.

In this section, the selection function is replaced with a 2-inverter chain to avoid complicating the selection structure. The main body of the selection function will be discussed later.

CMOS buffer circuit is a cascade of two CMOS inverters.

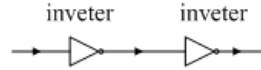


Figure 3.14: buffer

Figure 3.15 shows the structure of on-the-fly conversion scheme. The quotient digit has to be converted from signed-digit representation to conventional representation. We now discuss an algorithm that performs the conversion in a digit serial manner as the digits of the quotient are produced. For all the division we discussed below, this scheme is used for quotient digit converting. Equation (3.15) shows an algorithm.

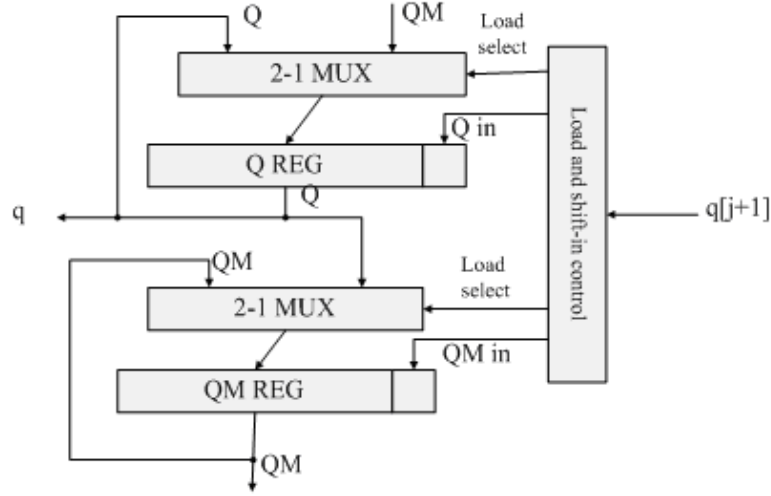


Figure 3.15: on-the-fly converter structure

$$Q[j] = \sum_{i=1}^j q_i \cdot r^{-i} \quad (3.15)$$

$Q[j]$  is the digit vector of the converted quotient consisting of the  $j$  most significant digits. From (3.15) we can get that:

$$Q[j+1] = Q[j] + q_{j+1} \cdot r^{-(j+1)} \quad (3.16)$$

Since  $q_{j+1}$  can be negative, we can use the algorithm on [1].

$$Q[j+1] = \begin{cases} Q[j] + q_{j+1} \cdot r^{-(j+1)} & \text{if } q_{j+1} \geq 0 \\ Q[j] - r^{-j} + (r - |q_{j+1}| \cdot r^{-(j+1)}) & \text{if } q_{j+1} < 0 \end{cases} \quad (3.17)$$

This algorithm has a big disadvantage that the subtraction requires the propagation of a borrow and , therefor, is slow. [1] gives another form  $QM[j+1]$  to avoid this.



$$QM[j] = Q[j] - r^{-j} \quad (3.18)$$

With this  $QM$  form,  $Q[j + 1]$  can be obtained from equation in [1].

$$Q[j + 1] = \begin{cases} (Q[j], q_{j+1}) & \text{if } q_{j+1} \geq 0 \\ (QM[j], (r - |q_{j+1}|)) & \text{if } q_{j+1} < 0 \end{cases} \quad (3.19)$$

$$QM[j + 1] = \begin{cases} (Q[j], q_{j+1} - 1) & \text{if } q_{j+1} > 0 \\ (QM[j], (r - 1 - |q_{j+1}|)) & \text{if } q_{j+1} \leq 0 \end{cases} \quad (3.20)$$

With equation (3.19) and (3.20), now we can apply this to the on-the-fly converter architecture in figure 3.14. the  $Q[0]$  and  $QM[0]$  are initialized to 0.

In figure 3.15, the outputs of multiplexers are shifted 1 bit with  $Q$  register and  $QM$  register. The control component on the right is used for load select bit,  $Q_{in}$  and  $QM_{in}$ . Specific schemes for those signals are presented in equation (3.21) and (3.22).

$$Q \leftarrow \begin{cases} \text{shift } Q \text{ with insert } (Q_{in}) & \text{if } C_{shiftQ} = 1 \\ \text{shift } QM \text{ with insert } (Q_{in}) & \text{if } C_{loadQ} = 1 \end{cases} \quad (3.21)$$

$$QM \leftarrow \begin{cases} \text{shift } QM \text{ with insert } (QM_{in}) & \text{if } C_{shiftQM} = 1 \\ \text{shift } Q \text{ with insert } (QM_{in}) & \text{if } C_{loadQM} = 1 \end{cases} \quad (3.22)$$

Figure 3.16 shows the critical path for this radix 2 division scheme. Our logical effort is calculated according to this critical path as well. When calculating delay, we do not consider the delay, area and load of interconnections. From figure 3.15, we can see that not all of the carry save adders are on the critical path but only half adder. However, the selection function box is on the critical path, which should be seriously considered to decrease delay.



Figure 3.16: critical path of radix 2 division scheme

### 3.3.3 Radix 4 divider implementation

The radix 4 algorithm with the residuals in carry save form is similar to the radix 2 algorithm. [1] presents some differences between those two algorithms.

1. We consider the case where the quotient-digit set is  $\{-2, -1, 0, 1, 2\}$ . This is a redundant digit set and allows a simple implementation of  $q_{j+1} \cdot d$ .
2. For this case ( $p < 1$ ) we initialize  $WS[0]$  with  $\frac{x}{4}$
3. The quotient digit selection has as arguments the truncated carry save shifted residual  $\hat{y}$  and the truncated  $\hat{d}$ . This is contrast to the radix 2 scheme, in which the selection function is independent with the divisor. 7 bits of truncated  $\hat{y}$  and  $\hat{d}$  are used in this selection function.
4. Because of the initialization, the final quotient is produced by multiplying the obtained quotient by four.
5. The on-the-fly converter is changed to radix 4 scheme. While the architecture of this OTF is the same as radix 2 scheme.

Below is the criterion of radix 4 division scheme which is obtained according to the radix 2 algorithm of [2].

1.  $WS[0] \leftarrow \frac{x}{4}; WC[0] \leftarrow 0; Q[-1] = 0$
2. for  $j = 0 \dots n + 1$       $q_{j+1} \leftarrow SEL(\hat{y});$   
     $(WC[j + 1], WS[j + 1]) \leftarrow CSADD(4WC[j], 4WS[j], -q_{j+1} \cdot d);$   
     $Q[j] \leftarrow CONVERT(Q[j - 1], q_j)$  end for

3. if  $w[n+2] < 0$  then  $q = 4(\text{CONVERT}(Q[n+1], q_{n+2} - 1))$  else  
 $q = 4(\text{CONVERT}(Q[n+1], q_{n+2}))$

In above criterion,  $WS$  is sum,  $WC$  is the stored-carry bits, CSA is redundant carry save adder, and OTF is on-the-fly converter. Figure 3.17 gives the implementation of radix 4 case.

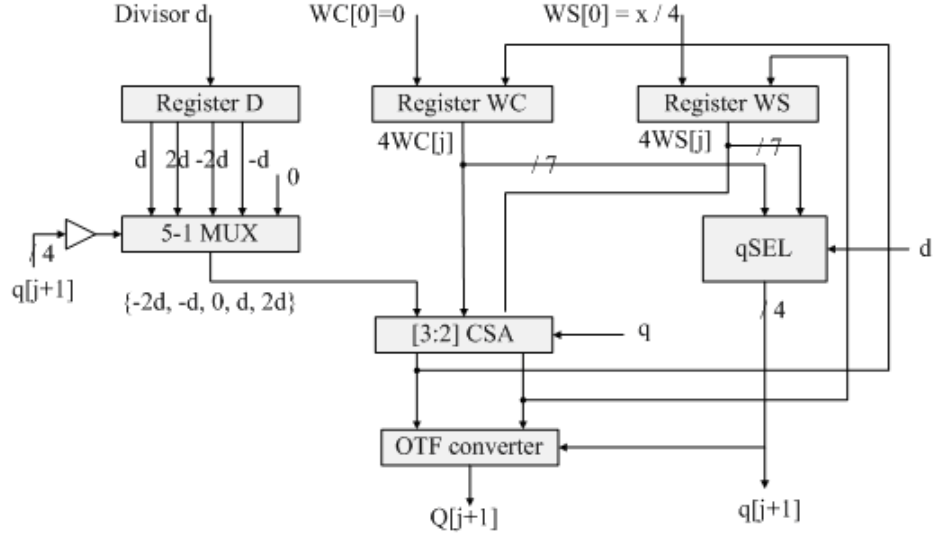


Figure 3.17: Implementation of radix 4 scheme.

Figure 3.18 shows an implementation of radix 4 scheme. Since the quotient digit set is  $\{-2, -1, 0, 1, 2\}$ , we used a 5-1 multiplexer to choose digits. Two  $[3:1]$  multiplexers are used to composed this  $[5:1]$  multiplexer.

Two schemes are used to implement the selection function. Using selection function with select table and comparison. We will discuss those schemes in Chapter 4. We use four bits to presents the quotient digit bits.  $\{-2, -1, 0, 1, 2\}$  are coded as digits set  $\{1000, 0100, 0000, 0001, 0010\}$

The on-the-fly converter is using structure of figure 3.14. The difference only exists in the shift register component. Since it is using  $\{-2 \dots 2\}$  digit set, which will be coded with 2 bits, the  $Q$  and  $QM$  will shift 2 bits in the shift

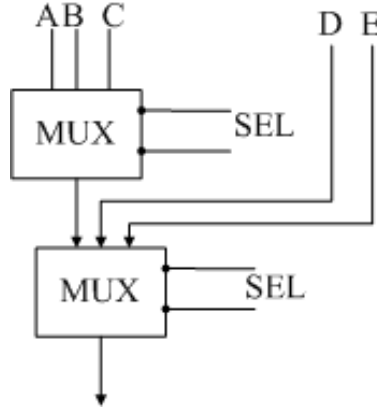


Figure 3.18: 5-1 multiplexer

register every criterion.

The critical path of radix 4 scheme is the same as radix 2 scheme.

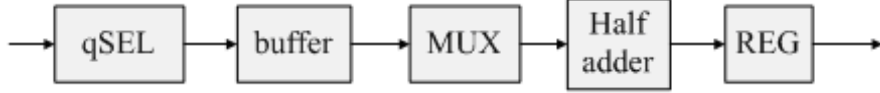


Figure 3.19: critical path of radix 4 scheme

### 3.3.4 Radix 8 divider implementation

For the radix 8 implementation, we follow the structures in figure 3.19.

Comparing with radix 4 scheme, radix 8 scheme uses double multiplexers and double carry save adders. The quotient digit set is in the range  $\{-7, \dots, 7\}$ . To simplify the generation of  $d \cdot q_{j+1}$ , it is decomposed into two components  $q_L$  and  $q_H$ .  $q_j$  is generated by adding  $q_L$  and  $q_H$  together.

$$q_{j+1} = q_{j+1}^L + q_{j+1}^H \quad (3.23)$$

Where  $q \cdot L_{j+1}$  is within  $\{-2, -1, 0, 1, 2\}$  and  $q_{j+1}^H$  is within  $\{-8, -4, 0, 4, 8\}$ . As

a consequence of this, the two carry save adders are used for  $qL$  and  $qH$  operations.

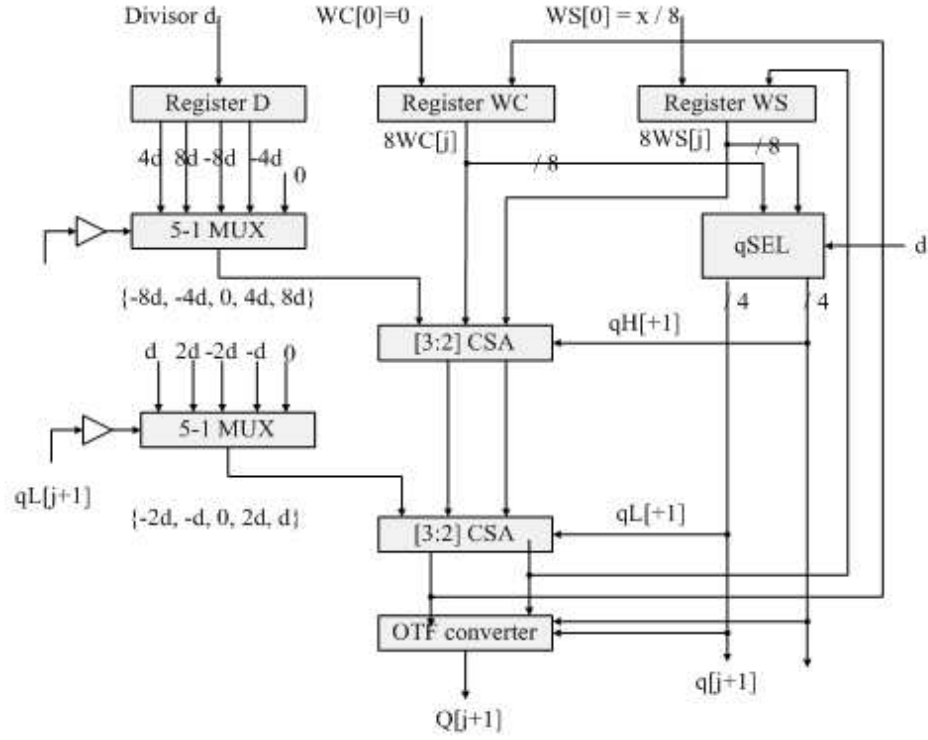


Figure 3.20: Implementation of radix 8 scheme.

The quotient digit selection depends on the truncated shifted residual, 8 bits, and the truncated divisor, 4 bits. Since the two components of  $q_{j+1}$  do not affect in the same way the critical path, the design of the selection function is done so as to minimize the critical path. In figure 3.20, part of the two carry save adder can work concurrently. Thus, only the second half adder of high bits adder and first half adder of low bits adder are on the critical path. Particular description can be seen from figure 3.21. In addition, Figure 3.21 shows the critical path of radix 8 division algorithm.

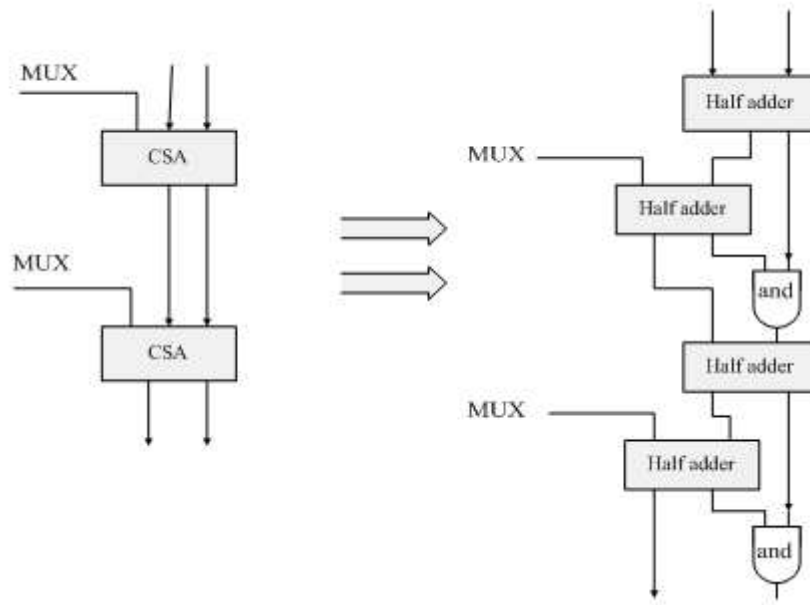


Figure 3.21: interconnections of two carry save adder.

### 3.4 Delay, area and power evaluation

In this part, we discuss the evaluation of delay and area for modules we presented above. Delay and area are simulated by Synopsys. For carry save adders, we use the modules shown in figure 3.22. Division modules are shown as figure 3.12, 3.17 and 3.20.

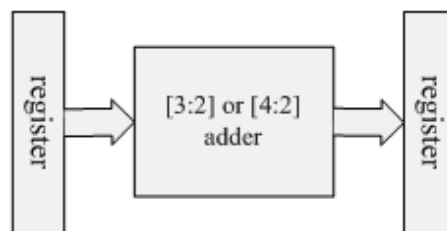


Figure 3.22: carry save adders module

### 3.4.1 Delay evaluation

Simulation delay is compared with logical effort. Since we have discussed the critical path for dividers and adders, we get

$$[3:2]\text{adder} : T_{\text{delay}} = T_{\text{xor}} + T_{\text{xor}}$$

$$[4:2]\text{adder} : T_{\text{delay}} = T_{\text{xor}} + T_{\text{xor}} + T_{\text{xor}}$$

Logical effort of Radix 2 divider :

$$T_{\text{delay}} = T_{\text{register}} + T_{\text{sel}} + T_{\text{buf}} + T_{\text{mux31}} + T_{\text{half-adder}}$$

Logical effort of radix 4 divider :

$$T_{\text{delay}} = T_{\text{register}} + T_{\text{sel}} + T_{\text{buf}} + T_{\text{mux31}} + T_{\text{mux31}} + T_{\text{half-adder}}$$

Logical effort of radix 8 divider :

$$T_{\text{delay}} = T_{\text{register}} + T_{\text{sel}} + T_{\text{buf}} + T_{\text{mux31}} + T_{\text{mux31}} + T_{\text{half-adder}} + T_{\text{half-adder}}$$

Logical effort is calculated according to equations we introduced in Chapter 2, the input load capacitance of each component is referred as the output load of last component. Table 3.1 gives some of the components' capacitance used in this simulation. All data is referred from [7].

Gate	Capacitance(u:pF)
<i>Inverter</i>	0.0021
<i>Nand2</i>	0.0021
<i>Xor</i>	0.00505
<i>Register</i>	0.0020
<i>Buffer</i>	0.0014
<i>[3 : 1]multiplexer</i>	0.0031

Table 3.9: gate capacitance

Table 3.9 gives the evaluation results and the error between logical effort and simulation data. Logical delay is calculated using equation (2.11) in terms of

FO4 inverter. Selection functions in those modules are ignored. One thing has to be noted is that when doing the simulation, Synopsys can simplify and optimize some blocks with other components instead of using components assigned. This causes the difference between the calculation results and simulation results. For example, when synthesizing the  $[3 : 1]$  multiplexer module, Synopsys will do an alternative components mapping, with gate level implementation using XOR and AND gates instead of using standard  $[3 : 1]$  multiplexer defined in the library. In consequence of this, the optimized simulation data is obviously better than calculation results. Another solution we can do is to change all our modules with gate level implementation which will be exactly identical with Synopsys synthesis. However, it is meaningless to do so, since Synopsys tool is not using RC mode as the same as logical effort.

module	logical delay(u:inverter)	Synopsys delay(u:ns)	error
<i>FO4</i>	6	0.06	0
$[3 : 2]adder$	17.1803	0.16	-7%
$[4 : 2]adder$	25.7365	0.27	+5%
$r - 2divider$	31.8315	0.31	-3%
$r - 4divider$	38.9294	0.40	+3%
$r - 8divider$	51.8634	0.52	+3%

Table 3.10: comparison between logical delay and Synopsys synthesis data

In table 3.10, we use FO4 inverter as an reference, the error is set to 0%.

According to the data, we can find that the error is all within 10%, which is quite acceptable. In the next step, we will particularly discuss the error inside each module. The adder modules are ignored since they are simpler comparing with other division modules.

