

AN ABSTRACT OF THE THESIS OF

Ravichandran Ramachandran for the degree of Master of Science in
Electrical and Computer Engineering presented on September 2, 1994.

Title: Efficient Arithmetic Using Self-timing

Redacted for Privacy

Abstract approved: _____

Shih-Lien Lu

The recent advances in VLSI technology have facilitated feature shrinking and hence a rapid increase in the levels of integration at the chip level. This increase in the level of integration has brought along with it a host of other constraints, the most crucial being timing management and increased power dissipation. Such constraints potentially prevent the full exploitation of the increased processing power made possible by technological advances.

Timing in complex digital systems has traditionally been managed by using a global clock, controlled by which all the actions take place in lock-step. An alternative means of managing timing, called self-timing, simplifies the problems of timing management and results in a reduced power dissipation of complex digital systems. Systems designed using this self-timed or asynchronous protocol, work on a principle of hand-shaking, running at their own speed, governed by local timers and the availability of data on which to work. However, this hand-shaking introduces an overhead both in terms of hardware and computational speed.

The work presented here examines the implementation of an adder, called a Parallel Half-Adder (PHA), which gains its speed by exploiting the power of asynchro-

ny to calculate the sum. The adder has been implemented in the form of a tunable micropipeline and compared to traditional adders in terms of hardware complexity and speed. Comparable results have been obtained, implying that the overhead due to hand shaking is justified and the performance improvements due to self-timing can be fully exploited. The design of an array divider using the PHA has also been presented.

Efficient Arithmetic Using Self-timing

by

Ravichandran Ramachandran

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed September 2, 1994

Commencement June, 1995

APPROVED:

Redacted for Privacy

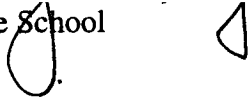
Assistant Professor of Electrical and Computer Engineering in charge of major

Redacted for Privacy

Head of Department of Electrical and Computer Engineering

Redacted for Privacy

Dean of Graduate School

Handwritten signature and initials in black ink, consisting of a large loop and a small triangle.

Date thesis is presented: September 2, 1994

Typed by Ravichandran Ramachandran for : Ravichandran Ramachandran

ACKNOWLEDGEMENT

This work is dedicated to my parents and grand parents, but for whose understanding and help it would never have been possible. I extend gratitude to Ramya Ramachandran, my loving sister, for providing unconditional support through-out.

Special thanks to Dr. Shih-Lien Lu, my major professor, for his inspiring guidance during my stay here at Oregon State University. I owe most of the skills that I picked up as a graduate student to him. Thanks to Dr. Bella Bose, my minor professor, for being available always, for discussions, and for his interest in my project. Dr. Satish Reddy, the Graduate Council Representative and Dr. Vijay Tripathi deserve special thanks for consenting to be a part of the graduate committee and for making time for this defense amidst their tight schedules. Thanks to Ms. Rita Wells for helping me expedite most of the paper work during my stay here at OSU.

My research mate Chih-Ming Chang deserves a special mention for his insightful comments on the project. Thanks to Lalit Merani an ex-member of this research group and the other members of this group for their comments, and to Shankar Pennathur for patiently reading this document and providing valuable comments.

My very special friends Karthik Ramamurthy, Sridhar Kotikalapoodi, Sudha Subbaraman and Jayanthi Chandramouli deserve special mention for listening to my tales of woe under times of stress and distress.

Thanks to the rest of the gang comprising Vasudev Tanikella, Pattamata Srinivas, Ashuthosh Kale, Manoj Kumar, Praveen Manapragada, Satish Kulkarni and Shivani Gupta for making the stay here at Corvallis memorable.

This research is funded by a National Science Foundation Grant, #MIP-9211510.

ECE Support did a wonderful job responding to my weird queries and to my complaints about the computer system here.

TABLE OF CONTENTS

CHAPTER 1.		
INTRODUCTION		1
1.1 Motivation		2
1.1.1 The Power Factor		3
1.1.2 Physical Constraints		4
1.1.3 Alleviation of the Ground Bounce Problem		5
1.1.4 Scalability		5
1.1.5 Miscellaneous Advantages		5
1.2 Disadvantages		6
1.2.1 The handshaking overhead		6
1.2.2 Race Conditions		7
1.2.3 Meta-stability		7
1.3 Organization of this document		8
CHAPTER 2.		
ASYNCHRONOUS SYSTEM IMPLEMENTATION		10
2.1 Introduction		10
2.2 Signalling schemes		11
2.2.1 The 4-cycle request-acknowledge protocol		11
2.2.2 Two Phase or Transition signalling		13
2.2.3 Comparison of the two schemes		13
2.3 Event Control Primitives		15
2.4 Micropipelines		15
2.5 Completion Signal Generation		18
2.5.1 DCVSL(Differential Cascode Voltage Switch Logic)		18
2.5.2 ECDL (Enable-Disable CMOS Differential Logic)		20
CHAPTER 3.		
ARCHITECTURE OF THE PARALLEL HALFADDER		22
3.1 Addition Schemes		22
3.1.1 Ripple Carry Addition		23

3.1.2	Conditional Sum Adder	24
3.1.3	Carry-Completion Sensing Adders	25
3.1.4	Carry Lookahead adders	26
3.2	The Parallel Halfadder	28
3.3	The Proposed Architecture	33
3.4	Including Subtraction	35
3.5	Overflow detection	36
3.6	Hardware consumption estimation	37
3.7	Computational speed of the PHA	38
CHAPTER 4.		
THE PHA: A PERFORMANCE EVALUATION		40
4.1	An Efficiency model	40
4.2	Investment on adders	41
4.2.1	The Area time model	41
4.2.2	The log of Gate count model	42
4.2.3	The Area-time-squared Model	42
4.3	Determination of Hardware complexity and computation time	43
4.4	Efficiency calculations	45
4.5	Limitations of the efficiency model	47
4.6	Improving the Efficiency of the PHA	48
4.6.1	The Tandem Adder	49
4.6.2	Hiding the Zero-Detect delay	50
CHAPTER 5.		
IMPLEMENTATION OF THE PHA		55
5.1	Implementation Strategies	55
5.2	Partitioning the adder	56
5.3	The Final Implementation	58
5.4	Circuit diagrams	58
5.5	Simulation Results	63
CHAPTER 6.		
A SELF-TIMED 16 BIT DIVIDER USING THE PHA		70
6.1	Array Division	70
6.2	The design of a 16-Bit Divider	72

CHAPTER 7.	
CONCLUSIONS AND FUTURE WORK	76
6.3 Conclusions	76
6.4 Future Work	77
BIBLIOGRAPHY	79

LIST OF FIGURES

Figure 2.1. Two blocks communicating through handshaking	10
Figure 2.2. Detailed Request and acknowledge signal propagation	11
Figure 2.3. Timing Diagram for 4 cycle handshake	12
Figure 2.4. Transition Signalling	14
Figure 2.5. A Muller-C element	15
Figure 2.6. A Micropipeline using transition signalling	17
Figure 2.7. DCVSL Gate Structure	19
Figure 2.8. Structure of an ECDL gate	21
Figure 3.1. Carry Ripple Adder	23
Figure 3.2. A 2 level 16-bit CLA adder	27
Figure 3.3. Carry Propagation for two 4-bit numbers	29
Figure 3.4. C program that simulates the PHA[20]	31
Figure 3.5. Software Simulation of the PHA: Fast Convergence	32
Figure 3.6. Software simulation of the PHA: Worst Case convergence	33
Figure 3.7. The architecture of a PHA	34
Figure 3.8. Subtraction of 65532 from 23456	36
Figure 4.1. An Adder in a pipeline	41
Figure 4.2. Computation time and gate count of adders	45
Figure 4.3. Efficiency estimation of adders	46
Figure 4.4. Alternative addition of two numbers	49
Figure 4.5. Tandem addition	50
Figure 4.6. Modified Adder to support "predict incomplete"	53
Figure 5.1. Partitioning the PHA into cells	57
Figure 5.2. The PHA captured using VIEWlogic®	59
Figure 5.3. The insides of the Controlled Register	60
Figure 5.4. The computation plane	61
Figure 5.5. CONTROL unit for the PHA	62
Figure 5.6. Including Subtraction: Carry Select	64
Figure 5.7. Simulation of the PHA using VIEWsim	65
Figure 5.8. Simulation of the PHA(Worst Case)	66
Figure 5.9. Subtraction using the PHA	67