Table 3.11 and above list shows the radix 2 divider internal timing reports and comparison. We can find that the register delay error is much bigger than other components. That is because, logical effort can only give a general description of register delay, however, the Synopsys synthesis is much more complicate and it maps using specific flip-flops with difference respective delay. For the select



Point	Trans	Incr	Path
clock CLK_0 (rise edge)		0.00	0.00
clock network delay (ideal)		0.00	0.00
reg2/DATA_OUT_reg[5]/CP (FD2QSVTX2)	0.00	0.00	0.00 r
reg2/DATA_OUT_reg[5]/Q (FD2QSVTX2)	0.04	0.11	0.11 f
reg2/DATA_OUT[5] (REG_n15_0)		0.00	0.11 f
sele/Y[0] (Radix2Selector)		0.00	0.11 f
sele/U27/B (NR2SVTX4)	0.04	0.00	0.11 f
sele/U27/Z (NR2SVTX4)	0.03	0.03	0.13 r
sele/U21/A (ND2SVTX4)	0.03	0.00	0.13 r
sele/U21/Z (ND2SVTX4)	0.02	0.02	0.15 f
sele/U25/A (IVSVTX4)	0.02	0.00	0.15 f
sele/U25/Z (IVSVTX4)	0.02	0.02	0.17 r
sele/U26/C (ND3SVTX8)	0.02	0.00	0.17 r
sele/U26/Z (ND3SVTX8)	0.03	0.02	0.19 f
sele/Z[0] (Radix2Selector)		0.00	0.19 f
buf0/input[0] (Buf_n1)		0.00	0.19 f
buf0/output[0] (Buf_n1)		0.00	0.19 f
mux00/SEL[0] (MUX31_n15)		0.00	0.19 f
mux00/U474/B (ND2ASVTX8)	0.03	0.00	0.19 f
mux00/U474/Z (ND2ASVTX8)	0.02	0.02	0.21 r
mux00/U477/A (IVSVTX8)	0.02	0.00	0.21 r
mux00/U477/Z (IVSVTX8)	0.02	0.02	0.23 f
mux00/U445/A (IVSVTX10)	0.02	0.00	0.23 f
mux00/U445/Z (IVSVTX10)	0.02	0.02	0.25 r
mux00/U506/B (ND3HVTX1)	0.02	0.00	0.25 r
mux00/U506/Z (ND3HVTX1)	0.05	0.04	0.29 f
mux00/U500/C (ND3SVTX2)	0.05	0.00	0.29 f
mux00/U500/Z (ND3SVTX2)	0.05	0.04	0.33 r
mux00/Z[10] (MUX31_n15)		0.00	0.33 r
adder0/Z[10] (Adder3to2)		0.00	0.33 r
adder0/adder10/C (Adder_5)		0.00	0.33 r
adder0/adder10/halfAdder1/Y (HalfAdder_10)		0.00	0.33 r
adder0/adder10/halfAdder1/U35/B (EOSVTX1)	0.05	0.00	0.33 r
adder0/adder10/halfAdder1/U35/Z (EOSVTX1)	0.04	0.08	0.41 r
adder0/adder10/halfAdder1/P (HalfAdder_10)		0.00	0.41 r
adder0/adder10/WS (Adder_5)		0.00	0.41 r
adder0/WS[10] (Adder3to2)		0.00	0.41 r
WS_shift_reg[11]/D (FD2QSVTX2)	0.04	0.00	0.41 r
data arrival time			0.41

Figure 3.23: Part of radix 2 divider simulation timing report

r-2 divider	logical delay	Synopsys(u:ns)	error
<i>register</i>	8.5478	0.11	22%
<i>selector</i>	0	0	0
<i>buffer</i>	4.3475	0.00	0
$[3 : 1]mux$	12.3556	0.14	12%
$[3 : 2]adder$	6.5806	0.06	10%

Table 3.11: radix 2 divider internal components analysis

function, as we said in the beginning of this section, even though there is some delay during this component, we ignore this data. Since the buffer is implemented with 2 inverters, the delay is small enough to ignore. We can even see from above figure that the multiplexer is synthesized with a nand2, an inverter and a nand3. From above analyzing, we know that the internal delay error is much more evident when using different implementation components. Anyway, more work should be done to find out if our module delay calculation can be applied with general modules simulation.

Radix 4 and radix 8 data can be found in Appendix.

### 3.4.2 Area evaluation

Table 3.12 shows the area comparison between simulation data and calculation data. Selection functions of dividers are not added into the logical area calculation. The control component of on-the-fly convert is split with gates to calculate. Wires are not taken into account as well.

### 3.4.3 Power evaluation

Here we only give the power report from Synopsys. More work should be done to reduce power consumption in the future.

module	Synopsys area(u:um <sup>2</sup> )	Logical area(u:um <sup>2</sup> )
<i>FO4inv</i>	17.561600	16.465
<i>[3 : 2]adder</i>	533.4338	544.416
<i>[4 : 2]adder</i>	1166.7486	1071.28
<i>r - 2div</i>	3725.499	3235.73
<i>r - 4div</i>	5011.642	4801.206
<i>r - 8div</i>	8217.513	8275.063

Table 3.12: area analysis

module	power(u:uW)
<i>FO4inv</i>	122.9452
<i>[3 : 2]adder</i>	2287.4
<i>[4 : 2]adder</i>	2785.6
<i>r - 2div</i>	3346.5
<i>r - 4div</i>	5129.7
<i>r - 8div</i>	7407.3

Table 3.13: power consumption analysis



## Select Function

---

In this chapter, we discuss particularly the quotient digits selection function of dividers. The quotient digits selection function determines the value of the quotient digits as a function of residual  $w$  and of the divisor  $d$  in some cases. Its delay is an important contributor to the iteration time. In current part of this project, two methods are implemented, select table and select comparison.

Select table is a conventional quotient digits selection rule of division. When division is implemented, a large portion of critical path in the implementation is related to the delay of the select table. To increase the performance and reduce this delay, another selection function rule is approached. That is the division selection function with comparison. Comparing with the former one, a large table is avoid and big performance optimization is approached by determining quotient digits only with carry free subtraction and sign detection.

Since carry propagation adder (CPA) is used in both of these two selection rules, in part 1, we will introduce CPA and its delay modeling. In part 2 and 3, we will give a particular description of the selection function implementation. In part 4, we analyze the selection function simulation delay with the logical effort delay.

## 4.1 CPA Modeling

We know that the quotient digit selection function is related with the residual  $w[j]$  and the divisor  $d$ .  $w[j]$  is obtained by adding  $WS[j]$  and  $WC[j]$  together with and carry propagation adder (CPA). Figure (4.1) gives a structure of CPA. It has two input operators and one input carry bit. The critical path is shown in equation (4.1).  $n$  denotes the number of input bits. Thus, the CPA delay is a linear proportion of input operator bits.

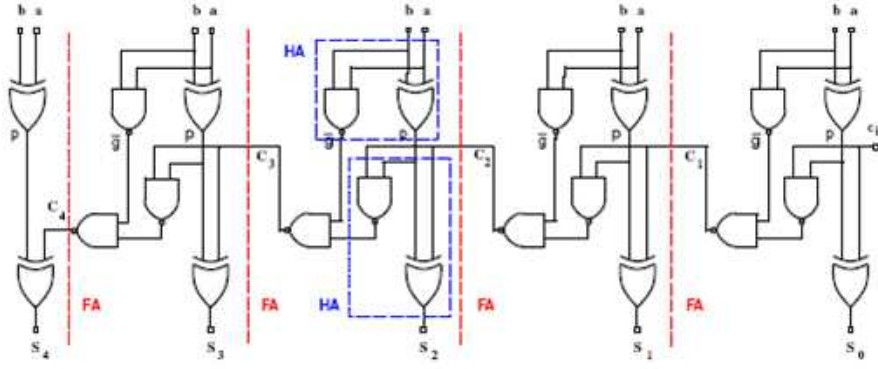


Figure 4.1: 5 bits carry propagate adder cited from [8]

$$t_p = t_{xor} + 2 \cdot (n - 1) \cdot t_{nand} + t_{xor} \quad (4.1)$$

In chapter 3, we approve that logical effort can be applied to this delay calculation for general division blocks without selection function. In this part, we goal is to find out if logical effort can be used in this selection part. To archive our goal, the modeling data is compared with logical effort calculation for this CPA first. Once the logical effort is approved, we can combined the selection function with other division blocks and analyze the whole algorithm simulation data with the logical effort delay.

In this modeling, we build 4, 8, 16, 32 and 64 bits CPA to find the equation related with bits' number and delay. Another parameter that affects delay is the

clock speed. When we changing the clock cycle, the delay will be increase or reduce correspondingly. However, the slowest and the fastest delay should be stable. Figure (4.2) shows the modeling results below.

From lots of raw simulation data sets, we abstract the interaction between delay and clock restrict as shown in figure 4.2. Figure 4.2 demonstrates that for specific bits operator, delay is almost linear growing with the clock restrict within a range. When clock is set to some value big enough, the delay will stop increasing, we obtain the smallest delay,  $T_{fast}$ . On the opposite, when clock is set to some value small enough, we obtain the biggest delay,  $T_{slow}$ . According to above figure sets, equations about  $T_{fast}$  and  $T_{slow}$  can be gained with number of input bits as a parameter.

$$T_{fast} = 32.5n + 98.7 \quad (4.2)$$

$$T_{slow} = 118.7n + 136.7 \quad (4.3)$$

Equation (4.2), (4.3) and figure 4.3 shows how the linear relationship between delay and input signal bits. Data is obtained from Synopsys simulation and equations are obtained with Matlab tool. We only focus on  $T_{slow}$ . The reason is that to approach the smallest delay, Synopsys may change some structure of the module, which can cause some inaccuracy. By using the slowest delay, this inaccuracy can be avoided.

With the implementation shows as figure 4.1, logical effort can be easily gained.

$$d = 9.873n + 11.0775 \quad (4.4)$$

Equations (4.2) and (4.3) are obtained in terms of  $ns$ , while equation (4.4) is obtained in terms of an inverter of FO4 delay. To be comparable, we change

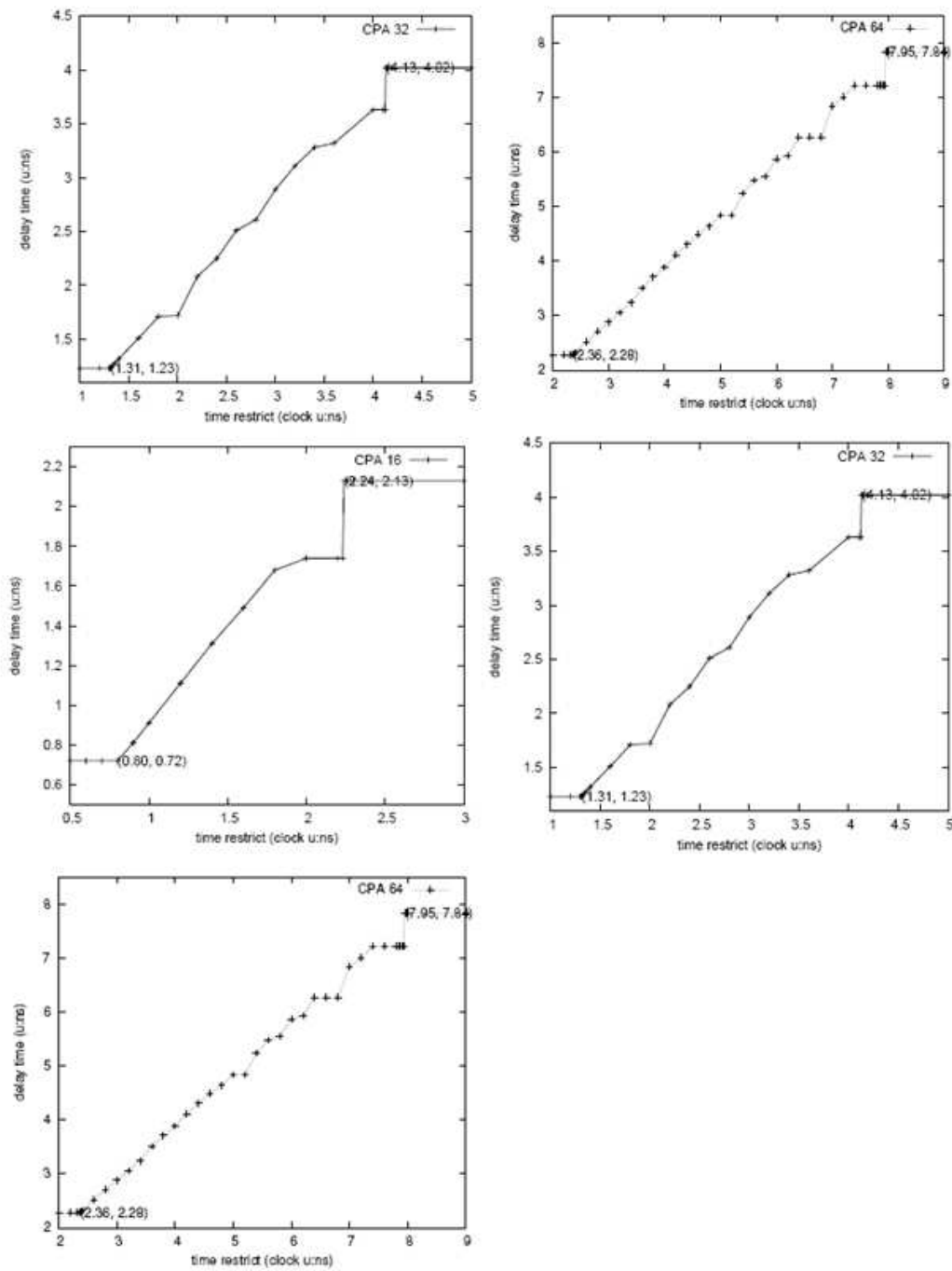


Figure 4.2: CPA delay modeling



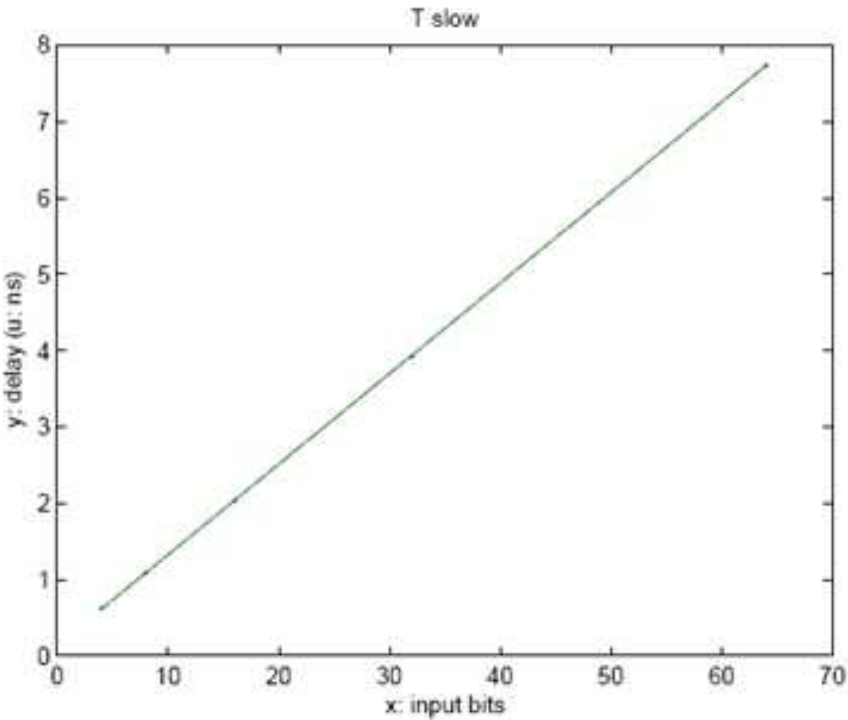


Figure 4.3: CPA delay as a function of input bits

(4.4) into ns set and get:

$$d = 98.73n + 110.775 \quad (4.5)$$

Comparing logical effort with  $T_{slow}$  equation, the error is around 10%, which is acceptable. In the next step, we will apply this logical effort to selection function modeling.

## 4.2 Select Function with Select Table

### 4.2.1 Radix 2 division select table

For radix 2 based division scheme, the quotient digits set is from  $\{-1, 0, 1\}$ . As we defined in equation (3.4), the redundancy factor  $p$ , decides the range of the partial remainder, in the  $(j - 1)th$  iteration. The quotient digit selection has as argument only the truncated carry save shifted residual  $\hat{y}$ . [1] and [9] give complete and particular description of the radix 2 selection table. In this project, we choose to implement the quotient digit selection using selection constants described in [1]. Table (4.1) gives the radix 2 select table.

$q_j + 1$	$w[j]range$
1	$[0, \frac{1}{2}]$
0	$\frac{-1}{2}$
-1	$[\frac{-5}{2}, -1]$

Table 4.1: radix 2 select table

$W[j]$  in table (4.2) is obtained by appending  $ws[j]$  and  $wc[j]$  together using a carry propagate adder (CPA). The first 4 floating bits are truncated to get the quotient digits. Since the quotient digits set is within  $\{-1, \dots, 1\}$ , only 2 bits are used as selected quotient digits from table.

The structure is shown in figure 4.4. Since the select table is written in memory, its time consuming is very small. The critical path includes this CPA and looking up table.

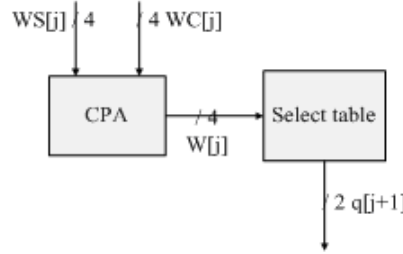


Figure 4.4: 10 or 15 inverter chain

#### 4.2.2 Radix 4 division select table

For radix 4 based division, the quotient digit selection has as arguments the truncated carry save shifted residual  $\hat{y}$  and the truncated divisor  $\hat{d}$ . the quotient digits set is within  $\{-2, -1, 0, 1, 2\}$ , as presented in detail in [1], the selection is described in terms of selection constants  $m_k(i)$ , so that

$$q_j + 1 = k \text{ if } m_k(i) \leq y < m_{k+1}(i) \quad (4.6)$$

Equation (4.6) is referred from [1], for that we are using the constants select table presented in [1]. The select table is shown in table (5.2), which is obtained from above equation.

In table (4.2),  $\{8 \dots 15\}$  denotes the truncated divisor  $\hat{d}$ , we used the first 4 fractional bits as  $\hat{d}$ , the column value set is obtained as  $16\hat{d}$ , since  $d$  is within  $[\frac{1}{2}, 1)$ , this  $16\hat{d}$  is within  $[8, 15)$ . Digits set  $\{-2 \dots 2\}$  denotes the selected quotient digits set. The table content denotes  $16\hat{y}$ ,  $\hat{y}$  is  $4w[j]$  in carry save form and truncated to the fourth fractional bit, which is obtained by adding  $ws[j]$  and  $wc[j]$  together using a CPA. Its range is within  $[42, -44]$ . When  $\hat{y}$  is within

	8	9	10	11	12	13	14	15
2	[12, 42]	[14, 42]	[15, 42]	[16, 42]	[18, 42]	[20, 42]	[20, 42]	[24, 42]
1	[4, 12)	[4, 14)	[4, 15)	[4, 16)	[6, 18)	[6, 20)	[8, 20)	[8, 24)
0	[-4, 4)	[-6, 4)	[-6, 4)	[-6, 4)	[-8, 6)	[-8, 6)	[-8, 8)	[-8, 8)
-1	[-13, -4)	[-15, -6)	[-16, -6)	[-18, -6)	[-20, -8)	[-20, -8)	[-22, -8)	[-24, -8)
-2	[-44, -13)	[-44, -15)	[-44, -16)	[-44, -18)	[-44, -20)	[-44, -20)	[-44, -22)	[-44, -24)

Table 4.2: radix 4 division select table

certain range, the corresponding quotient digit is selected. For example, when  $16\hat{d} = 8$  and  $\hat{y}$  is within  $[12, 42]$ , the selected quotient digit is set to 2. In our implementation, 7 bits are used including 3 integer bits, and 4 fractional bits. The structure is shown in figure 4.5.