Figure 5.10. SPICE simulation of the CONTROL circuit	68
Figure 5.11. SPICE simulation of the bitslice	69
Figure 6.1. A 16 bit array Divider	71
Figure 6.2. Design capture of the 16-bit divider	73

LIST OF TABLES

Table 3.1: Hardware consumption estimation of a n bit PHA	38
Table 4.1: Gate counts and speeds of different adders	44
Table 4.2: Gate count and computational speeds of the CLA and the PHA	48

Efficient Arithmetic Using Self-timing

CHAPTER 1. INTRODUCTION

The size and complexity of digital systems has increased tremendously in the last quarter century. This increase in complexity has been supported in part by the advances in VLSI technology. Increased processing power is a direct consequence of scaling down feature sizes in the IC process. However, this level of integration has also brought along with it a host of other constraints that potentially prevent the exploitation of this increased processing power. One of the key constraints is managing timing. Timing management has now become formidable, if not insurmountable. A ubiquitous paradigm in recent times has been the presence of a global clock which introduces a sense of discreteness into time. It is this master clock in "synchronism" with which all the actions take place in lockstep within the system. In recent years clock speed has been used as a universal distinguishing factor for seemingly identical systems. It has become a signifier of computing machismo.

It has been pointed out that the global clocking scheme has been pushed to its limit[1]. This has necessitated the need for alternate methodologies of managing timing in digital systems. One direct offshoot of this search is the advent of the "asynchronous protocol," which dictates that various blocks of a system maintain a sense of timing with respect to the blocks to which they are connected to. Variouslly called as selftimed or asynchronous or speed independent, systems designed with this protocol run at their own speed, governed by local timers and the availability of data on which to work. This in turn introduces a sense of locality in timing management that can be exploited to overcome some of the constraints imposed by global clocking.

This work examines the design and implementation of an asynchronous array divider. The underlying philosophy of this work is to provide an example of mapping an algo-

rithm which is inherently asynchronous in nature to a system which implements a globally asynchronous design philosophy and to analyze its performance.

1.1 Motivation

"We might say that the clock enables us to introduce a discreteness into time, so that time for some purposes can be regarded as a succession of instants instead of a continuous flow. A digital machine must essentially deal with discrete objects, and in case of the ACE (automatic computing engine) that is made possible by the use of a clock. All other digital computing machines except for humans and other brains that I know of do the same. One can think up ways of avoiding it, but they are very awkward. "

*Alan Turing, 1947
Lecture to the London
Mathematical Society*

The idea of using a clock in a system is as old as the first computer made. Some of the important factors that have led to the use of clocks extensively in digital systems were:

- The clock gained a symbolic importance. Seemingly identical systems could be compared based on their clock speeds.
- A lock step execution allows for a certain amount of discreteness in the system, and tracing through the operation is remarkably easy.
- It is hard to even find a metaphor on the basis of which to imagine an asynchronous system. This has led VLSI designers to use the clocked logic framework, which is easier to design with, and understand.

Going asynchronous then, means unlearning some concepts and relearning a few others. Asynchronous system implementation is not new. Digital communication systems, which are characterized by communication channel delays which are many orders greater

than the internal clocking speed of the chip, use some ingenious ways to maintain the correct sequence of operation.

The prime factors that make this relearning worth the while are enumerated below:

1.1.1 The Power Factor

As an example, An asynchronous Personal Computer's (PC) Central Processing Unit (CPU) running under maximum load will consume as much power as a synchronous CPU, but when loaded partially it expends only as much power[2]. It is a known fact that most PC CPUs are rarely loaded to their maximum capacity. This saving is made possible as only some of the units, say, the multiplier unit or the divider unit, needs to be active most of the time. Any unit not used for the execution of the particular instruction does not consume any power. At the same time when needed it can become active with minimum setup time. Generalizing this example to any system, only the active components of the system will consume power. This implies that the worst case power drawn will be equal to the synchronous case and in all other cases the power consumption is lower, assuming that the power consumed in the hand shaking interface is negligible.

The power consumption of a CMOS system depends on it's total parasitic capacitance[3]. In a globally clocked system, the clock signal should be capable of driving the load contributed by all the memory elements in the system. This is ensured by employing a tapered buffer to distribute the clock signal. A tapered buffer consists of a string of inverters, each varying in size from the immediate predecessor in the string by a constant magnification factor[4]. As an example, if a system comprising 10000 memory elements and a taper clock buffer possessing a magnification factor of m is considered, then the effective parasitic load after buffering would be :

10000 in the last buffer stage

10000/m in the preceding stage

$10000/m^2$ in the preceding stage

and so on. Totalling all these parasitic loads,

$$C_{total} = 10000 + \frac{10000}{m} + \frac{10000}{m^2} + \dots + 1 \quad (1.1)$$

Thus for a magnification factor of 4, which is typical for most clock buffers, C_{total} would be approximately 13300, which represents a 33% increase over original value. Since asynchronous systems use local clock generation strategies, rarely are such clock buffers employed. Hence the overall capacitance of such a system is potentially lower. This translates directly into reduced power dissipation.

1.1.2 Physical Constraints

Clock skew is another important consideration. The very fact that the clock is global imposes constraints like clock skew, which is the phase difference of the global synchronization signal at different locations in the system. This problem has been aggravated by increasing the level of integration in the chip. The same global signal now needs to connect to a lot more components, thus increasing the load on the signal line and hence increasing the possibility of skewing. As the various interconnected blocks in an asynchronous system communicate with each other by means of completion and other synchronization signals that are locally generated, this problem of clock skew can be virtually avoided.

The life of a chip fabricated using asynchronous timing can exceed the life of its synchronous counterpart. An argument in favour would be this simple example: An increase in clock frequency would mean reduction in rise and fall times of the clock. Reduced rise and fall times mean sharper transient currents and hence an increased problem of electro-migration. This can potentially cause the lines carrying the clock signals to snap over time. Generalizing this argument, since various blocks in an asynchronous system start working only when they have valid data available to them, and they generate control signals only on

events like completion on the interface, these transients are greatly reduced. Consequently the chip life may be increased.

1.1.3 Alleviation of the Ground Bounce Problem

Ground bounce[4], is the term applied to describe the excursion of the ground lines from their normal zero potential in a large system. Broadly described as noise, ground bounce can cause spurious switching of gates and hence incorrect operation. This problem is noticed when a large number of loads are switched simultaneously. In a self-timed system, only the parts of the system that are necessary for processing are active. Thus, on the average, the total number of loads switching is tremendously reduced in comparison to a clock driven implementation where loads always switch in response to a clock pulse. This reduction in the number of loads actively switching simultaneously, can substantially reduce the problem of ground bounce.

1.1.4 Scalability

There is a possibility that a system might need some extended functionality some where down its life cycle. In such cases, it is easy to improve an asynchronous system. In particular, pipelined systems can benefit by this. Since no global constraints exist, a new block can be added as long as it follows the hand-shaking convention. Synchronous systems on the other hand resist such midlife improvement cycles. Radical redesigning of sections may be required to incorporate the newly required functionality in a synchronous system.

1.1.5 Miscellaneous Advantages

Another esoteric advantage is that better and faster circuits can be designed by VLSI designers. As an example, the speed of an adder circuit depends on the number of car-

ries that need to be propagated for different operands. In the case of a synchronous system the worst case is taken into account in the design process, as everything should keep in step with the clock. The completion signal generation in the self-timed system can determine when carry propagation is complete, and thus potentially the process can be speeded up.

A subtle advantage of using this philosophy is that it promotes modular design. This will simplify the laying out the design at the chip level. A modular design yields very well to silicon implementation. This modularity will aid the macro-cell design methodology adopted by ASIC CAD tools. This would also mean that incremental performance gains are easier to implement. Any element or block in the critical path of a system can be replaced by a improved version without disturbing any other parts of the system.