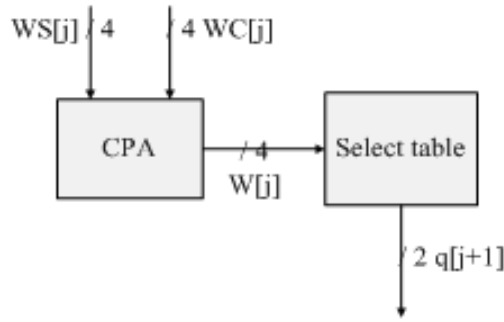


Figure 4.5: radix 4 select function

### 4.2.3 radix 8 division select table

For radix 8 division scheme, as we have discussed above, the quotient digit is decomposed into two components,  $q_H$  and  $q_L$ .  $q_H$  is within set  $\{-8, -4, 0, 4, 8\}$ ;  $q_L$  is within set  $\{-2, -1, 0, 1, 2\}$ . The digit selection scheme is derived from [10]. Paper [10] introduced two select schemes: SELPRI and SELSEC. SELPRI provides a rough estimate of the current quotient digit, which is followed by a refinement in order to get the correct value. When using SELSEC, the

corresponding digits must be derived from the decompositions of  $q_L$  and  $q_H$ , and from the digit selection intervals of SELPRI. Details about these two selection rules can be found in [10]. In our implementation, we use SELSEC rule as this select rule.

When composing  $q$  from  $q_H$  and  $q_L$ , it will imply the existence of a wide overlapping between the consecutive digit selections. For example, in equation (5.3), quotient 6 can be composed by  $q_H = 4$ ;  $q_L = 3$  or by  $q_H = 8$ ;  $q_L = -2$ .

$$q(6) = q_H(4) + q_L(2) = q_H(8) + q_L(-2) \quad (4.7)$$

Therefore, in table 4.3, we designate certain combinations for these overlapped digits.

quotient	$q_H$	$q_L$
6	4	2
2	0	2
-2	0	-2
-6	-4	-2

Table 4.3: combination of  $q_H$  and  $q_L$

The select table of rule SELSEC is derived from [10] as shown in table 4.4. The truncated  $\hat{y}$  is got from eight bits of  $w[j]$ , in which 4 bits are integer bits and 4 bits are fractional bits. Similar with radix 4 select scheme, 4 most significant fractional bits of  $d$  as  $\hat{d}$  are needed.  $W[j]$  is added by  $ws[j]$  and  $wc[j]$  using a 8 bits CPA.

In table 4.4, the number of bits of the constants representing the lower bounds of the digit selection rules is minimized. Observe that some digit selection rules are denoted with three dots, these areas correspond to no-selection-zones since the whole range of possible  $\widehat{w[j]}$  values is already covered by other digit selections, and in these cases the corresponding digit on the same line is not necessary. [10].

...	$8_l$	$8_h$	$9_l$	$9_h$	$10_l$	$10_h$	$11_l$	$11_h$	$12_l$	$12_h$	$13_l$	$13_h$	$14_l$	$14_h$	$15_l$	$15_h$
7	7	7	8	8	8	9	9	9	10	11	10	11	12	13	12	13
6	6	6	6	7	7	7	8	8	8	9	8	9	10	11	10	11
5	5	5	5	5	6	6	6	7	6	7	...	...	8	9	8	9
4	4	4	4	4	4	5	4	5	...	...	6	7	6	7	...	...
3	2	3	2	3	...	...	...	...	4	5	4	5	4	5	4	7
2	...	...	...	...	2	3	2	3	2	3	2	3	...	...	...	...
1	0	1	0	1	0	1	0	1	0	1	0	1	0	3	0	3
0	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
-1	-2	-1	-2	-1	-2	-1	-2	-1	-2	-1	-2	-1	-4	-1	-4	-1
-2	-3	-3	-3	-3	-4	-3	-4	-3	-4	-3	-4	-3	...	...	...	...
-3	-4	-4	-4	-4	...	...	6	-5	-6	-5	-6	-5	-6	-5	-8	-5
-4	-5	-5	-6	-5	-6	-5	...	...	-8	-7	-8	-7	-8	-7	...	...
-5	-6	-6	...	...	-8	-7	-8	-7	...	...	-10	-9	-10	-9	-12	-9
-6	-7	-7	-8	-7	...	...	-10	-9	-10	-9	-12	-11	-12	-11	...	...
-7	-8	-8	-9	-9	-10	-9	-11	-11	-12	-11	...	...	-14	-13	-14	-13

Table 4.4: radix 8 division select table

Quotient digit  $q$  can be derived by combining table 4.3 and table 4.4 together.

The structure and critical path of select functions are:

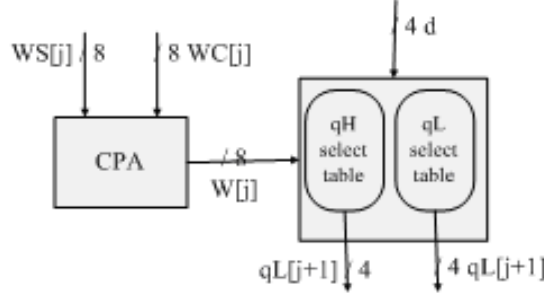


Figure 4.6: radix 8 select function

### 4.3 Select Function with Comparison

From above discussion, we know that carry propagate adder is used for generating operators of looking up table, which is very time consuming. To minimize select function delay, another selection scheme is introduced: select with comparison. In this scheme, only a carry save adder and sign detector are used. A small look up table is also included which is out of critical path.

For this select function with comparison, we only discuss radix-4 scheme. The selection rule is implemented according to paper [11] and [12].

According to [11], the algorithm includes 2 steps:

1. pre-loading the selection constants,  $m_k$
2. perform the comparison in 2 steps:
  - produce subtraction with carry save adder.  $z_k = \hat{y} - m_k$
  - perform the comparison by a sign detector.

In the implementation of [11], the range of radix 4 constant is

$$-\frac{65}{16} \leq \hat{y} - m_k \leq \frac{64}{16} \quad (4.8)$$

Which is shown in table 4.5[12].

$i$	0	1	2	3	4	5	6	7
$m_{-1}$	-13	-15	-16	$[-18, -17]$	$[-20, -19]$	$[-21, -20]$	$[-23, -20]$	$[-25, -23]$
$m_0$	$[-5, -4]$	$[-6, -5]$	$[-6, -5]$	$[-7, -5]$	$[-8, -6]$	$[-8, -6]$	$[-9, -6]$	$[-10, -7]$
$m_1$	$[3, 4]$	$[4, 5]$	$[4, 5]$	$[4, 6]$	$[5, 7]$	$[5, 7]$	$[5, 8]$	$[6, 9]$
$m_2$	12	14	15	$[16, 17]$	$[18, 19]$	$[19, 20]$	$[20, 22]$	$[22, 24]$

Table 4.5: Range of constants scaled by 16 for radix-4 selection scheme.[11]

Figure 4.7 shows the architecture of this comparison scheme. The truncated 4 most significant fractional bits of  $d$  are used as the operator of constant selection. For the selected constant value  $m_k$ , a carry save adder is used to do the subtraction between  $\hat{y}$  and  $m_k$ , the comparison is done with a sign detection of  $z_k$ .

$\hat{y}$	$SD_2$	$SD_1$	$SD_0$	$SD_{-1}$	
$[m_2, H]$	+	+	+	+	$q = 2$ if $(SD_2, SD_1) = (+, +)$
$[m_1, m_2]$	-	+	+	+	$q = 1$ if $(SD_2, SD_1) = (-, +)$
$[m_0, m_1]$	$x$	-	+	+	$q = 0$ if $(SD_1, SD_0) = (-, +)$
$[m_{-1}, m_0]$	$x$	-	-	+	$q = -1$ if $(SD_0, SD_{-1}) = (-, +)$
$[L, m_{-1}]$	$x$	-	-	-	$q = -2$ if $(SD_0, SD_{-1}) = (-, -)$

Table 4.6: quotient digit selection with sign detector.[11]

The critical path of this comparison scheme is :



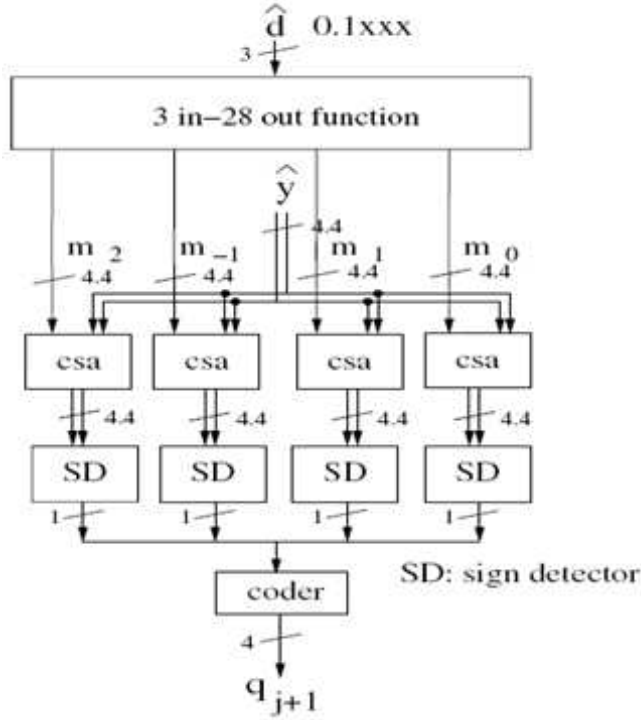


Figure 4.7: select function with comparison architecture [11]



Figure 4.8: critical path of comparison scheme.

## 4.4 Simulation Analyzation

Above select functions are applied in the division and analyzed using Synopsys simulation. Data is shown:

division with select table	delay( $u : ns$ )
radix-2 division	1.17
radix-4 division	1.50
radix-8 division	1.90

Table 4.7: delay analyzation for division with select table

Table 4.7 shows the delay with select table. Select function corresponds almost 40% of the iteration delay. Even though the look up table is written into memory, which is very little time consuming, the carry propagate adder will cost more delay. In particular, 80% of the select function delay is consumed on carry propagate adder. (Timing report can be found in Appendix.)

Table 4.8 shows the comparison between select table and comparison scheme for radix 4 division. It can be seen that the comparison scheme has a much better delay than conventional select table.

division	delay with select table( $u : ns$ )	delay with comparison( $u : ns$ )
radix 4	1.50	1.13

Table 4.8: comparison between two scheme.

## Conclusion

---

In this project, we implemented several arithmetic modules with given algorithms; analyzed and discussed the trade off among the delay, area and power. This project is focus on the delay and area estimation and comparison based on logical effort calculation. Moreover, from particular discussion, we found out the relationship, which can be presented by logical effort, of module delay with input operator bit width. Our simulation results approved that logical effort can be used for general delay calculation for different modules.

We implemented radix 2, 4 and 8 dividers, from the simulation results in chapter 4, we can see that with radix increasing, both of the delay and area are increasing. Simulation results are seen below:

division with select table	delay( $u : ns$ )
radix-2 division	1.17
radix-4 division	1.50
radix-8 division	1.90

Table 5.1: delay analyzation for division with select table

When implementing dividers, we discussed the main architecture and select function separately. Since the select function delay can correspond to half of the

iteration delay. We use two algorithms when implementing it: the conventional look up table and the selection scheme with comparison. From the simulation result in Chapter 4, we can see that the selection with comparison has much better delay than conventional algorithm.

division	delay with select table( $u : ns$ )	delay with comparison( $u : ns$ )
radix 4	1.50	1.13

Table 5.2: comparison between two scheme.

This thesis mainly discusses the delay and area of general modules, more discussion should be done with the power consumption. In addition, more modules should be simulated and applied with logical effort delay calculation to obtain accurate conclusion.

# Bibliography

---

- [1] Milos D. Ercegovic, Tomas Lang. Digital Arithmetic. Morgan Kaufmann, Published Sept. 2003. ISBN 1-55860-798-6.
- [2] Ivan E.Sutherland, Bob F.Sproull, David L.Harris. Logical Effort: Designing Fast CMOS Circuits.
- [3] Alberto Nannarelli, Tomas Lang. Low-Power Radix-8 Divider.
- [4] A.E.Bashagha. Area-efficient nonrestoring radix- $2^k$  division. *Digital Signal Processing*, 15(2005) 367-381.
- [5] Alberto.Nannarelli, Tomas Lang. Lowe-Power Division:Comparison among implementations of radix-4,8 and 16.
- [6] Elisardo Antelo. Delay Model Based on Logical Effort Calculations.
- [7] CORE90GPSVT\_1.00V\_DATABOOK. *CENTRAL R D Design Automation and Intgrated Systems*.
- [8] Neil H.E. Weste, David Harris. CMOS VLSI Design: A Circuits and Systems Perspective. Pearson Education Inc, 3rd. edition, 2005, ISBN 0-321-26977-2.
- [9] Hosahalli R.Srinivas, Keshab K.Parhi, Luis A.Montalvo. Radix 2 Division with Over-Redundant Quotient Selection. *IEEE TRANSACTIONS ON COMPUTERS*, VOL. 46, NO. 1, January 1997.

- [10] Paolo Montuschi, Luigi Ciminiera. Radix-8 Division with Over-Redundant Digit Set. *Journal of VLSI Signal Processing*, 7, 259-270 (1994).
- [11] Elisardo Antelo, Tomas Lang, Paolo Montuschi, Alberto Nannarelli. Fast Radix-4 Retimed Division with Selection by Comparison. Internal Report, Dept. Electrical and Computer Eng., University of California at Irvine, USA.
- [12] Elisardo Antelo, Tomas Lang, Paolo Montuschi, Alberto Nannarelli. Digit-Recurrence Dividers with Reduced Logical Depth. *IEEE TRANSACTIONS ON COMPUTERS*, VOL. 54, NO. 7, July 2005.

## APPENDIX A

# Synopsys Report without Select Function

---

## A.1 Radix-2 Timing Report

\*\*\*\*\*

Report : timing  
    -path full  
    -delay max  
    -input\_pins  
    -max\_paths 1  
    -transition\_time  
    -sort\_by group

Design : DIV2

Version: X-2005.09-SP1

Date : Thu Nov 1 11:52:17 2007

\*\*\*\*\*

Operating Conditions: NomLeak Library: CORE90GPSVT

Wire Load Model Mode: enclosed

Startpoint: reg2/DATA\_OUT\_reg[5]  
                   (rising edge-triggered flip-flop clocked by CLK\_0)  
 Endpoint: WS\_shift\_reg[11]  
                   (rising edge-triggered flip-flop clocked by CLK\_0)  
 Path Group: CLK\_0  
 Path Type: max

Des/Clust/Port	Wire Load Model	Library
DIV2	area_0K	CORE90GPSVT
Radix2Selector	area_0to1K	CORE90GPSVT
MUX31_n15	area_0to1K	CORE90GPSVT

Point	Trans	Incr	Path
clock CLK_0 (rise edge)		0.00	0.00
clock network delay (ideal)		0.00	0.00
reg2/DATA_OUT_reg[5]/CP (FD2QSVTX2)	0.00	0.00	0.00 r
reg2/DATA_OUT_reg[5]/Q (FD2QSVTX2)	0.04	0.11	0.11 f
reg2/DATA_OUT[5] (REG_n15_0)		0.00	0.11 f
sele/Y[0] (Radix2Selector)		0.00	0.11 f
sele/U27/B (NR2SVTX4)	0.04	0.00	0.11 f
sele/U27/Z (NR2SVTX4)	0.03	0.03	0.13 r
sele/U21/A (ND2SVTX4)	0.03	0.00	0.13 r
sele/U21/Z (ND2SVTX4)	0.02	0.02	0.15 f
sele/U25/A (IVSVTX4)	0.02	0.00	0.15 f
sele/U25/Z (IVSVTX4)	0.02	0.02	0.17 r
sele/U26/C (ND3SVTX8)	0.02	0.00	0.17 r
sele/U26/Z (ND3SVTX8)	0.03	0.02	0.19 f
sele/Z[0] (Radix2Selector)		0.00	0.19 f
buf0/input[0] (Buf_n1)		0.00	0.19 f



buf0/output[0] (Buf_n1)		0.00	0.19	f
mux00/SEL[0] (MUX31_n15)		0.00	0.19	f
mux00/U474/B (ND2ASVTX8)	0.03	0.00	0.19	f
mux00/U474/Z (ND2ASVTX8)	0.02	0.02	0.21	r
mux00/U477/A (IVSVTX8)	0.02	0.00	0.21	r
mux00/U477/Z (IVSVTX8)	0.02	0.02	0.23	f
mux00/U445/A (IVSVTX10)	0.02	0.00	0.23	f
mux00/U445/Z (IVSVTX10)	0.02	0.02	0.25	r
mux00/U506/B (ND3HVTX1)	0.02	0.00	0.25	r
mux00/U506/Z (ND3HVTX1)	0.05	0.04	0.29	f
mux00/U500/C (ND3SVTX2)	0.05	0.00	0.29	f
mux00/U500/Z (ND3SVTX2)	0.05	0.04	0.33	r
mux00/Z[10] (MUX31_n15)		0.00	0.33	r
adder0/Z[10] (Adder3to2)		0.00	0.33	r
adder0/adder10/C (Adder_5)		0.00	0.33	r
adder0/adder10/halfAdder1/Y (HalfAdder_10)		0.00	0.33	r
adder0/adder10/halfAdder1/U35/B (EOSVTX1)	0.05	0.00	0.33	r
adder0/adder10/halfAdder1/U35/Z (EOSVTX1)	0.04	0.08	0.41	r
adder0/adder10/halfAdder1/P (HalfAdder_10)		0.00	0.41	r
adder0/adder10/WS (Adder_5)		0.00	0.41	r
adder0/WS[10] (Adder3to2)		0.00	0.41	r
WS_shift_reg[11]/D (FD2QSVTX2)	0.04	0.00	0.41	r
data arrival time			0.41	
clock CLK_0 (rise edge)		0.50	0.50	
clock network delay (ideal)		0.00	0.50	
WS_shift_reg[11]/CP (FD2QSVTX2)		0.00	0.50	r
library setup time		-0.09	0.41	
data required time			0.41	
-----				
data required time			0.41	
data arrival time			-0.41	
-----				

slack (MET) 0.00

## A.2 Radix-2 Area Report

\*\*\*\*\*

Report : area

Design : DIV2

Version: X-2005.09-SP1

Date : Thu Nov 1 11:52:32 2007

\*\*\*\*\*

Library(s) Used:

CORE90GPSVT (File: /cell\_libs/cmos090\_50a/CORE90GPSVT\_SNPS-AVT\_2.1/  
SIGNOFF/bc\_1.10V\_m40C\_wc\_0.90V\_105C/PT\_LIB/CORE90GPSVT\_NomLeak.db)

CORE90GPHVT (File: /cell\_libs/cmos090\_50a/CORE90GPHVT\_SNPS-AVT\_2.1.a/SIGNOFF/  
bc\_1.10V\_m40C\_wc\_0.90V\_105C/PT\_LIB/CORE90GPHVT\_NomLeak.db)

Number of ports: 26

Number of nets: 233

Number of cells: 62

Number of references: 14

Combinational area: 1161.259521

Noncombinational area: 2220.443604

Net Interconnect area: undefined (Wire load has zero net area)

Total cell area: 3381.705566

Total area: undefined

Information: This design contains unmapped logic. (RPT-7)

Information: This design contains black box (unknown) components. (RPT-8)

## A.3 Radix-2 Power Report

\*\*\*\*\*

Report : power

-analysis\_effort low

Design : DIV2

Version: X-2005.09-SP1

Date : Thu Nov 1 11:54:41 2007

\*\*\*\*\*

Library(s) Used:

CORE90GPSVT (File: /cell\_libs/cmos090\_50a/CORE90GPSVT\_SNPS-AVT\_2.1/  
SIGNOFF/bc\_1.10V\_m40C\_wc\_0.90V\_105C/PT\_LIB/CORE90GPSVT\_NomLeak.db)

CORE90GPHVT (File: /cell\_libs/cmos090\_50a/CORE90GPHVT\_SNPS-AVT\_2.1.a/  
SIGNOFF/bc\_1.10V\_m40C\_wc\_0.90V\_105C/PT\_LIB/CORE90GPHVT\_NomLeak.db)

Operating Conditions: NomLeak Library: CORE90GPSVT

Wire Load Model Mode: enclosed

Design	Wire Load Model	Library
-----		
DIV2	area_0K	CORE90GPSVT
Adder3to2	area_0to1K	CORE90GPSVT
Adder_15	area_0to1K	CORE90GPSVT
HalfAdder_31	area_0to1K	CORE90GPSVT
HalfAdder_30	area_0to1K	CORE90GPSVT
NAND_GATE_15	area_0to1K	CORE90GPSVT
Adder_14	area_0to1K	CORE90GPSVT