1.2 Disadvantages

1.2.1 The handshaking overhead

One directly observable disadvantage of such asynchronous handshaking is the overhead involved in the handshaking hardware itself. Since explicit synchronizations are essential at a number of points in the system, the hardware costs associated with these could be tremendous. Another potential disadvantage could be a reduction in the throughput, since a finite amount of time is involved in ensuring synchronization. Both of these problems can be offset if systems having large computation threads between synchronization points are considered. Such systems would reduce the synchronization time and hardware costs to a small fraction of the overall computation time and hardware cost. This is a trade off which will pay off if the system is partitioned intelligently

1.2.2 Race Conditions

A change in signal values in more than one line in a combinational circuit can cause temporary errors in output values. In a feed-back free system this erroneous operation is transient, and the output reaches a stable state after a finite amount of time determined by the delay paths in the combinational network. In cases where these delays are bounded, these temporary errors can be eliminated by including all the prime implicants in the network and ensuring that only one input changes at a time[5]. However in sequential networks these can cause steady state errors. In bounded delay cases these can be eliminated by introducing redundant states and making the feedback loop delay long enough to ensure proper operation.

This poses a problem in asynchronous sequential circuits with unbounded gate delays. Use of data detectors and spacers[6] have been reported. This decreases the hardware efficiency of the system.

1.2.3 Meta-stability

Any cross-coupled gate pair potentially exhibits meta-stability. In synchronous systems, the clock rates are so tailored that the clock signal always trails the data. Another place where meta-stability is noticed is when mutual exclusion for the access of a resource is to be absolutely guaranteed. This sort of a mutual exclusion is usually handled by an arbiter. Since both synchronous and asynchronous systems carry such mutual exclusion elements, the problem of meta-stability exists in both. Ensuring that the data and the clock bear the correct relation with each other is easier to ensure and visualize in a synchronous system.

However, this problem is brought to the fore in asynchronous systems as there is no master driver like a clock with respect to which meta-stability is studied. Hence the problem is one of visualization. While it is true that an asynchronous system will have more synchronization points and mutual exclusion elements than its synchronous counterpart, this

just increases the possibility of arbiter failures and does not introduce a new problem. Hence the problem is present in both types of systems, but possibly asynchronicity increases the possibility of its occurrence. If the data generates the synchronization signal, then there would be no synchronization signal until the data is valid. This is a possible solution to the meta-stability problem and needs further scrutiny. Another solution to this problem is the use of Q Flops[7].

Weighing the advantages and disadvantages, it seems quite likely that asynchronous systems could perform better than their synchronous counterparts in the following cases. The list is indicative of the types of possible applications but is by no means exhaustive.

- Cases where power dissipation is an important consideration.
- Cases where the computation algorithm is in itself asynchronous in nature. This work is one such example.
- Cases where the clock becomes the limiting factor in the performance of the system.
- Cases where the computation threads are large in nature and hence would justify the overhead of asynchronous system implementation.

Thus asynchronous systems offer an alternative paradigm of timing management of complex digital systems. Weighing the advantages presented above it appears that this paradigm warrants further consideration. This document will study one such asynchronous system, an asynchronous parallel halfadder.

1.3 Organization of this document

In Chapter 2, we present general asynchronous system design methodologies and choose a particular implementation philosophy for the work presented here. In Chapter 3,

we explain a new addition scheme which exploits the input bit patterns to converge to a sum rapidly and the corresponding architecture necessary to implement the adder in hardware. In Chapter 4, we characterize the adder, compare it with some known implementations of adders and discuss methods to improve it. Chapter 5 presents the implementation strategies to implement the adder in silicon and presents the simulation results of addition using the adder. Chapter 6 describes the application of the adder to a non-restoring array divider. In Chapter 7 we present conclusions and indicate directions for future work.

CHAPTER 2. ASYNCHRONOUS SYSTEM IMPLEMENTATION

Timing management in an asynchronous system depends wholly on the generation of local handshaking signals. The interconnect between two blocks of a system synchronizes these signals and thus the control flow is effected. Two popular protocols governing the interconnect behavior exist. These are the transition signalling scheme and the four-phase scheme. The transition signalling scheme is also called as the two-phase or non-return-to-zero scheme. The four-phase scheme is also called the Muller signalling or return-to-zero scheme. This chapter discusses these protocols and touches on their advantages and disadvantages in general. A few important methods to generate completion signals to flag actions in an asynchronous system are also discussed.