---

HalfAdder_29	area_0to1K	CORE90GPSVT
HalfAdder_28	area_0to1K	CORE90GPSVT
NAND_GATE_14	area_0to1K	CORE90GPSVT
Adder_13	area_0to1K	CORE90GPSVT
HalfAdder_27	area_0to1K	CORE90GPSVT
HalfAdder_26	area_0to1K	CORE90GPSVT
NAND_GATE_13	area_0to1K	CORE90GPSVT
Adder_12	area_0to1K	CORE90GPSVT
HalfAdder_25	area_0to1K	CORE90GPSVT
HalfAdder_24	area_0to1K	CORE90GPSVT
NAND_GATE_12	area_0to1K	CORE90GPSVT
Adder_11	area_0to1K	CORE90GPSVT
HalfAdder_23	area_0to1K	CORE90GPSVT
HalfAdder_22	area_0to1K	CORE90GPSVT
NAND_GATE_11	area_0to1K	CORE90GPSVT
Adder_10	area_0to1K	CORE90GPSVT
HalfAdder_21	area_0to1K	CORE90GPSVT
HalfAdder_20	area_0to1K	CORE90GPSVT
NAND_GATE_10	area_0to1K	CORE90GPSVT
Adder_9	area_0to1K	CORE90GPSVT
HalfAdder_19	area_0to1K	CORE90GPSVT
HalfAdder_18	area_0to1K	CORE90GPSVT
NAND_GATE_9	area_0to1K	CORE90GPSVT
Adder_8	area_0to1K	CORE90GPSVT
HalfAdder_17	area_0to1K	CORE90GPSVT
HalfAdder_16	area_0to1K	CORE90GPSVT
NAND_GATE_8	area_0to1K	CORE90GPSVT
Adder_7	area_0to1K	CORE90GPSVT
HalfAdder_15	area_0to1K	CORE90GPSVT
HalfAdder_14	area_0to1K	CORE90GPSVT
NAND_GATE_7	area_0to1K	CORE90GPSVT
Adder_6	area_0to1K	CORE90GPSVT

---

HalfAdder_13	area_0to1K	CORE90GPSVT
HalfAdder_12	area_0to1K	CORE90GPSVT
NAND_GATE_6	area_0to1K	CORE90GPSVT
Adder_5	area_0to1K	CORE90GPSVT
HalfAdder_11	area_0to1K	CORE90GPSVT
HalfAdder_10	area_0to1K	CORE90GPSVT
NAND_GATE_5	area_0to1K	CORE90GPSVT
Adder_4	area_0to1K	CORE90GPSVT
HalfAdder_9	area_0to1K	CORE90GPSVT
HalfAdder_8	area_0to1K	CORE90GPSVT
NAND_GATE_4	area_0to1K	CORE90GPSVT
Adder_3	area_0to1K	CORE90GPSVT
HalfAdder_7	area_0to1K	CORE90GPSVT
HalfAdder_6	area_0to1K	CORE90GPSVT
NAND_GATE_3	area_0to1K	CORE90GPSVT
Adder_2	area_0to1K	CORE90GPSVT
HalfAdder_5	area_0to1K	CORE90GPSVT
HalfAdder_4	area_0to1K	CORE90GPSVT
NAND_GATE_2	area_0to1K	CORE90GPSVT
Adder_1	area_0to1K	CORE90GPSVT
HalfAdder_3	area_0to1K	CORE90GPSVT
HalfAdder_2	area_0to1K	CORE90GPSVT
NAND_GATE_1	area_0to1K	CORE90GPSVT
Adder_0	area_0to1K	CORE90GPSVT
HalfAdder_1	area_0to1K	CORE90GPSVT
HalfAdder_0	area_0to1K	CORE90GPSVT
NAND_GATE_0	area_0to1K	CORE90GPSVT
Radix2Selector	area_0to1K	CORE90GPSVT
OTFC2	area_0to1K	CORE90GPSVT
REG12_n15	area_0to1K	CORE90GPSVT
REG_n15_1	area_0to1K	CORE90GPSVT
REG_n15_0	area_0to1K	CORE90GPSVT

MUX31_n15	area_0to1K	CORE90GPSVT
Buf_n1	area_0to1K	CORE90GPSVT

Global Operating Voltage = 1

Power-specific unit information :

Voltage Units = 1V

Capacitance Units = 1.000000pf

Time Units = 1ns

Dynamic Power Units = 1mW (derived from V,C,T units)

Leakage Power Units = 1pW

Cell Internal Power = 3.7569 mW (87%)

Net Switching Power = 544.5258 uW (13%)

-----

Total Dynamic Power = 4.3014 mW (100%)

Cell Leakage Power = 4.7948 uW

## A.4 Radix-4 Timing Report

\*\*\*\*\*

Report : timing

-path full

-delay max

-input\_pins

-max\_paths 1

-transition\_time

-sort\_by group

Design : DIV4

Version: X-2005.09-SP1

Date : Thu Nov 1 12:25:02 2007

\*\*\*\*\*

Operating Conditions: NomLeak Library: CORE90GPSVT

Wire Load Model Mode: enclosed

Startpoint: reg1/DATA\_OUT\_reg[4]

(rising edge-triggered flip-flop clocked by CLK\_0)

Endpoint: WS\_shift\_reg[6]

(rising edge-triggered flip-flop clocked by CLK\_0)

Path Group: CLK\_0

Path Type: max

Des/Clust/Port	Wire Load Model	Library
-----		
DIV4	area_0K	CORE90GPSVT
Radix4Selector	area_0to1K	CORE90GPSVT
MUX31_n15_1	area_0to1K	CORE90GPSVT
MUX51_n15	area_0to1K	CORE90GPSVT
MUX31_n15_0	area_0to1K	CORE90GPSVT
HalfAdder_22	area_0to1K	CORE90GPSVT

Point	Trans	Incr	Path
-----			
clock CLK_0 (rise edge)		0.00	0.00
clock network delay (ideal)		0.00	0.00
reg1/DATA_OUT_reg[4]/CP (FD2QSVTX2)	0.00	0.00	0.00 r
reg1/DATA_OUT_reg[4]/Q (FD2QSVTX2)	0.05	0.11	0.11 f
reg1/DATA_OUT[4] (REG_n15_1)		0.00	0.11 f
sele/X[0] (Radix4Selector)		0.00	0.11 f
sele/U27/B (NR3SVTX6)	0.05	0.00	0.11 f
sele/U27/Z (NR3SVTX6)	0.05	0.04	0.16 r
sele/U25/A (ND3SVTX8)	0.05	0.00	0.16 r

sele/U25/Z (ND3SVTX8)	0.03	0.03	0.19 f
sele/U24/A (IVSVTX6)	0.03	0.00	0.19 f
sele/U24/Z (IVSVTX6)	0.02	0.02	0.20 r
sele/U23/A (ND2SVTX8)	0.02	0.00	0.20 r
sele/U23/Z (ND2SVTX8)	0.03	0.02	0.23 f
sele/Z[2] (Radix4Selector)		0.00	0.23 f
buf0/input[2] (Buf_n3)		0.00	0.23 f
buf0/output[2] (Buf_n3)		0.00	0.23 f
mux00/SEL[2] (MUX51_n15)		0.00	0.23 f
mux00/mux0/SEL[0] (MUX31_n15_1)		0.00	0.23 f
mux00/mux0/U466/B (OR2SVTX4)	0.03	0.00	0.23 f
mux00/mux0/U466/Z (OR2SVTX4)	0.02	0.05	0.28 f
mux00/mux0/U465/A (ND2SVTX6)	0.02	0.00	0.28 f
mux00/mux0/U465/Z (ND2SVTX6)	0.03	0.03	0.31 r
mux00/mux0/U469/A (BFSVTX12)	0.03	0.00	0.31 r
mux00/mux0/U469/Z (BFSVTX12)	0.02	0.04	0.34 r
mux00/mux0/U448/A (A03SVTX1)	0.02	0.00	0.34 r
mux00/mux0/U448/Z (A03SVTX1)	0.06	0.04	0.38 f
mux00/mux0/Z[4] (MUX31_n15_1)		0.00	0.38 f
mux00/mux1/C[4] (MUX31_n15_0)		0.00	0.38 f
mux00/mux1/U487/A (IVSVTX2)	0.06	0.00	0.38 f
mux00/mux1/U487/Z (IVSVTX2)	0.02	0.02	0.40 r
mux00/mux1/U455/A (A023SVTX2)	0.02	0.00	0.40 r
mux00/mux1/U455/Z (A023SVTX2)	0.07	0.05	0.45 f
mux00/mux1/Z[4] (MUX31_n15_0)		0.00	0.45 f
mux00/Z[4] (MUX51_n15)		0.00	0.45 f
adder0/Z[4] (Adder3to2)		0.00	0.45 f
adder0/adder04/C (Adder_11)		0.00	0.45 f
adder0/adder04/halfAdder1/Y (HalfAdder_22)		0.00	0.45 f
adder0/adder04/halfAdder1/U34/A (IVSVTX2)	0.07	0.00	0.45 f
adder0/adder04/halfAdder1/U34/Z (IVSVTX2)	0.02	0.03	0.47 r
adder0/adder04/halfAdder1/U35/B (ND2SVTX2)			



	0.02	0.00	0.47 r
adder0/adder04/halfAdder1/U35/Z (ND2SVTX2)			
	0.03	0.02	0.49 f
adder0/adder04/halfAdder1/U36/A (ND2SVTX2)			
	0.03	0.00	0.49 f
adder0/adder04/halfAdder1/U36/Z (ND2SVTX2)			
	0.02	0.02	0.51 r
adder0/adder04/halfAdder1/P (HalfAdder_22)		0.00	0.51 r
adder0/adder04/WS (Adder_11)		0.00	0.51 r
adder0/WS[4] (Adder3to2)		0.00	0.51 r
WS_shift_reg[6]/D (FD2QSVTX2)	0.02	0.00	0.51 r
data arrival time			0.51
<hr/>			
clock CLK_0 (rise edge)		0.60	0.60
clock network delay (ideal)		0.00	0.60
WS_shift_reg[6]/CP (FD2QSVTX2)		0.00	0.60 r
library setup time		-0.09	0.51
data required time			0.51
<hr/>			
data required time			0.51
data arrival time			-0.51
<hr/>			
slack (MET)			0.00

## A.5 Radix-4 Area Report

\*\*\*\*\*

Report : area

Design : DIV4

Version: X-2005.09-SP1

Date : Thu Nov 1 12:26:14 2007

\*\*\*\*\*

Library(s) Used:

CORE90GPSVT (File: /cell\_libs/cmos090\_50a/CORE90GPSVT\_SNPS-AVT\_2.1/  
SIGNOFF/bc\_1.10V\_m40C\_wc\_0.90V\_105C/PT\_LIB/CORE90GPSVT\_NomLeak.db)

CORE90GPHVT (File: /cell\_libs/cmos090\_50a/CORE90GPHVT\_SNPS-AVT\_2.1.a/  
SIGNOFF/bc\_1.10V\_m40C\_wc\_0.90V\_105C/PT\_LIB/CORE90GPHVT\_NomLeak.db)

Number of ports:	26
Number of nets:	262
Number of cells:	58
Number of references:	13

Combinational area:	1867.016968
Noncombinational area:	3437.680420
Net Interconnect area:	undefined (Wire load has zero net area)

Total cell area:	5304.700684
Total area:	undefined

Information: This design contains unmapped logic. (RPT-7)

## A.6 Radix-4 Power Report

\*\*\*\*\*

Report : power

-analysis\_effort low

Design : DIV4

Version: X-2005.09-SP1

Date : Thu Nov 1 12:26:39 2007

\*\*\*\*\*

Library(s) Used:

CORE90GPSVT (File: /cell\_libs/cmos090\_50a/CORE90GPSVT\_SNPS-AVT\_2.1/  
SIGNOFF/bc\_1.10V\_m40C\_wc\_0.90V\_105C/PT\_LIB/CORE90GPSVT\_NomLeak.db)

CORE90GPHVT (File: /cell\_libs/cmos090\_50a/CORE90GPHVT\_SNPS-AVT\_2.1.a/  
SIGNOFF/bc\_1.10V\_m40C\_wc\_0.90V\_105C/PT\_LIB/CORE90GPHVT\_NomLeak.db)

Operating Conditions: NomLeak Library: CORE90GPSVT

Wire Load Model Mode: enclosed

Design	Wire Load Model	Library
-----		
DIV4	area_0K	CORE90GPSVT
Adder3to2	area_0to1K	CORE90GPSVT
Adder_15	area_0to1K	CORE90GPSVT
HalfAdder_31	area_0to1K	CORE90GPSVT
HalfAdder_30	area_0to1K	CORE90GPSVT
NAND_GATE_15	area_0to1K	CORE90GPSVT
Adder_14	area_0to1K	CORE90GPSVT
HalfAdder_29	area_0to1K	CORE90GPSVT
HalfAdder_28	area_0to1K	CORE90GPSVT
NAND_GATE_14	area_0to1K	CORE90GPSVT
Adder_13	area_0to1K	CORE90GPSVT
HalfAdder_27	area_0to1K	CORE90GPSVT
HalfAdder_26	area_0to1K	CORE90GPSVT
NAND_GATE_13	area_0to1K	CORE90GPSVT
Adder_12	area_0to1K	CORE90GPSVT
HalfAdder_25	area_0to1K	CORE90GPSVT
HalfAdder_24	area_0to1K	CORE90GPSVT
NAND_GATE_12	area_0to1K	CORE90GPSVT

---

Adder_11	area_0to1K	CORE90GPSVT
HalfAdder_23	area_0to1K	CORE90GPSVT
HalfAdder_22	area_0to1K	CORE90GPSVT
NAND_GATE_11	area_0to1K	CORE90GPSVT
Adder_10	area_0to1K	CORE90GPSVT
HalfAdder_21	area_0to1K	CORE90GPSVT
HalfAdder_20	area_0to1K	CORE90GPSVT
NAND_GATE_10	area_0to1K	CORE90GPSVT
Adder_9	area_0to1K	CORE90GPSVT
HalfAdder_19	area_0to1K	CORE90GPSVT
HalfAdder_18	area_0to1K	CORE90GPSVT
NAND_GATE_9	area_0to1K	CORE90GPSVT
Adder_8	area_0to1K	CORE90GPSVT
HalfAdder_17	area_0to1K	CORE90GPSVT
HalfAdder_16	area_0to1K	CORE90GPSVT
NAND_GATE_8	area_0to1K	CORE90GPSVT
Adder_7	area_0to1K	CORE90GPSVT
HalfAdder_15	area_0to1K	CORE90GPSVT
HalfAdder_14	area_0to1K	CORE90GPSVT
NAND_GATE_7	area_0to1K	CORE90GPSVT
Adder_6	area_0to1K	CORE90GPSVT
HalfAdder_13	area_0to1K	CORE90GPSVT
HalfAdder_12	area_0to1K	CORE90GPSVT
NAND_GATE_6	area_0to1K	CORE90GPSVT
Adder_5	area_0to1K	CORE90GPSVT
HalfAdder_11	area_0to1K	CORE90GPSVT
HalfAdder_10	area_0to1K	CORE90GPSVT
NAND_GATE_5	area_0to1K	CORE90GPSVT
Adder_4	area_0to1K	CORE90GPSVT
HalfAdder_9	area_0to1K	CORE90GPSVT
HalfAdder_8	area_0to1K	CORE90GPSVT
NAND_GATE_4	area_0to1K	CORE90GPSVT

Adder_3	area_0to1K	CORE90GPSVT
HalfAdder_7	area_0to1K	CORE90GPSVT
HalfAdder_6	area_0to1K	CORE90GPSVT
NAND_GATE_3	area_0to1K	CORE90GPSVT
Adder_2	area_0to1K	CORE90GPSVT
HalfAdder_5	area_0to1K	CORE90GPSVT
HalfAdder_4	area_0to1K	CORE90GPSVT
NAND_GATE_2	area_0to1K	CORE90GPSVT
Adder_1	area_0to1K	CORE90GPSVT
HalfAdder_3	area_0to1K	CORE90GPSVT
HalfAdder_2	area_0to1K	CORE90GPSVT
NAND_GATE_1	area_0to1K	CORE90GPSVT
Adder_0	area_0to1K	CORE90GPSVT
HalfAdder_1	area_0to1K	CORE90GPSVT
HalfAdder_0	area_0to1K	CORE90GPSVT
NAND_GATE_0	area_0to1K	CORE90GPSVT
Radix4Selector	area_0to1K	CORE90GPSVT
OTFC4	area_1Kto2K	CORE90GPSVT
REG_n7_1	area_0to1K	CORE90GPSVT
REG_n7_0	area_0to1K	CORE90GPSVT
MUX21_n7_1	area_0to1K	CORE90GPSVT
MUX21_n7_0	area_0to1K	CORE90GPSVT
REG14_n15	area_1Kto2K	CORE90GPSVT
REG_n15_1	area_0to1K	CORE90GPSVT
REG_n15_0	area_0to1K	CORE90GPSVT
MUX51_n15	area_0to1K	CORE90GPSVT
MUX31_n15_1	area_0to1K	CORE90GPSVT
MUX31_n15_0	area_0to1K	CORE90GPSVT
Buf_n3	area_0to1K	CORE90GPSVT

Global Operating Voltage = 1

Power-specific unit information :

```

Voltage Units = 1V
Capacitance Units = 1.000000pf
Time Units = 1ns
Dynamic Power Units = 1mW      (derived from V,C,T units)
Leakage Power Units = 1pW

Cell Internal Power   =   5.1072 mW   (85%)
Net Switching Power  =  884.2540 uW   (15%)
-----
Total Dynamic Power   =   5.9915 mW   (100%)

Cell Leakage Power    =   7.4303 uW

```

## A.7 Radix-8 Timing Report

\*\*\*\*\*

Report : timing

```

-path full
-delay max
-input_pins
-max_paths 1
-transition_time
-sort_by group

```

Design : DIV8

Version: X-2005.09-SP1

Date : Thu Nov 1 12:36:18 2007

\*\*\*\*\*

Operating Conditions: NomLeak Library: CORE90GPSVT

Wire Load Model Mode: enclosed

Startpoint: reg2/DATA\_OUT\_reg[6]

(rising edge-triggered flip-flop clocked by CLK\_0)

Endpoint: ws\_shift\_reg[10]

(rising edge-triggered flip-flop clocked by CLK\_0)

Path Group: CLK\_0

Path Type: max

Des/Clust/Port	Wire Load Model	Library
----------------	-----------------	---------

DIV8	area_0K	CORE90GPSVT
Radix8Selector	area_0to1K	CORE90GPSVT
MUX31_n15_3	area_0to1K	CORE90GPSVT
MUX51_n15_1	area_0to1K	CORE90GPSVT
Adder_29	area_0to1K	CORE90GPSVT
HalfAdder_60	area_0to1K	CORE90GPSVT
Adder_30	area_0to1K	CORE90GPSVT

Point	Trans	Incr	Path
clock CLK_0 (rise edge)		0.00	0.00
clock network delay (ideal)		0.00	0.00
reg2/DATA_OUT_reg[6]/CP (FD2QSVTX2)	0.00	0.00	0.00 r
reg2/DATA_OUT_reg[6]/Q (FD2QSVTX2)	0.03	0.08	0.08 r
reg2/DATA_OUT[6] (REG_n15_1)		0.00	0.08 r
sele/X[2] (Radix8Selector)		0.00	0.08 r
sele/U47/A (NR2SVTX4)	0.03	0.00	0.08 r
sele/U47/Z (NR2SVTX4)	0.03	0.03	0.11 f
sele/U50/B (ND3ASVTX6)	0.03	0.00	0.11 f
sele/U50/Z (ND3ASVTX6)	0.03	0.03	0.14 r
sele/U49/A (IVSVTX4)	0.03	0.00	0.14 r
sele/U49/Z (IVSVTX4)	0.02	0.02	0.16 f
sele/U45/B (AN2SVTX6)	0.02	0.00	0.16 f

sele/U45/Z (AN2SVTX6)	0.02	0.04	0.20 f
sele/U43/A (ND2SVTX6)	0.02	0.00	0.20 f
sele/U43/Z (ND2SVTX6)	0.03	0.02	0.22 r
sele/out_h[3] (Radix8Selector)		0.00	0.22 r
buf0/input[3] (Buf_n3_1)		0.00	0.22 r
buf0/output[3] (Buf_n3_1)		0.00	0.22 r
mux0/SEL[3] (MUX51_n15_1)		0.00	0.22 r
mux0/mux0/SEL[1] (MUX31_n15_3)		0.00	0.22 r
mux0/mux0/U471/A (IVSVTX8)	0.03	0.00	0.22 r
mux0/mux0/U471/Z (IVSVTX8)	0.01	0.01	0.24 f
mux0/mux0/U470/A (OR2SVTX8)	0.01	0.00	0.24 f
mux0/mux0/U470/Z (OR2SVTX8)	0.03	0.06	0.30 f
mux0/mux0/U466/B (AN2SVTX2)	0.03	0.00	0.30 f
mux0/mux0/U466/Z (AN2SVTX2)	0.02	0.05	0.35 f
mux0/mux0/U491/A (ND2SVTX4)	0.02	0.00	0.35 f
mux0/mux0/U491/Z (ND2SVTX4)	0.02	0.02	0.37 r
mux0/mux0/U480/B (ND3SVTX4)	0.02	0.00	0.37 r
mux0/mux0/U480/Z (ND3SVTX4)	0.02	0.02	0.39 f
mux0/mux0/Z[6] (MUX31_n15_3)		0.00	0.39 f
mux0/mux1/C[6] (MUX31_n15_2)		0.00	0.39 f
mux0/mux1/U453/A (A03ASVTX6)	0.02	0.00	0.39 f
mux0/mux1/U453/Z (A03ASVTX6)	0.04	0.05	0.45 f
mux0/mux1/Z[6] (MUX31_n15_2)		0.00	0.45 f
mux0/Z[6] (MUX51_n15_1)		0.00	0.45 f
adder0/Z[6] (Adder3to2_1)		0.00	0.45 f
adder0/adder06/C (Adder_29)		0.00	0.45 f
adder0/adder06/halfAdder1/Y (HalfAdder_59)		0.00	0.45 f
adder0/adder06/halfAdder1/U38/B (ND2SVTX2)	0.04	0.00	0.45 f
adder0/adder06/halfAdder1/U38/Z (ND2SVTX2)	0.03	0.03	0.47 r
adder0/adder06/halfAdder1/G (HalfAdder_59)		0.00	0.47 r



adder0/adder06/nandGate/X (NAND_GATE_29)		0.00	0.47	r
adder0/adder06/nandGate/U20/B (ND2SVTX2)	0.03	0.00	0.47	r
adder0/adder06/nandGate/U20/Z (ND2SVTX2)	0.03	0.03	0.50	f
adder0/adder06/nandGate/Z (NAND_GATE_29)		0.00	0.50	f
adder0/adder06/WC (Adder_29)		0.00	0.50	f
adder0/WC[7] (Adder3to2_1)		0.00	0.50	f
adder1/Y[7] (Adder3to2_0)		0.00	0.50	f
adder1/adder07/Y (Adder_30)		0.00	0.50	f
adder1/adder07/halfAdder0/Y (HalfAdder_60)		0.00	0.50	f
adder1/adder07/halfAdder0/U38/A (IVSVTX2)	0.03	0.00	0.50	f
adder1/adder07/halfAdder0/U38/Z (IVSVTX2)	0.02	0.02	0.51	r
adder1/adder07/halfAdder0/U35/B (ND2SVTX2)				
	0.02	0.00	0.51	r
adder1/adder07/halfAdder0/U35/Z (ND2SVTX2)				
	0.03	0.02	0.54	f
adder1/adder07/halfAdder0/U34/A (ND2SVTX4)				
	0.03	0.00	0.54	f
adder1/adder07/halfAdder0/U34/Z (ND2SVTX4)				
	0.03	0.03	0.56	r
adder1/adder07/halfAdder0/P (HalfAdder_60)		0.00	0.56	r
adder1/adder07/halfAdder1/X (HalfAdder_62)		0.00	0.56	r
adder1/adder07/halfAdder1/U37/A (ENSVTX2)	0.03	0.00	0.56	r
adder1/adder07/halfAdder1/U37/Z (ENSVTX2)	0.01	0.05	0.62	r
adder1/adder07/halfAdder1/P (HalfAdder_62)		0.00	0.62	r
adder1/adder07/WS (Adder_30)		0.00	0.62	r
adder1/WS[7] (Adder3to2_0)		0.00	0.62	r
ws_shift_reg[10]/D (FD2QSVTX2)	0.01	0.00	0.62	r
data arrival time			0.62	
clock CLK_0 (rise edge)		0.70	0.70	
clock network delay (ideal)		0.00	0.70	
ws_shift_reg[10]/CP (FD2QSVTX2)		0.00	0.70	r

library setup time	-0.08	0.62
data required time		0.62
-----		
data required time		0.62
data arrival time		-0.62
-----		
slack (MET)		0.00

## A.8 Radix-8 Area Report

\*\*\*\*\*

Report : area

Design : DIV8

Version: X-2005.09-SP1

Date : Thu Nov 1 12:37:10 2007

\*\*\*\*\*

Library(s) Used:

CORE90GPSVT (File: /cell\_libs/cmos090\_50a/CORE90GPSVT\_SNPS-AVT\_2.1/  
SIGNOFF/bc\_1.10V\_m40C\_wc\_0.90V\_105C/PT\_LIB/CORE90GPSVT\_NomLeak.db)

CORE90GPHVT (File: /cell\_libs/cmos090\_50a/CORE90GPHVT\_SNPS-AVT\_2.1.a/  
SIGNOFF/bc\_1.10V\_m40C\_wc\_0.90V\_105C/PT\_LIB/CORE90GPHVT\_NomLeak.db)

Number of ports: 26

Number of nets: 392

Number of cells: 72

Number of references: 23

Combinational area: 3743.918213

Noncombinational area: 4598.940430

Net Interconnect area: undefined (Wire load has zero net area)

Total cell area: 8342.857422

Total area: undefined

Information: This design contains unmapped logic. (RPT-7)

## A.9 Radix-8 Power Report

\*\*\*\*\*

Report : power

-analysis\_effort low

Design : DIV8

Version: X-2005.09-SP1

Date : Thu Nov 1 12:37:20 2007

\*\*\*\*\*

Library(s) Used:

CORE90GPSVT (File: /cell\_libs/cmos090\_50a/CORE90GPSVT\_SNPS-AVT\_2.1/  
SIGNOFF/bc\_1.10V\_m40C\_wc\_0.90V\_105C/PT\_LIB/CORE90GPSVT\_NomLeak.db)

CORE90GPHVT (File: /cell\_libs/cmos090\_50a/CORE90GPHVT\_SNPS-AVT\_2.1.a/  
SIGNOFF/bc\_1.10V\_m40C\_wc\_0.90V\_105C/PT\_LIB/CORE90GPHVT\_NomLeak.db)

Operating Conditions: NomLeak Library: CORE90GPSVT

Wire Load Model Mode: enclosed

Design	Wire Load Model	Library
DIV8	area_0K	CORE90GPSVT
Adder3to2_1	area_0to1K	CORE90GPSVT

---

Adder_17	area_0to1K	CORE90GPSVT
HalfAdder_33	area_0to1K	CORE90GPSVT
HalfAdder_35	area_0to1K	CORE90GPSVT
NAND_GATE_17	area_0to1K	CORE90GPSVT
Adder_19	area_0to1K	CORE90GPSVT
HalfAdder_37	area_0to1K	CORE90GPSVT
HalfAdder_39	area_0to1K	CORE90GPSVT
NAND_GATE_19	area_0to1K	CORE90GPSVT
Adder_21	area_0to1K	CORE90GPSVT
HalfAdder_41	area_0to1K	CORE90GPSVT
HalfAdder_43	area_0to1K	CORE90GPSVT
NAND_GATE_21	area_0to1K	CORE90GPSVT
Adder_23	area_0to1K	CORE90GPSVT
HalfAdder_45	area_0to1K	CORE90GPSVT
HalfAdder_47	area_0to1K	CORE90GPSVT
NAND_GATE_23	area_0to1K	CORE90GPSVT
Adder_25	area_0to1K	CORE90GPSVT
HalfAdder_49	area_0to1K	CORE90GPSVT
HalfAdder_51	area_0to1K	CORE90GPSVT
NAND_GATE_25	area_0to1K	CORE90GPSVT
Adder_27	area_0to1K	CORE90GPSVT
HalfAdder_53	area_0to1K	CORE90GPSVT
HalfAdder_55	area_0to1K	CORE90GPSVT
NAND_GATE_27	area_0to1K	CORE90GPSVT
Adder_29	area_0to1K	CORE90GPSVT
HalfAdder_57	area_0to1K	CORE90GPSVT
HalfAdder_59	area_0to1K	CORE90GPSVT
NAND_GATE_29	area_0to1K	CORE90GPSVT
Adder_31	area_0to1K	CORE90GPSVT
HalfAdder_61	area_0to1K	CORE90GPSVT
HalfAdder_63	area_0to1K	CORE90GPSVT
NAND_GATE_31	area_0to1K	CORE90GPSVT

---

Adder_33	area_0to1K	CORE90GPSVT
HalfAdder_65	area_0to1K	CORE90GPSVT
HalfAdder_67	area_0to1K	CORE90GPSVT
NAND_GATE_33	area_0to1K	CORE90GPSVT
Adder_35	area_0to1K	CORE90GPSVT
HalfAdder_69	area_0to1K	CORE90GPSVT
HalfAdder_71	area_0to1K	CORE90GPSVT
NAND_GATE_35	area_0to1K	CORE90GPSVT
Adder_37	area_0to1K	CORE90GPSVT
HalfAdder_73	area_0to1K	CORE90GPSVT
HalfAdder_75	area_0to1K	CORE90GPSVT
NAND_GATE_37	area_0to1K	CORE90GPSVT
Adder_39	area_0to1K	CORE90GPSVT
HalfAdder_77	area_0to1K	CORE90GPSVT
HalfAdder_79	area_0to1K	CORE90GPSVT
NAND_GATE_39	area_0to1K	CORE90GPSVT
Adder_41	area_0to1K	CORE90GPSVT
HalfAdder_81	area_0to1K	CORE90GPSVT
HalfAdder_83	area_0to1K	CORE90GPSVT
NAND_GATE_41	area_0to1K	CORE90GPSVT
Adder_43	area_0to1K	CORE90GPSVT
HalfAdder_85	area_0to1K	CORE90GPSVT
HalfAdder_87	area_0to1K	CORE90GPSVT
NAND_GATE_43	area_0to1K	CORE90GPSVT
Adder_45	area_0to1K	CORE90GPSVT
HalfAdder_89	area_0to1K	CORE90GPSVT
HalfAdder_91	area_0to1K	CORE90GPSVT
NAND_GATE_45	area_0to1K	CORE90GPSVT
Adder_47	area_0to1K	CORE90GPSVT
HalfAdder_93	area_0to1K	CORE90GPSVT
HalfAdder_95	area_0to1K	CORE90GPSVT
NAND_GATE_47	area_0to1K	CORE90GPSVT

---

Adder3to2_0	area_0to1K	CORE90GPSVT
Adder_16	area_0to1K	CORE90GPSVT
HalfAdder_32	area_0to1K	CORE90GPSVT
HalfAdder_34	area_0to1K	CORE90GPSVT
NAND_GATE_16	area_0to1K	CORE90GPSVT
Adder_18	area_0to1K	CORE90GPSVT
HalfAdder_36	area_0to1K	CORE90GPSVT
HalfAdder_38	area_0to1K	CORE90GPSVT
NAND_GATE_18	area_0to1K	CORE90GPSVT
Adder_20	area_0to1K	CORE90GPSVT
HalfAdder_40	area_0to1K	CORE90GPSVT
HalfAdder_42	area_0to1K	CORE90GPSVT
NAND_GATE_20	area_0to1K	CORE90GPSVT
Adder_22	area_0to1K	CORE90GPSVT
HalfAdder_44	area_0to1K	CORE90GPSVT
HalfAdder_46	area_0to1K	CORE90GPSVT
NAND_GATE_22	area_0to1K	CORE90GPSVT
Adder_24	area_0to1K	CORE90GPSVT
HalfAdder_48	area_0to1K	CORE90GPSVT
HalfAdder_50	area_0to1K	CORE90GPSVT
NAND_GATE_24	area_0to1K	CORE90GPSVT
Adder_26	area_0to1K	CORE90GPSVT
HalfAdder_52	area_0to1K	CORE90GPSVT
HalfAdder_54	area_0to1K	CORE90GPSVT
NAND_GATE_26	area_0to1K	CORE90GPSVT
Adder_28	area_0to1K	CORE90GPSVT
HalfAdder_56	area_0to1K	CORE90GPSVT
HalfAdder_58	area_0to1K	CORE90GPSVT
NAND_GATE_28	area_0to1K	CORE90GPSVT
Adder_30	area_0to1K	CORE90GPSVT
HalfAdder_60	area_0to1K	CORE90GPSVT
HalfAdder_62	area_0to1K	CORE90GPSVT

---

NAND_GATE_30	area_0to1K	CORE90GPSVT
Adder_32	area_0to1K	CORE90GPSVT
HalfAdder_64	area_0to1K	CORE90GPSVT
HalfAdder_66	area_0to1K	CORE90GPSVT
NAND_GATE_32	area_0to1K	CORE90GPSVT
Adder_34	area_0to1K	CORE90GPSVT
HalfAdder_68	area_0to1K	CORE90GPSVT
HalfAdder_70	area_0to1K	CORE90GPSVT
NAND_GATE_34	area_0to1K	CORE90GPSVT
Adder_36	area_0to1K	CORE90GPSVT
HalfAdder_72	area_0to1K	CORE90GPSVT
HalfAdder_74	area_0to1K	CORE90GPSVT
NAND_GATE_36	area_0to1K	CORE90GPSVT
Adder_38	area_0to1K	CORE90GPSVT
HalfAdder_76	area_0to1K	CORE90GPSVT
HalfAdder_78	area_0to1K	CORE90GPSVT
NAND_GATE_38	area_0to1K	CORE90GPSVT
Adder_40	area_0to1K	CORE90GPSVT
HalfAdder_80	area_0to1K	CORE90GPSVT
HalfAdder_82	area_0to1K	CORE90GPSVT
NAND_GATE_40	area_0to1K	CORE90GPSVT
Adder_42	area_0to1K	CORE90GPSVT
HalfAdder_84	area_0to1K	CORE90GPSVT
HalfAdder_86	area_0to1K	CORE90GPSVT
NAND_GATE_42	area_0to1K	CORE90GPSVT
Adder_44	area_0to1K	CORE90GPSVT
HalfAdder_88	area_0to1K	CORE90GPSVT
HalfAdder_90	area_0to1K	CORE90GPSVT
NAND_GATE_44	area_0to1K	CORE90GPSVT
Adder_46	area_0to1K	CORE90GPSVT
HalfAdder_92	area_0to1K	CORE90GPSVT
HalfAdder_94	area_0to1K	CORE90GPSVT

NAND_GATE_46	area_0to1K	CORE90GPSVT
Radix8Selector	area_0to1K	CORE90GPSVT
OTFC8	area_1Kto2K	CORE90GPSVT
REG_n7_1	area_0to1K	CORE90GPSVT
REG_n7_0	area_0to1K	CORE90GPSVT
MUX21_n7_1	area_0to1K	CORE90GPSVT
MUX21_n7_0	area_0to1K	CORE90GPSVT
REG14_n15	area_1Kto2K	CORE90GPSVT
REG18_n15	area_1Kto2K	CORE90GPSVT
REG_n15_1	area_0to1K	CORE90GPSVT
REG_n15_0	area_0to1K	CORE90GPSVT
MUX51_n15_1	area_0to1K	CORE90GPSVT
MUX31_n15_3	area_0to1K	CORE90GPSVT
MUX31_n15_2	area_0to1K	CORE90GPSVT
MUX51_n15_0	area_0to1K	CORE90GPSVT
MUX31_n15_1	area_0to1K	CORE90GPSVT
MUX31_n15_0	area_0to1K	CORE90GPSVT
Buf_n3_1	area_0to1K	CORE90GPSVT
Buf_n3_0	area_0to1K	CORE90GPSVT

Global Operating Voltage = 1

Power-specific unit information :

Voltage Units = 1V

Capacitance Units = 1.000000pf

Time Units = 1ns

Dynamic Power Units = 1mW (derived from V,C,T units)

Leakage Power Units = 1pW

Cell Internal Power = 6.1716 mW (83%)

Net Switching Power = 1.2357 mW (17%)

-----

Total Dynamic Power = 7.4073 mW (100%)



Cell Leakage Power      =   13.1898 uW



## APPENDIX B

# Synopsys Report with Select Function

---

## B.1 Radix-2 Timing Report

\*\*\*\*\*

Report : timing

-path full

-delay max

-max\_paths 1

-sort\_by group

Design : DIV2

Version: X-2005.09-SP1

Date : Thu Jan 10 08:42:50 2008

\*\*\*\*\*

Operating Conditions: NomLeak Library: CORE90GPSVT

Wire Load Model Mode: enclosed

Startpoint: reg1/DATA\_OUT\_reg[7]

(rising edge-triggered flip-flop clocked by CLK\_0)

Endpoint: WS\_shift\_reg[8]

(rising edge-triggered flip-flop clocked by CLK\_0)

Path Group: CLK\_0

Path Type: max

Des/Clust/Port	Wire Load Model	Library
----------------	-----------------	---------

-----

DIV2	area_0K	CORE90GPSVT
Adder_3	area_0to1K	CORE90GPSVT
CPA4	area_0to1K	CORE90GPSVT
Adder_2	area_0to1K	CORE90GPSVT
Adder_1	area_0to1K	CORE90GPSVT
Radix2Selector	area_0to1K	CORE90GPSVT
MUX31_n15	area_0to1K	CORE90GPSVT

Point	Incr	Path
-------	------	------

-----

clock CLK_0 (rise edge)	0.00	0.00
clock network delay (ideal)	0.00	0.00
reg1/DATA_OUT_reg[7]/CP (FD2QSVTX2)	0.00	0.00 r
reg1/DATA_OUT_reg[7]/Q (FD2QSVTX2)	0.11	0.11 f
reg1/DATA_OUT[7] (REG_n15_1)	0.00	0.11 f
sele/X[0] (Radix2Selector)	0.00	0.11 f
sele/cpa/X[0] (CPA4)	0.00	0.11 f
sele/cpa/adder00/X (Adder_3)	0.00	0.11 f
sele/cpa/adder00/halfAdder0/X (HalfAdder_7)	0.00	0.11 f
sele/cpa/adder00/halfAdder0/U9/Z (EOSVTX1)	0.10	0.21 r
sele/cpa/adder00/halfAdder0/P (HalfAdder_7)	0.00	0.21 r
sele/cpa/adder00/halfAdder1/X (HalfAdder_6)	0.00	0.21 r
sele/cpa/adder00/halfAdder1/U9/Z (ND2SVTX2)	0.03	0.25 f
sele/cpa/adder00/halfAdder1/G (HalfAdder_6)	0.00	0.25 f

sele/cpa/adder00/nandGate/X (NAND_GATE_3)	0.00	0.25 f
sele/cpa/adder00/nandGate/U8/Z (ND2SVTX2)	0.03	0.28 r
sele/cpa/adder00/nandGate/Z (NAND_GATE_3)	0.00	0.28 r
sele/cpa/adder00/WC (Adder_3)	0.00	0.28 r
sele/cpa/adder01/C (Adder_2)	0.00	0.28 r
sele/cpa/adder01/halfAdder1/Y (HalfAdder_4)	0.00	0.28 r
sele/cpa/adder01/halfAdder1/U10/Z (ND2SVTX2)	0.02	0.30 f
sele/cpa/adder01/halfAdder1/G (HalfAdder_4)	0.00	0.30 f
sele/cpa/adder01/nandGate/X (NAND_GATE_2)	0.00	0.30 f
sele/cpa/adder01/nandGate/U8/Z (ND2SVTX2)	0.03	0.33 r
sele/cpa/adder01/nandGate/Z (NAND_GATE_2)	0.00	0.33 r
sele/cpa/adder01/WC (Adder_2)	0.00	0.33 r
sele/cpa/adder02/C (Adder_1)	0.00	0.33 r
sele/cpa/adder02/halfAdder1/Y (HalfAdder_2)	0.00	0.33 r
sele/cpa/adder02/halfAdder1/U10/Z (ND2SVTX2)	0.02	0.36 f
sele/cpa/adder02/halfAdder1/G (HalfAdder_2)	0.00	0.36 f
sele/cpa/adder02/nandGate/X (NAND_GATE_1)	0.00	0.36 f
sele/cpa/adder02/nandGate/U8/Z (ND2SVTX2)	0.03	0.39 r
sele/cpa/adder02/nandGate/Z (NAND_GATE_1)	0.00	0.39 r
sele/cpa/adder02/WC (Adder_1)	0.00	0.39 r
sele/cpa/adder03/C (Adder_0)	0.00	0.39 r
sele/cpa/adder03/halfAdder1/Y (HalfAdder_0)	0.00	0.39 r
sele/cpa/adder03/halfAdder1/U9/Z (EOSVTX1)	0.08	0.47 r
sele/cpa/adder03/halfAdder1/P (HalfAdder_0)	0.00	0.47 r
sele/cpa/adder03/WS (Adder_0)	0.00	0.47 r
sele/cpa/WS[3] (CPA4)	0.00	0.47 r
sele/U21/Z (A07SVTX1)	0.07	0.54 f
sele/U20/Z (A06ASVTX2)	0.07	0.60 r
sele/Z[1] (Radix2Selector)	0.00	0.60 r
buf0/input[1] (Buf_n1)	0.00	0.60 r
buf0/output[1] (Buf_n1)	0.00	0.60 r
mux00/SEL[1] (MUX31_n15)	0.00	0.60 r

mux00/U29/Z (NR2ASVTX1)	0.28	0.88 r
mux00/U39/Z (A011NSVTX1)	0.21	1.09 r
mux00/Z[7] (MUX31_n15)	0.00	1.09 r
adder0/Z[7] (Adder3to2)	0.00	1.09 r
adder0/adder07/C (Adder_12)	0.00	1.09 r
adder0/adder07/halfAdder1/Y (HalfAdder_24)	0.00	1.09 r
adder0/adder07/halfAdder1/U9/Z (EOSVTX1)	0.07	1.17 r
adder0/adder07/halfAdder1/P (HalfAdder_24)	0.00	1.17 r
adder0/adder07/WS (Adder_12)	0.00	1.17 r
adder0/WS[7] (Adder3to2)	0.00	1.17 r
WS_shift_reg[8]/D (FD2QSVTX2)	0.00	1.17 r
data arrival time		1.17
<hr/>		
clock CLK_0 (rise edge)	2.00	2.00
clock network delay (ideal)	0.00	2.00
WS_shift_reg[8]/CP (FD2QSVTX2)	0.00	2.00 r
library setup time	-0.09	1.91
data required time		1.91
<hr/>		
data required time		1.91
data arrival time		-1.17
<hr/>		
slack (MET)		0.74

## B.2 Radix-4 Timing Report (select table)

\*\*\*\*\*

Report : timing

-path full

-delay max

-max\_paths 1

-sort\_by group

Design : divr4\_rec

Version: X-2005.09-SP1

Date : Thu Jan 17 07:45:14 2008

\*\*\*\*\*

Operating Conditions: NomLeak Library: CORE90GPSVT

Wire Load Model Mode: enclosed

Startpoint: I\_REG1/ZC\_reg[5]

(rising edge-triggered flip-flop clocked by CLK\_0)

Endpoint: I\_REG2/ZS\_reg[45]

(rising edge-triggered flip-flop clocked by CLK\_0)

Path Group: CLK\_0

Path Type: max

Des/Clust/Port	Wire Load Model	Library
-----		
divr4_rec	area_6Kto12K	CORE90GPSVT
QDSEL	area_0to1K	CORE90GPSVT
QDS_TABLE	area_0to1K	CORE90GPSVT
MULT	area_0to1K	CORE90GPSVT
gl_dualreg_ld_n45	area_2Kto3K	CORE90GPSVT

Point	Incr	Path
-----		
clock CLK_0 (rise edge)	0.00	0.00
clock network delay (ideal)	0.00	0.00
I_REG1/ZC_reg[5]/CP (FD2QSVTX2)	0.00	0.00 r
I_REG1/ZC_reg[5]/Q (FD2QSVTX2)	0.13	0.13 f
I_REG1/ZC[5] (gl_dualreg_ld_n10)	0.00	0.13 f

I_SEL/A2[1] (QDSEL)	0.00	0.13 f
I_SEL/I_2/A2[1] (QDS_ADDER)	0.00	0.13 f
I_SEL/I_2/U48/Z (E03HVTX1)	0.28	0.41 f
I_SEL/I_2/Y[1] (QDS_ADDER)	0.00	0.41 f
I_SEL/I_1/Y[1] (QDS_TABLE)	0.00	0.41 f
I_SEL/I_1/U835/Z (IVSVTX4)	0.04	0.45 r
I_SEL/I_1/U840/Z (ENSVTX1)	0.08	0.53 f
I_SEL/I_1/U789/Z (A020CHVTX1)	0.15	0.67 f
I_SEL/I_1/U787/Z (A07HVTX1)	0.09	0.76 r
I_SEL/I_1/U847/Z (A07SVTX2)	0.06	0.83 f
I_SEL/I_1/U833/Z (A08SVTX8)	0.14	0.96 r
I_SEL/I_1/P2 (QDS_TABLE)	0.00	0.96 r
I_SEL/P2 (QDSEL)	0.00	0.96 r
I_MULT/P2 (MULT)	0.00	0.96 r
I_MULT/U56/Z (A02HVTX1)	0.13	1.09 f
I_MULT/U55/Z (A023HVTX1)	0.13	1.23 r
I_MULT/Z[45] (MULT)	0.00	1.23 r
I_CSA2/C[45] (csa32LSBs_n47)	0.00	1.23 r
I_CSA2/U15/Z (EOHVTX1)	0.16	1.38 r
I_CSA2/Z[45] (csa32LSBs_n47)	0.00	1.38 r
I_REG2/AS[45] (gl_dualreg_ld_n45)	0.00	1.38 r
I_REG2/U209/Z (A02NHVTX1)	0.12	1.50 r
I_REG2/ZS_reg[45]/D (FD2QSVTX2)	0.00	1.50 r
data arrival time		1.50
clock CLK_0 (rise edge)	1.60	1.60
clock network delay (ideal)	0.00	1.60
I_REG2/ZS_reg[45]/CP (FD2QSVTX2)	0.00	1.60 r
library setup time	-0.09	1.51
data required time		1.51
-----		
data required time		1.51



data arrival time	-1.50
-----	
slack (MET)	0.00

### B.3 Radix-4 Timing Report (Comparison)

\*\*\*\*\*

Report : timing

-path full  
-delay max  
-max\_paths 1  
-sort\_by group

Design : divr4\_rec

Version: X-2005.09-SP1

Date : Thu Jan 17 08:53:11 2008

\*\*\*\*\*

Operating Conditions: NomLeak Library: CORE90GPSVT

Wire Load Model Mode: enclosed

Startpoint: I\_REG1/ZS\_reg[9]

(rising edge-triggered flip-flop clocked by CLK\_0)

Endpoint: I\_REG1/ZS\_reg[4]

(rising edge-triggered flip-flop clocked by CLK\_0)

Path Group: CLK\_0

Path Type: max

Des/Clust/Port	Wire Load Model	Library
-----		
divr4_rec	area_6Kto12K	CORE90GPSVT
Adder_17	area_0to1K	CORE90GPSVT

QDSEL2	area_2Kto3K	CORE90GPSVT
SD_2	area_0to1K	CORE90GPSVT
Coder4	area_0to1K	CORE90GPSVT
MULT	area_1Kto2K	CORE90GPSVT
gl_dualreg_ld_n10	area_0to1K	CORE90GPSVT

Point	Incr	Path
-----		
clock CLK_0 (rise edge)	0.00	0.00
clock network delay (ideal)	0.00	0.00
I_REG1/ZS_reg[9]/CP (FD2QSVTX4)	0.00	0.00 r
I_REG1/ZS_reg[9]/Q (FD2QSVTX4)	0.15	0.15 f
I_REG1/ZS[9] (gl_dualreg_ld_n10)	0.00	0.15 f
I_SEL/A1[6] (QDSEL2)	0.00	0.15 f
I_SEL/csa1/Y[6] (CSA8_2)	0.00	0.15 f
I_SEL/csa1/adder06/Y (Adder_17)	0.00	0.15 f
I_SEL/csa1/adder06/halfAdder0/Y (HalfAdder_35)	0.00	0.15 f
I_SEL/csa1/adder06/halfAdder0/U10/Z (EOSVTX4)	0.09	0.24 f
I_SEL/csa1/adder06/halfAdder0/P (HalfAdder_35)	0.00	0.24 f
I_SEL/csa1/adder06/halfAdder1/X (HalfAdder_34)	0.00	0.24 f
I_SEL/csa1/adder06/halfAdder1/U9/Z (EOSVTX8)	0.05	0.29 f
I_SEL/csa1/adder06/halfAdder1/P (HalfAdder_34)	0.00	0.29 f
I_SEL/csa1/adder06/WS (Adder_17)	0.00	0.29 f
I_SEL/csa1/WS[6] (CSA8_2)	0.00	0.29 f
I_SEL/sd1/WS[6] (SD_2)	0.00	0.29 f
I_SEL/sd1/U57/Z (IVSVTX2)	0.02	0.31 r
I_SEL/sd1/U49/Z (AN2BSVTX1)	0.10	0.41 f
I_SEL/sd1/U38/Z (A017ASVTX4)	0.04	0.45 r
I_SEL/sd1/U46/Z (E03SVTX4)	0.12	0.57 f
I_SEL/sd1/U56/Z (NR2ASVTX8)	0.03	0.59 r
I_SEL/sd1/S (SD_2)	0.00	0.59 r
I_SEL/coder0/S_1 (Coder4)	0.00	0.59 r

I_SEL/coder0/U26/Z (AN2BSVTX6)	0.08	0.68 f
I_SEL/coder0/U29/Z (IVSVTX4)	0.01	0.69 r
I_SEL/coder0/U22/Z (ND4ABSVTX4)	0.03	0.72 f
I_SEL/coder0/U28/Z (IVSVTX10)	0.02	0.74 r
I_SEL/coder0/Q[2] (Coder4)	0.00	0.74 r
I_SEL/P1 (QDSEL2)	0.00	0.74 r
I_MULT/P1 (MULT)	0.00	0.74 r
I_MULT/U177/Z (IVSVTX12)	0.04	0.78 f
I_MULT/U37/Z (AO23HVTX1)	0.13	0.91 r
I_MULT/Z[50] (MULT)	0.00	0.91 r
I_CSA1/C[2] (gl_csa32_n8)	0.00	0.91 r
I_CSA1/U44/Z (EOSVTX4)	0.12	1.03 f
I_CSA1/Z[2] (gl_csa32_n8)	0.00	1.03 f
I_REG1/AS[4] (gl_dualreg_ld_n10)	0.00	1.03 f
I_REG1/U93/Z (AO2NSVTX1)	0.10	1.13 f
I_REG1/ZS_reg[4]/D (FD2QSVTX4)	0.00	1.13 f
data arrival time		1.13
clock CLK_0 (rise edge)	1.20	1.20
clock network delay (ideal)	0.00	1.20
I_REG1/ZS_reg[4]/CP (FD2QSVTX4)	0.00	1.20 r
library setup time	-0.07	1.13
data required time		1.13
-----		
data required time		1.13
data arrival time		-1.13
-----		
slack (MET)		0.00

## B.4 Radix-8 Timing Report

\*\*\*\*\*

Report : timing

-path full

-delay max

-max\_paths 1

-sort\_by group

Design : DIV8

Version: X-2005.09-SP1

Date : Thu Jan 10 11:34:47 2008

\*\*\*\*\*

Operating Conditions: NomLeak Library: CORE90GPSVT

Wire Load Model Mode: enclosed

Startpoint: reg3/DATA\_OUT\_reg[5]

(rising edge-triggered flip-flop clocked by CLK\_0)

Endpoint: ws\_shift\_reg[13]

(rising edge-triggered flip-flop clocked by CLK\_0)

Path Group: CLK\_0

Path Type: max

Des/Clust/Port	Wire Load Model	Library
-----		
DIV8	area_0K	CORE90GPSVT
HalfAdder_15	area_0to1K	CORE90GPSVT
Adder_7	area_0to1K	CORE90GPSVT
CPA8	area_0to1K	CORE90GPSVT
Adder_6	area_0to1K	CORE90GPSVT
Adder_5	area_0to1K	CORE90GPSVT
Adder_4	area_0to1K	CORE90GPSVT

Adder_3	area_0to1K	CORE90GPSVT
Adder_2	area_0to1K	CORE90GPSVT
Adder_1	area_0to1K	CORE90GPSVT
Radix8Selector	area_1Kto2K	CORE90GPSVT
MUX31_n15_3	area_0to1K	CORE90GPSVT
MUX51_n15_1	area_0to1K	CORE90GPSVT
Adder_13	area_0to1K	CORE90GPSVT

Point	Incr	Path
-----		
clock CLK_0 (rise edge)	0.00	0.00
clock network delay (ideal)	0.00	0.00
reg3/DATA_OUT_reg[5]/CP (FD2QSVTX2)	0.00	0.00 r
reg3/DATA_OUT_reg[5]/Q (FD2QSVTX2)	0.11	0.11 f
reg3/DATA_OUT[5] (REG_n15_0)	0.00	0.11 f
sele/Y[0] (Radix8Selector)	0.00	0.11 f
sele/cpa/Y[0] (CPA8)	0.00	0.11 f
sele/cpa/adder00/Y (Adder_7)	0.00	0.11 f
sele/cpa/adder00/halfAdder0/Y (HalfAdder_15)	0.00	0.11 f
sele/cpa/adder00/halfAdder0/U12/Z (IVSVTX2)	0.02	0.13 r
sele/cpa/adder00/halfAdder0/U10/Z (ND2SVTX1)	0.03	0.16 f
sele/cpa/adder00/halfAdder0/U14/Z (ND2SVTX2)	0.04	0.19 r
sele/cpa/adder00/halfAdder0/P (HalfAdder_15)	0.00	0.19 r
sele/cpa/adder00/halfAdder1/X (HalfAdder_14)	0.00	0.19 r
sele/cpa/adder00/halfAdder1/U10/Z (ND2SVTX2)	0.03	0.22 f
sele/cpa/adder00/halfAdder1/G (HalfAdder_14)	0.00	0.22 f
sele/cpa/adder00/nandGate/X (NAND_GATE_7)	0.00	0.22 f
sele/cpa/adder00/nandGate/U8/Z (ND2SVTX2)	0.03	0.25 r
sele/cpa/adder00/nandGate/Z (NAND_GATE_7)	0.00	0.25 r
sele/cpa/adder00/WC (Adder_7)	0.00	0.25 r
sele/cpa/adder01/C (Adder_6)	0.00	0.25 r
sele/cpa/adder01/halfAdder1/Y (HalfAdder_12)	0.00	0.25 r

sele/cpa/adder01/halfAdder1/U10/Z (ND2SVTX2)	0.02	0.27 f
sele/cpa/adder01/halfAdder1/G (HalfAdder_12)	0.00	0.27 f
sele/cpa/adder01/nandGate/X (NAND_GATE_6)	0.00	0.27 f
sele/cpa/adder01/nandGate/U8/Z (ND2SVTX2)	0.03	0.30 r
sele/cpa/adder01/nandGate/Z (NAND_GATE_6)	0.00	0.30 r
sele/cpa/adder01/WC (Adder_6)	0.00	0.30 r
sele/cpa/adder02/C (Adder_5)	0.00	0.30 r
sele/cpa/adder02/halfAdder1/Y (HalfAdder_10)	0.00	0.30 r
sele/cpa/adder02/halfAdder1/U10/Z (ND2SVTX2)	0.02	0.33 f
sele/cpa/adder02/halfAdder1/G (HalfAdder_10)	0.00	0.33 f
sele/cpa/adder02/nandGate/X (NAND_GATE_5)	0.00	0.33 f
sele/cpa/adder02/nandGate/U8/Z (ND2SVTX2)	0.03	0.36 r
sele/cpa/adder02/nandGate/Z (NAND_GATE_5)	0.00	0.36 r
sele/cpa/adder02/WC (Adder_5)	0.00	0.36 r
sele/cpa/adder03/C (Adder_4)	0.00	0.36 r
sele/cpa/adder03/halfAdder1/Y (HalfAdder_8)	0.00	0.36 r
sele/cpa/adder03/halfAdder1/U10/Z (ND2SVTX2)	0.02	0.38 f
sele/cpa/adder03/halfAdder1/G (HalfAdder_8)	0.00	0.38 f
sele/cpa/adder03/nandGate/X (NAND_GATE_4)	0.00	0.38 f
sele/cpa/adder03/nandGate/U8/Z (ND2SVTX2)	0.03	0.41 r
sele/cpa/adder03/nandGate/Z (NAND_GATE_4)	0.00	0.41 r
sele/cpa/adder03/WC (Adder_4)	0.00	0.41 r
sele/cpa/adder04/C (Adder_3)	0.00	0.41 r
sele/cpa/adder04/halfAdder1/Y (HalfAdder_6)	0.00	0.41 r
sele/cpa/adder04/halfAdder1/U14/Z (ND2SVTX2)	0.02	0.44 f
sele/cpa/adder04/halfAdder1/G (HalfAdder_6)	0.00	0.44 f
sele/cpa/adder04/nandGate/X (NAND_GATE_3)	0.00	0.44 f
sele/cpa/adder04/nandGate/U8/Z (ND2SVTX2)	0.03	0.47 r
sele/cpa/adder04/nandGate/Z (NAND_GATE_3)	0.00	0.47 r
sele/cpa/adder04/WC (Adder_3)	0.00	0.47 r
sele/cpa/adder05/C (Adder_2)	0.00	0.47 r
sele/cpa/adder05/halfAdder1/Y (HalfAdder_4)	0.00	0.47 r

sele/cpa/adder05/halfAdder1/U10/Z (ND2SVTX2)	0.02	0.49 f
sele/cpa/adder05/halfAdder1/G (HalfAdder_4)	0.00	0.49 f
sele/cpa/adder05/nandGate/X (NAND_GATE_2)	0.00	0.49 f
sele/cpa/adder05/nandGate/U8/Z (ND2SVTX2)	0.03	0.53 r
sele/cpa/adder05/nandGate/Z (NAND_GATE_2)	0.00	0.53 r
sele/cpa/adder05/WC (Adder_2)	0.00	0.53 r
sele/cpa/adder06/C (Adder_1)	0.00	0.53 r
sele/cpa/adder06/halfAdder1/Y (HalfAdder_2)	0.00	0.53 r
sele/cpa/adder06/halfAdder1/U10/Z (ND2SVTX2)	0.02	0.55 f
sele/cpa/adder06/halfAdder1/G (HalfAdder_2)	0.00	0.55 f
sele/cpa/adder06/nandGate/X (NAND_GATE_1)	0.00	0.55 f
sele/cpa/adder06/nandGate/U8/Z (ND2SVTX2)	0.03	0.58 r
sele/cpa/adder06/nandGate/Z (NAND_GATE_1)	0.00	0.58 r
sele/cpa/adder06/WC (Adder_1)	0.00	0.58 r
sele/cpa/adder07/C (Adder_0)	0.00	0.58 r
sele/cpa/adder07/halfAdder1/Y (HalfAdder_0)	0.00	0.58 r
sele/cpa/adder07/halfAdder1/U10/Z (EOSVTX1)	0.11	0.69 r
sele/cpa/adder07/halfAdder1/P (HalfAdder_0)	0.00	0.69 r
sele/cpa/adder07/WS (Adder_0)	0.00	0.69 r
sele/cpa/WS[7] (CPA8)	0.00	0.69 r
sele/U1529/Z (IVSVTX2)	0.05	0.74 f
sele/U1528/Z (NR2AHVTX1)	0.10	0.84 r
sele/U1498/Z (A020DSVTX1)	0.21	1.05 r
sele/out_h[2] (Radix8Selector)	0.00	1.05 r
buf0/input[2] (Buf_n3_1)	0.00	1.05 r
buf0/output[2] (Buf_n3_1)	0.00	1.05 r
mux0/SEL[2] (MUX51_n15_1)	0.00	1.05 r
mux0/mux0/SEL[0] (MUX31_n15_3)	0.00	1.05 r
mux0/mux0/U36/Z (AN2BHVTX1)	0.17	1.22 f
mux0/mux0/U34/Z (BFSVTX4)	0.07	1.29 f
mux0/mux0/U50/Z (A011NSVTX1)	0.16	1.45 f
mux0/mux0/Z[10] (MUX31_n15_3)	0.00	1.45 f

mux0/mux1/C[10] (MUX31_n15_2)	0.00	1.45 f
mux0/mux1/U49/Z (AO11NSVTX1)	0.18	1.63 f
mux0/mux1/Z[10] (MUX31_n15_2)	0.00	1.63 f
mux0/Z[10] (MUX51_n15_1)	0.00	1.63 f
adder0/Z[10] (Adder3to2_1)	0.00	1.63 f
adder0/adder10/C (Adder_29)	0.00	1.63 f
adder0/adder10/halfAdder1/Y (HalfAdder_58)	0.00	1.63 f
adder0/adder10/halfAdder1/U10/Z (EOSVTX1)	0.08	1.71 r
adder0/adder10/halfAdder1/P (HalfAdder_58)	0.00	1.71 r
adder0/adder10/WS (Adder_29)	0.00	1.71 r
adder0/WS[10] (Adder3to2_1)	0.00	1.71 r
adder1/X[10] (Adder3to2_0)	0.00	1.71 r
adder1/adder10/X (Adder_13)	0.00	1.71 r
adder1/adder10/halfAdder0/X (HalfAdder_27)	0.00	1.71 r
adder1/adder10/halfAdder0/U10/Z (EOSVTX1)	0.10	1.82 r
adder1/adder10/halfAdder0/P (HalfAdder_27)	0.00	1.82 r
adder1/adder10/halfAdder1/X (HalfAdder_26)	0.00	1.82 r
adder1/adder10/halfAdder1/U10/Z (EOSVTX1)	0.09	1.90 r
adder1/adder10/halfAdder1/P (HalfAdder_26)	0.00	1.90 r
adder1/adder10/WS (Adder_13)	0.00	1.90 r
adder1/WS[10] (Adder3to2_0)	0.00	1.90 r
ws_shift_reg[13]/D (FD2QSVTX2)	0.00	1.90 r
data arrival time		1.90
clock CLK_0 (rise edge)	2.00	2.00
clock network delay (ideal)	0.00	2.00
ws_shift_reg[13]/CP (FD2QSVTX2)	0.00	2.00 r
library setup time	-0.09	1.91
data required time		1.91
<hr/>		
data required time		1.91
data arrival time		-1.90



---

slack (MET)	0.00
-------------	------



## APPENDIX C

# VHDL Code

---

```
-----  
-- Radix-2 Division  
-- s050985@student.dtu.dk  
-- Oct 18th, 2007  
-----  
  
library IEEE;  
    use IEEE.std_logic_1164.all;  
    use IEEE.std_logic_misc.all;  
    use IEEE.std_logic_signed.all;  
    use IEEE.std_logic_arith.all;  
  
Entity DIV2 is  
    port(input    : in    std_logic_vector(7 downto 0);  
          d_in    : in    std_logic_vector(7 downto 0);  
          clk      : in    std_logic;  
          reset    : in    std_logic;  
          output   : out   std_logic_vector(7 downto 0)  
    );  
end DIV2;
```

Architecture behavioral of DIV2 is

```

signal D      :   std_logic_vector(15 downto 0);
signal reg_D   :   std_logic_vector(15 downto 0);
signal not_D   :   std_logic_vector(15 downto 0);
signal zero    :   std_logic_vector(15 downto 0);
signal q_selected :   std_logic_vector(1 downto 0);
signal q_buf    :   std_logic_vector(1 downto 0);
signal d_selected :   std_logic_vector(15 downto 0);
signal WS      :   std_logic_vector(15 downto 0);
signal WC      :   std_logic_vector(15 downto 0);
signal WS_reg   :   std_logic_vector(15 downto 0);
signal WC_reg   :   std_logic_vector(15 downto 0);

signal WS_shift :   std_logic_vector(15 downto 0);
signal WC_shift :   std_logic_vector(15 downto 0);

component MUX31
  GENERIC(n : integer);
  Port (
    A : In   std_logic_vector (n downto 0);
    B : In   std_logic_vector (n downto 0);
    C : In   std_logic_vector (n downto 0);
    SEL : In  std_logic_vector(1 downto 0);
    Z : Out  std_logic_vector (n downto 0) );
end component;

component Adder3to2
  port(
    X : in  std_logic_vector(15 downto 0);
    Y : in  std_logic_vector(15 downto 0);
    Z : in  std_logic_vector(15 downto 0);
    C : in  std_logic ;

```

---

```

        WS : out std_logic_vector(15 downto 0);
        WC : out std_logic_vector(15 downto 0)

    );
end component;
component Radix2selector
    port(
        X   : in   std_logic_vector(3 downto 0);
        Y   : in   std_logic_vector(3 downto 0);
        Z   : out  std_logic_vector(1 downto 0)
    );
end component;
component REG is
    GENERIC(n : integer);
    Port ( DATA_IN      : In   std_logic_vector (n downto 0);
          RESET          : In   std_logic;
          CLOCK           : In   std_logic;
          DATA_OUT       : Out  std_logic_vector (n downto 0) );
end component;
component REG12 is
    GENERIC(n : integer);
    Port ( DATA_IN      : In   std_logic_vector (n downto 0);
          RESET          : In   std_logic;
          CLOCK           : In   std_logic;
          DATA_OUT       : Out  std_logic_vector (n downto 0);
          DATA_OUT_NOT   : Out  std_logic_vector (n downto 0) );
end component;
component Buf is
    Generic(n: integer);
    port(input  : in  std_logic_vector(n downto 0);
          output : buffer std_logic_vector(n downto 0)
    );

```

```
end component;
component OTFC2 is
    port(
        Qinput      : in  std_logic_vector(1 downto 0);
        clk          : in  std_logic;
        reset        : in  std_logic;
        Qoutput      : out std_logic_vector(7 downto 0)
    );
end component;
begin

reg0 :REG12 Generic Map(n=>15)
    port map (DATA_IN => D,
              RESET => reset,
              CLOCK => clk,
              DATA_OUT => reg_D,
              DATA_OUT_NOT => not_D
    );

reg1 :REG Generic Map(n=>15)
    port map (DATA_IN => WS_shift,
              RESET => reset,
              CLOCK => clk,
              DATA_OUT => WS_reg
    );

reg2 :REG Generic Map(n=>15)
    port map (DATA_IN => WC_shift,
              RESET => reset,
              CLOCK => clk,
              DATA_OUT => WC_reg
    );

mux00 :MUX31 Generic Map(n=>15)
    port map (A => reg_D,
```

---

```

        B => not_D,
        C => zero,
        SEL => q_buf,
        Z => d_selected
    );
adder0 : Adder3to2
    port map (X => WS_reg,
              Y => WC_reg,
              Z => d_selected,
              C => q_selected(1),
              WS => WS,
              WC => WC
    );
buf0   : Buf Generic Map(n=>1)
    port map(input => q_selected,
              output=> q_buf
    );
sele   : Radix2selector
    port map(X =>WS_reg(10 downto 7),
              Y =>WC_reg(10 downto 7),
              Z => q_selected);
otfc00 : OTFC2
    port map(Qinput => q_selected, clk => clk, reset => reset, Qoutput => output);

process(clk, reset)
begin

    if ( reset = '0' ) then
        WC_shift <= (others => '0');
        WS_shift(6 downto 0) <= input(7 downto 1);
        WS_shift(15 downto 7) <= "000000000";
        D(7 downto 0) <= d_in(7 downto 0);
    end if;
end process;

```

```
D(15 downto 8) <= "00000000";
zero <= (others => '0');

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

    WC_shift <= to_stdlogicvector(to_bitvector(WC) sll 1);
    WS_shift <= to_stdlogicvector(to_bitvector(WS) sll 1);

end if;

end process;

end behavioral;

-----
-- Radix-2 selection
-----

library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.std_logic_misc.all;
    use IEEE.std_logic_signed.all;
    use IEEE.std_logic_arith.all;

Entity Radix2Selector is
    port(
        X    :    in    std_logic_vector(3 downto 0);
        Y    :    in    std_logic_vector(3 downto 0);
        Z    :    out   std_logic_vector(1 downto 0)
    );
end Radix2Selector;

Architecture behavioral of Radix2Selector is
```



---

```
component CPA4 is
port(
    X  : in  std_logic_vector(3 downto 0);
    Y  : in  std_logic_vector(3 downto 0);
    C  : in  std_logic_vector(0 downto 0);
    WS : out std_logic_vector(3 downto 0)
);
End component;

signal select_bits : std_logic_vector(3 downto 0);

begin

cpa : CPA4
    port map( X => X,
              Y => Y,
              C => "0",
              WS => select_bits
    );

process(select_bits)
begin
    case select_bits is
        when "0000" => Z <= "01" ;
        when "0001" => Z <= "01" ;
        when "0010" => Z <= "01" ;
        when "0011" => Z <= "01" ;
        when "0100" => Z <= "11" ;
        when "0101" => Z <= "11" ;
        when "0110" => Z <= "11" ;
        when "0111" => Z <= "00" ;
```

```

        when "1000" => Z <= "01" ;
        when "1001" => Z <= "11" ;
        when "1010" => Z <= "11" ;
        when "1011" => Z <= "00" ;
        when "1100" => Z <= "11" ;
        when "1101" => Z <= "11" ;
        when "1110" => Z <= "11" ;
        when others => Z <= "00" ; --"1111"
    end case;

    end process;
end behavioral;

component Radix4Selector2 is
    port(
        X    : in    std_logic_vector(7 downto 0);
        Y    : in    std_logic_vector(7 downto 0);
        D    : in    std_logic_vector(3 downto 0);
        Z    : out   std_logic_vector(3 downto 0)
    );
end component;

component REG is
    GENERIC(n : integer);
    Port ( DATA_IN      : In    std_logic_vector (n downto 0);
          RESET          : In    std_logic;
          CLOCK           : In    std_logic;
          DATA_OUT       : Out   std_logic_vector (n downto 0) );
end component;

component REG14 is
    GENERIC(n : integer);
    Port ( DATA_IN      : In    std_logic_vector (n downto 0);
          RESET          : In    std_logic;

```

---

```

        CLOCK      : In    std_logic;
        DATA_OUT   : Out   std_logic_vector (n downto 0);
        DATA_OUT_NOT : Out  std_logic_vector (n downto 0);
        DATA_OUT_2  : Out   std_logic_vector (n downto 0);
        DATA_OUT_2_NOT : Out std_logic_vector (n downto 0)
    );
end component;

component Buf is
    Generic(n: integer);
    port(input  : in  std_logic_vector(n downto 0);
          output : out std_logic_vector(n downto 0)
    );
end component;

component OTFC4 is
    port(
        Qinput      : in  std_logic_vector(3 downto 0);
        clk          : in  std_logic;
        reset        : in  std_logic;
        Qoutput      : out std_logic_vector(7 downto 0)
    );
end component;

begin

reg0 :REG14 Generic Map(n=>15)
    port map (DATA_IN => D,
              RESET => reset,
              CLOCK => clk,
              DATA_OUT => reg_D,
              DATA_OUT_NOT => not_D,
              DATA_OUT_2 => reg_2D,
              DATA_OUT_2_NOT => not_2D
    );

```

```
reg1 :REG Generic Map(n=>15)
    port map (DATA_IN => WS_shift,
              RESET => reset,
              CLOCK => clk,
              DATA_OUT => WS_reg
    );
reg2 :REG Generic Map(n=>15)
    port map (DATA_IN => WC_shift,
              RESET => reset,
              CLOCK => clk,
              DATA_OUT => WC_reg
    );
mux00 :MUX51 Generic Map(n=>15)
    port map (A => not_2D,
              B => not_D,
              C => zero,
              D => reg_2D,
              E => reg_D,
              SEL => q_buf,
              Z => d_selected
    );
adder0 : Adder3to2
    port map (X => WS_reg,
              Y => WC_reg,
              Z => d_selected,
              C => q_selected(1),
              WS => WS,
              WC => WC
    );
buf0   : Buf Generic Map(n=>3)
    port map(input => q_selected,
              output => q_buf
```

```

        );
sele    : Radix4selector2
        port map(X =>WS_reg(11 downto 4),
                  Y =>WC_reg(11 downto 4),
                  D =>reg_D(7 downto 4),
                  Z => q_selected
        );
otfc00 : OTFC4
        port map(Qinput => q_selected, clk => clk, reset => reset, Qoutput => output);

process(clk, reset)
begin
    if ( reset = '0' ) then
        WC_shift <= (others => '0');
        WS_shift(5 downto 0) <= input(7 downto 2);
        WS_shift(15 downto 6) <= "0000000000";
        D(7 downto 0) <= d_in(7 downto 0);
        D(15 downto 8) <= "00000000";
        zero <= (others => '0');
    elsif (( clk = '1' ) and (clk'EVENT)) then
        WS_shift <= to_stdlogicvector(to_bitvector(WS) sll 2);
        WC_shift <= to_stdlogicvector(to_bitvector(WC) sll 2);
    end if;

end process;

end behavioral;

-----
-- Radix-4 Selection with comparison
-----

library IEEE;
```

```
use IEEE.std_logic_1164.all;
use IEEE.std_logic_misc.all;
use IEEE.std_logic_signed.all;
use IEEE.std_logic_arith.all;
```

Entity Radix4Selector2 is

```
    port(
        X    : in    std_logic_vector(7 downto 0);
        Y    : in    std_logic_vector(7 downto 0);
        D    : in    std_logic_vector(3 downto 0);
        Z    : out   std_logic_vector(3 downto 0)
    );
end entity;
```

Architecture Behavioral of Radix4Selector2 is

component Selector4Control is

```
port(X    : in    std_logic_vector(3 downto 0);
     M_2   : out   std_logic_vector(7 downto 0);
     M_1   : out   std_logic_vector(7 downto 0);
     M_0   : out   std_logic_vector(7 downto 0);
     M_m_1 : out   std_logic_vector(7 downto 0)
);
end component;
```

component CSA8 is

```
port(
    X : in  std_logic_vector(7 downto 0);
    Y : in  std_logic_vector(7 downto 0);
    Z : in  std_logic_vector(7 downto 0);
    C : in  std_logic ;
    WS : out std_logic_vector(7 downto 0);
    WC : out std_logic_vector(7 downto 0)
);
```

```

    );
End component;
component SD is
    port (WS   : in   std_logic_vector(7 downto 0);
          WC   : in   std_logic_vector(7 downto 0);
          S    : out  std_logic
    );
end component;
component Coder is
port(S_2   : in   std_logic;
      S_1   : in   std_logic;
      S_0   : in   std_logic;
      S_m_1 : in   std_logic;
      Q     : out  std_logic_vector(3 downto 0)
);
end component;

signal m_0   : std_logic_vector(7 downto 0);
signal m_1   : std_logic_vector(7 downto 0);
signal m_2   : std_logic_vector(7 downto 0);
signal m_m_1 : std_logic_vector(7 downto 0);
signal ws_0  : std_logic_vector(7 downto 0);
signal wc_0  : std_logic_vector(7 downto 0);
signal ws_1  : std_logic_vector(7 downto 0);
signal wc_1  : std_logic_vector(7 downto 0);
signal ws_2  : std_logic_vector(7 downto 0);
signal wc_2  : std_logic_vector(7 downto 0);
signal ws_m_1 : std_logic_vector(7 downto 0);
signal wc_m_1 : std_logic_vector(7 downto 0);
signal s_2    : std_logic;
signal s_1    : std_logic;
signal s_0    : std_logic;

```

```
signal s_m_1      :    std_logic;

begin

control : Selector4Control
    port map(X => D,
              M_2 => m_2,
              M_1 => m_1,
              M_0 => m_0,
              M_m_1 => m_m_1
    );

csa0    :    CSA8
    port map(X  => m_2,
              Y  => X,
              Z  => Y,
              C  => '0',
              WS => ws_2,
              WC => wc_2
    );

csa1    :    CSA8
    port map(X  => m_1,
              Y  => X,
              Z  => Y,
              C  => '0',
              WS => ws_1,
              WC => wc_1
    );

csa2    :    CSA8
    port map(X  => m_0,
              Y  => X,
              Z  => Y,
              C  => '0',
```



```
        WS => ws_0,
        WC => wc_0
    );
csa3  :  CSA8
    port map(X  => m_m_1,
             Y  => X,
             Z  => Y,
             C  => '0',
             WS => ws_m_1,
             WC => wc_m_1
    );
sd0   :  SD
    port map( WS => ws_2,
             WC => wc_2,
             S  => s_2
    );
sd1   :  SD
    port map( WS => ws_1,
             WC => wc_1,
             S  => s_1
    );
sd2   :  SD
    port map( WS => ws_0,
             WC => wc_0,
             S  => s_0
    );
sd3   :  SD
    port map( WS => ws_m_1,
             WC => wc_m_1,
             S  => s_m_1
    );
coder0 :  Coder
```

```

        port map(S_2 => s_2,
                  S_1 => s_1,
                  S_0 => s_0,
                  S_m_1 => s_m_1,
                  Q   => Z
        );
    end Behavioral;

-----
-- Radix-8 Division
-----

library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.std_logic_misc.all;
    use IEEE.std_logic_signed.all;
    use IEEE.std_logic_arith.all;

Entity DIV8 is
    port(input      : in   std_logic_vector(7 downto 0);
          d_in      : in   std_logic_vector(7 downto 0);
          clk       : in   std_logic;
          reset     : in   std_logic;
          output    : out  std_logic_vector(7 downto 0)
    );
end DIV8;

Architecture behavioral of DIV8 is

    signal D      : std_logic_vector(15 downto 0);
    signal reg_D  : std_logic_vector(15 downto 0);
    signal not_D  : std_logic_vector(15 downto 0);
    signal reg_2D : std_logic_vector(15 downto 0);

```

---

```

signal not_2D    :    std_logic_vector(15 downto 0);
signal reg_4D    :    std_logic_vector(15 downto 0);
signal not_4D    :    std_logic_vector(15 downto 0);
signal reg_8D    :    std_logic_vector(15 downto 0);
signal not_8D    :    std_logic_vector(15 downto 0);
signal zero      :    std_logic_vector(15 downto 0);
signal q_selected :    std_logic_vector(7 downto 0);
signal q_selected_h :    std_logic_vector(3 downto 0);
signal q_buf_h    : std_logic_vector(3 downto 0);
signal q_selected_l :    std_logic_vector(3 downto 0);
signal q_buf_l    : std_logic_vector(3 downto 0);
signal d_selected_h :    std_logic_vector(15 downto 0);
signal d_selected_l :    std_logic_vector(15 downto 0);
signal WS        :    std_logic_vector(15 downto 0);
signal WS_h      :    std_logic_vector(15 downto 0);
signal WC_h      :    std_logic_vector(15 downto 0);
signal WC        :    std_logic_vector(15 downto 0);
signal WS_reg    :    std_logic_vector(15 downto 0);
signal WC_reg    :    std_logic_vector(15 downto 0);
signal ws_shift  :    std_logic_vector(15 downto 0);
signal wc_shift  :    std_logic_vector(15 downto 0);

component MUX51
  GENERIC(n : integer);
  Port (
    A : In    std_logic_vector (n downto 0);
    B : In    std_logic_vector (n downto 0);
    C : In    std_logic_vector (n downto 0);
    D : In    std_logic_vector (n downto 0);
    E : In    std_logic_vector (n downto 0);
    SEL : In   std_logic_vector(3 downto 0);
    Z : Out   std_logic_vector (n downto 0) );
end component;

```

```
component Adder3to2
  port(
    X   : in  std_logic_vector(15 downto 0);
    Y   : in  std_logic_vector(15 downto 0);
    Z   : in  std_logic_vector(15 downto 0);
    C   : in  std_logic ;
    WS  : out std_logic_vector(15 downto 0);
    WC  : out std_logic_vector(15 downto 0)

  );
end component;

component Radix8selector
  port(
    X      : in  std_logic_vector(7 downto 0);
    Y      : in  std_logic_vector(7 downto 0);
    D      : in  std_logic_vector(3 downto 0);
    out_h   : out std_logic_vector(3 downto 0);
    out_l   : out std_logic_vector(3 downto 0)

  );
end component;

component REG is
  GENERIC(n : integer);
  Port ( DATA_IN      : In    std_logic_vector (n downto 0);
        RESET         : In    std_logic;
        CLOCK          : In    std_logic;
        DATA_OUT      : Out   std_logic_vector (n downto 0) );
end component;

component REG14 is
  GENERIC(n : integer);
  Port ( DATA_IN      : In    std_logic_vector (n downto 0);
        RESET         : In    std_logic;
        CLOCK          : In    std_logic;
```

```

        DATA_OUT      : Out   std_logic_vector (n downto 0);
        DATA_OUT_NOT   : Out   std_logic_vector (n downto 0);
        DATA_OUT_2     : Out   std_logic_vector (n downto 0);
        DATA_OUT_2_NOT : Out   std_logic_vector (n downto 0)
    );
end component;

component REG18 is
    GENERIC(n : integer);
    Port ( DATA_IN      : In     std_logic_vector (n downto 0);
          RESET         : In     std_logic;
          CLOCK          : In     std_logic;
          DATA_OUT_4    : Out    std_logic_vector (n downto 0);
          DATA_OUT_4_NOT : Out    std_logic_vector (n downto 0);
          DATA_OUT_8    : Out    std_logic_vector (n downto 0);
          DATA_OUT_8_NOT : Out    std_logic_vector (n downto 0)
    );
end component;

component Buf is
    Generic(n: integer);
    port(input  : in  std_logic_vector(n downto 0);
          output : out std_logic_vector(n downto 0)
    );
end component;

component OTFC8 is
    port(
        Qinput      : in   std_logic_vector(7 downto 0);
        clk         : in   std_logic;
        reset       : in   std_logic;
        Qoutput     : out  std_logic_vector(7 downto 0)
    );
end component;

begin

```

```
reg0 :REG14 Generic Map(n=>15)
    port map (DATA_IN => D,
              RESET => reset,
              CLOCK => clk,
              DATA_OUT => reg_D,
              DATA_OUT_NOT => not_D,
              DATA_OUT_2 => reg_2D,
              DATA_OUT_2_NOT => not_2D
    );

reg1 :REG18 Generic Map(n=>15)
    port map (DATA_IN => D,
              RESET => reset,
              CLOCK => clk,
              DATA_OUT_4 => reg_4D,
              DATA_OUT_4_NOT => not_4D,
              DATA_OUT_8 => reg_8D,
              DATA_OUT_8_NOT => not_8D
    );

reg2 :REG Generic Map(n=>15)
    port map (DATA_IN => WS_shift,
              RESET => reset,
              CLOCK => clk,
              DATA_OUT => WS_reg
    );

reg3 :REG Generic Map(n=>15)
    port map (DATA_IN => WC_shift,
              RESET => reset,
              CLOCK => clk,
              DATA_OUT => WC_reg
    );

mux0 :MUX51 Generic Map(n=>15)
    port map (A => not_8D,
```

```

        B => not_4D,
        C => zero,
        D => reg_8D,
        E => reg_4D,
        SEL => q_buf_h,
        Z => d_selected_h
    );
mux1 :MUX51 Generic Map(n=>15)
    port map (A => not_2D,
        B => not_D,
        C => zero,
        D => reg_2D,
        E => reg_D,
        SEL => q_buf_l,
        Z => d_selected_l
    );
adder0 : Adder3to2
    port map (X => WS_reg,
        Y => WC_reg,
        Z => d_selected_h,
        C => q_selected_h(3),
        WS => WS_h,
        WC => WC_h
    );
adder1 : Adder3to2
    port map (X => WS_h,
        Y => WC_h,
        Z => d_selected_l,
        C => q_selected_l(3),
        WS => WS,
        WC => WC
    );

```

```

buf0    : Buf Generic Map(n=>3)
          port map(input => q_selected_h,
                    output => q_buf_h
          );
buf1    : Buf Generic Map(n=>3)
          port map(input => q_selected_l,
                    output => q_buf_l
          );
sele    : Radix8selector
          port map(X =>WS_reg(12 downto 5),
                    Y =>WC_reg(12 downto 5),
                    D => reg_D(7 downto 4),
                    out_h => q_selected_h,
                    out_l => q_selected_l
          );
otfc00 : OTFC8
          port map(Qinput => q_selected,
                    clk => clk,
                    reset => reset,
                    Qoutput => output);
process(clk, reset)
begin

    if ( reset = '0' ) then
        WC_shift <= (others => '0');
        WS_shift(5 downto 0) <= input(7 downto 2);
        WS_shift(15 downto 6) <= "0000000000";
        D(7 downto 0) <= d_in;
        D(15 downto 7) <= (others => '0');
        zero <= (others => '0');
    elsif (( clk = '1' ) and (clk'EVENT)) then

```



```

        WS_shift <= to_stdlogicvector(to_bitvector(WS) sll 3);
        WC_shift <= to_stdlogicvector(to_bitvector(WC) sll 3);
        q_selected(7 downto 4) <= q_selected_h;
        q_selected(3 downto 0) <= q_selected_l;
    end if;
end process;
end behavioral;

```

```

-----
-- Radix-8 Division Selection
-----

```

```

library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.std_logic_misc.all;
    use IEEE.std_logic_signed.all;
    use IEEE.std_logic_arith.all;

Entity Radix8Selector is
    port(
        X      : in  std_logic_vector(7 downto 0);
        Y      : in  std_logic_vector(7 downto 0);
        D      : in  std_logic_vector(3 downto 0);
        out_h   : out std_logic_vector(3 downto 0);
        out_l   : out std_logic_vector(3 downto 0)
    );
end Radix8Selector;

```

Architecture behavioral of Radix8Selector is

```

component CPA8 is
port(
    X  : in  std_logic_vector(7 downto 0);

```

```

        Y  : in  std_logic_vector(7 downto 0);
        C  : in  std_logic_vector(0 downto 0);
        WS : out std_logic_vector(7 downto 0)
    );
End component;

signal WS_bits : std_logic_vector(7 downto 0);
signal select_bits : std_logic_vector(5 downto 0);

begin

cpa : CPA8
    port map(X => X,
             Y => Y,
             C => "0",
             WS => WS_bits
    );
process(D, select_bits)
begin
    select_bits <= WS_bits(7 downto 2);
    case D is
        when "1000" =>
            case select_bits is
                when "001100" |
                     "001011" |
                     "001010" => out_h <= "0010"; out_l <= "0010" ;
                when "001001" => out_h <= "0010"; out_l <= "0001" ;
                when "001000" => out_h <= "0010"; out_l <= "0000" ;
                when "000111" => out_h <= "0010"; out_l <= "0100" ;
                when "000110" => out_h <= "0001"; out_l <= "0010" ;
                when "000101" => out_h <= "0001"; out_l <= "0001" ;
                when "000100" => out_h <= "0001"; out_l <= "0000" ;
            end case;
        end case;
    end process;
end;

```

---

```

        when "000011" |
            "000010" => out_h <= "0001"; out_l <= "0100" ;
        when "000001" |
            "000000" => out_h <= "0000"; out_l <= "0001" ;
        when "111111" |
            "111110" => out_h <= "0000"; out_l <= "0100" ;
        when "111101" => out_h <= "0000"; out_l <= "1000" ;
        when "111100" => out_h <= "0100"; out_l <= "0001" ;
        when "111011" => out_h <= "0100"; out_l <= "0000" ;
        when "111010" => out_h <= "0100"; out_l <= "0100" ;
        when "111001" => out_h <= "0100"; out_l <= "1000" ;
        when "111000" => out_h <= "1000"; out_l <= "0001" ;
        when "110111" => out_h <= "1000"; out_l <= "0000" ;
        when "110110" => out_h <= "1000"; out_l <= "0100" ;
        when  others  => out_h <= "1000"; out_l <= "1000" ;
    end case;
when "1001" =>
    case select_bits is
        when "001110" |
            "001101" |
            "001100" |
            "001011" => out_h <= "0010"; out_l <= "0010" ;
        when "001010" => out_h <= "0010"; out_l <= "0001" ;
        when "001001" => out_h <= "0010"; out_l <= "0000" ;
        when "001000" => out_h <= "0010"; out_l <= "0100" ;
        when "000111" |
            "000110" => out_h <= "0001"; out_l <= "0010" ;
        when "000101" => out_h <= "0001"; out_l <= "0001" ;
        when "000100" => out_h <= "0001"; out_l <= "0000" ;
        when "000011" |
            "000010" => out_h <= "0001"; out_l <= "0100" ;
        when "000001" |

```

```

        "000000" => out_h <= "0000"; out_l <= "0001" ;
    when "111111" |
        "111110" => out_h <= "0000"; out_l <= "0100" ;
    when "111101" => out_h <= "0000"; out_l <= "1000" ;
    when "111100" => out_h <= "0100"; out_l <= "0001" ;
    when "111011" |
        "111010" => out_h <= "0100"; out_l <= "0000" ;
    when "111001" |
        "111000" => out_h <= "0100"; out_l <= "1000" ;
    when "110111" => out_h <= "1000"; out_l <= "0001" ;
    when "110110" => out_h <= "1000"; out_l <= "0000" ;
    when "110101" => out_h <= "1000"; out_l <= "0100" ;
    when others => out_h <= "1000"; out_l <= "1000" ;
end case;
when "1010" =>
    case select_bits is
        when "001111" |
            "001110" |
            "001101" |
            "001100" => out_h <= "0010"; out_l <= "0010" ;
        when "001011" => out_h <= "0010"; out_l <= "0001" ;
        when "001010" => out_h <= "0010"; out_l <= "0000" ;
        when "001001" |
            "001000" => out_h <= "0010"; out_l <= "0100" ;
        when "000111" => out_h <= "0001"; out_l <= "0010" ;
        when "000110" => out_h <= "0001"; out_l <= "0001" ;
        when "000101" |
            "000100" => out_h <= "0001"; out_l <= "0000" ;
        when "000011" |
            "000010" => out_h <= "0000"; out_l <= "0010" ;
        when "000001" |
            "000000" => out_h <= "0000"; out_l <= "0001" ;
    end case;
end when;
end process;
end entity;

```

---

```

        when "111111" |
            "111110" => out_h <= "0000"; out_l <= "0100" ;
        when "111101" |
            "111100" => out_h <= "0000"; out_l <= "1000" ;
        when "111011" |
            "111010" => out_h <= "0100"; out_l <= "0000" ;
        when "111001" |
            "111000" => out_h <= "0100"; out_l <= "0100" ;
        when "110111" |
            "110110" => out_h <= "1000"; out_l <= "0001" ;
        when "110101" => out_h <= "1000"; out_l <= "0000" ;
        when "110100" |
            "110011" => out_h <= "1000"; out_l <= "0100" ;
        when others => out_h <= "1000"; out_l <= "1000" ;
    end case;
when "1011" =>
    case select_bits is
        when "010001" |
            "010000" |
            "001111" |
            "001110" => out_h <= "0010"; out_l <= "0010" ;
        when "001101" |
            "001100" => out_h <= "0010"; out_l <= "0001" ;
        when "001011" |
            "001010" => out_h <= "0010"; out_l <= "0000" ;
        when "001001" => out_h <= "0010"; out_l <= "0100" ;
        when "001000" => out_h <= "0001"; out_l <= "0010" ;
        when "000111" |
            "000110" => out_h <= "0001"; out_l <= "0001" ;
        when "000101" |
            "000100" => out_h <= "0001"; out_l <= "0000" ;
        when "000011" |

```

```

        "000010" => out_h <= "0000"; out_l <= "0010" ;
    when "000001" |
        "000000" => out_h <= "0000"; out_l <= "0001" ;
    when "111111" |
        "111110" => out_h <= "0000"; out_l <= "0100" ;
    when "111101" |
        "111100" => out_h <= "0000"; out_l <= "1000" ;
    when "111011" |
        "111010" => out_h <= "0100"; out_l <= "0001" ;
    when "111001" |
        "111000" => out_h <= "0100"; out_l <= "0100" ;
    when "110111" |
        "110110" => out_h <= "0100"; out_l <= "1000" ;
    when "110101" => out_h <= "1000"; out_l <= "0001" ;
    when "110100" => out_h <= "1000"; out_l <= "0000" ;
    when "110011" |
        "110010" => out_h <= "1000"; out_l <= "0100" ;
    when others => out_h <= "1000"; out_l <= "1000" ;
end case;
when "1100" =>
    case select_bits is
        when "010010" |
            "010001" |
            "010000" |
            "001111" |
            "001110" => out_h <= "0010"; out_l <= "0010" ;
        when "001101" => out_h <= "0010"; out_l <= "0001" ;
        when "001100" => out_h <= "0010"; out_l <= "0000" ;
        when "001011" |
            "001010" => out_h <= "0010"; out_l <= "0100" ;
        when "001001" |
            "001000" => out_h <= "0001"; out_l <= "0010" ;
    end case;

```

---

```

        when "000111" |
            "000110" => out_h <= "0010"; out_l <= "0001" ;
        when "000101" |
            "000100" => out_h <= "0001"; out_l <= "0100" ;
        when "000011" |
            "000010" => out_h <= "0000"; out_l <= "0010" ;
        when "000001" |
            "000000" => out_h <= "0000"; out_l <= "0001" ;
        when "111111" |
            "111110" => out_h <= "0000"; out_l <= "0100" ;
        when "111101" |
            "111100" => out_h <= "0000"; out_l <= "1000" ;
        when "111011" |
            "111010" => out_h <= "0100"; out_l <= "0001" ;
        when "111001" |
            "111000" => out_h <= "0100"; out_l <= "0000" ;
        when "110111" |
            "110110" => out_h <= "0100"; out_l <= "1000" ;
        when "110101" |
            "110100" => out_h <= "1000"; out_l <= "0001" ;
        when "110011" |
            "110010" => out_h <= "1000"; out_l <= "0000" ;
        when "110001" => out_h <= "1000"; out_l <= "0100" ;
        when others => out_h <= "1000"; out_l <= "1000" ;
    end case;
when "1101" =>
    case select_bits is
        when "010011" |
            "010010" |
            "010001" |
            "010000" => out_h <= "0010"; out_l <= "0010" ;
        when "001111" |

```

```

        "001110" => out_h <= "0010"; out_l <= "0001" ;
when "001101" |
        "001100" => out_h <= "0010"; out_l <= "0000" ;
when "001011" |
        "001010" => out_h <= "0010"; out_l <= "0100" ;
when "001001" |
        "001000" => out_h <= "0001"; out_l <= "0010" ;
when "000111" |
        "000110" => out_h <= "0010"; out_l <= "0000" ;
when "000101" |
        "000100" => out_h <= "0001"; out_l <= "0100" ;
when "000011" |
        "000010" => out_h <= "0000"; out_l <= "0010" ;
when "000001" |
        "000000" => out_h <= "0000"; out_l <= "0001" ;
when "111111" |
        "111110" => out_h <= "0000"; out_l <= "0100" ;
when "111101" |
        "111100" => out_h <= "0000"; out_l <= "1000" ;
when "111011" |
        "111010" => out_h <= "0100"; out_l <= "0001" ;
when "111001" |
        "111000" => out_h <= "0100"; out_l <= "0000" ;
when "110111" |
        "110110" => out_h <= "0100"; out_l <= "0100" ;
when "110101" |
        "110100" => out_h <= "0100"; out_l <= "1000" ;
when "110011" |
        "110010" => out_h <= "1000"; out_l <= "0000" ;
when "110001" |
        "110000" => out_h <= "1000"; out_l <= "0100" ;
when others => out_h <= "1000"; out_l <= "1000" ;

```



---

```
end case;
when "1110" =>
  case select_bits is
    when "010101" |
      "010100" |
      "010011" |
      "010010" => out_h <= "0010"; out_l <= "0010" ;
    when "010001" |
      "010000" => out_h <= "0010"; out_l <= "0001" ;
    when "001111" |
      "001110" => out_h <= "0010"; out_l <= "0000" ;
    when "001101" |
      "001100" => out_h <= "0010"; out_l <= "0100" ;
    when "001011" |
      "001010" => out_h <= "0001"; out_l <= "0010" ;
    when "001001" |
      "001000" => out_h <= "0001"; out_l <= "0001" ;
    when "000111" |
      "000110" => out_h <= "0001"; out_l <= "0000" ;
    when "000101" |
      "000100" => out_h <= "0001"; out_l <= "0100" ;
    when "000011" |
      "000010" |
      "000001" |
      "000000" => out_h <= "0000"; out_l <= "0001" ;
    when "111111" |
      "111110" |
      "111101" |
      "111100" => out_h <= "0000"; out_l <= "0100" ;
    when "111011" |
      "111010" => out_h <= "0100"; out_l <= "0001" ;
    when "111001" |
```

```

        "111000" => out_h <= "0100"; out_l <= "0000" ;
    when "110111" |
        "110110" => out_h <= "0100"; out_l <= "0100" ;
    when "110101" |
        "110100" => out_h <= "0100"; out_l <= "1000" ;
    when "110011" |
        "110010" => out_h <= "1000"; out_l <= "0001" ;
    when "110001" |
        "110000" => out_h <= "1000"; out_l <= "0000" ;
    when "101111" |
        "101110" => out_h <= "1000"; out_l <= "0100" ;
    when others => out_h <= "1000"; out_l <= "1000" ;
end case;
when others =>                                -- "1111"
    case select_bits is
        when "010110" |
            "010101" |
            "010100" |
            "010011" |
            "010010" => out_h <= "0010"; out_l <= "0010" ;
        when "010001" |
            "010000" => out_h <= "0010"; out_l <= "0001" ;
        when "001111" |
            "001110" => out_h <= "0010"; out_l <= "0000" ;
        when "001101" |
            "001100" => out_h <= "0010"; out_l <= "0100" ;
        when "001011" |
            "001010" => out_h <= "0001"; out_l <= "0010" ;
        when "001001" |
            "001000" => out_h <= "0001"; out_l <= "0001" ;
        when "000111" |
            "000110" |

```

---

```

        "000101" |
        "000100" => out_h <= "0001"; out_l <= "0100" ;
    when "000011" |
        "000010" |
        "000001" |
        "000000" => out_h <= "0000"; out_l <= "0001" ;
    when "111111" |
        "111110" |
        "111101" |
        "111100" => out_h <= "0000"; out_l <= "0100" ;
    when "111011" |
        "111010" |
        "111001" |
        "111000" => out_h <= "0100"; out_l <= "0001" ;
    when "110111" |
        "110110" |
        "110101" |
        "110100" => out_h <= "0100"; out_l <= "0100" ;
    when "110011" |
        "110010" => out_h <= "1000"; out_l <= "0001" ;
    when "110001" |
        "110000" => out_h <= "1000"; out_l <= "0000" ;
    when "101111" |
        "101110" => out_h <= "1000"; out_l <= "0100" ;
    when others => out_h <= "1000"; out_l <= "1000" ;
    end case;
end case;
end process;
end behavioral;

```