2.1 Introduction

A high level view of two interconnected blocks and the handshaking signals that are effectively needed to control their operation is shown in Fig. 2.1.

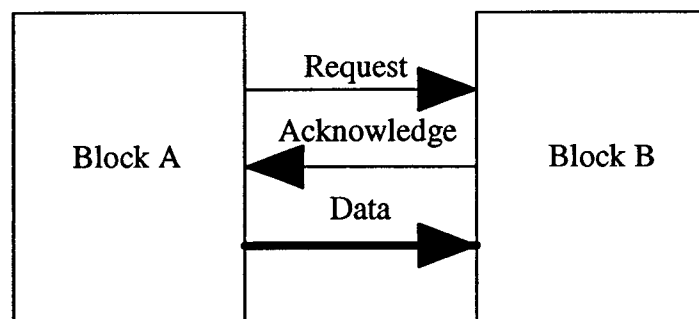


Figure 2.1. Two blocks communicating through handshaking

Block A places data on the data line and sends a request signal to block B. Block B uses the data after receiving the request signal. After the data is no longer needed in the data line, Block B produces an acknowledge signal, indicating to A that it can change data at the data line. This kind of handshaking scheme implies that A can pass on data to B but not vice-versa. A more complete representation of the handshaking scheme shown in Fig.2.2, allows two way communication between the blocks. The signals R_{in} , R_{out} , A_{in} and A_{out} indicate the handshaking signals.

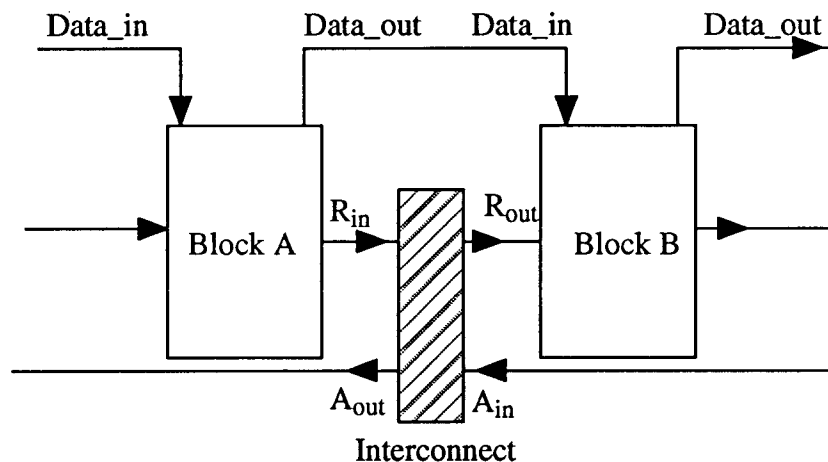


Figure 2.2. Detailed Request and acknowledge signal propagation

This *request-acknowledge* model is basic to all asynchronous systems. The order and sense of the request and acknowledge signals distinguishes different systems.

2.2 Signalling schemes

2.2.1 The 4-cycle request-acknowledge protocol

The simplest interconnection circuit is one that follows the four-phase handshaking protocol[8]. Assume all the signals R_{in} , R_{out} , A_{in} , A_{out} are all low initially (R_{in}^- , R_{out}^- , A_{in}^- , A_{out}^-). Block A after its computation raises R_{in} (R_{in}^+), indicating the availability of data

on its data line. Signal A_{in} is initially low indicating that Block B is ready to accept the data placed on the data line by Block A. The handshake circuit raises A_{out} (A_{out}^+), indicating to Block A that its output datum has been accepted and that R_{in} can be reset (R_{in}^-). The handshaking interface then raises R_{out} (R_{out}^+), indicating to Block B to begin computation. Block B finishes its computation and this information is fed back using the A_{in} signal, A_{in}^+ . This resets R_{out} which in turn resets A_{in} (A_{in}^-). This completes the four-phase handshaking scheme. Figure 2.3 depicts the timing diagram for a system following a four-phase handshaking scheme.

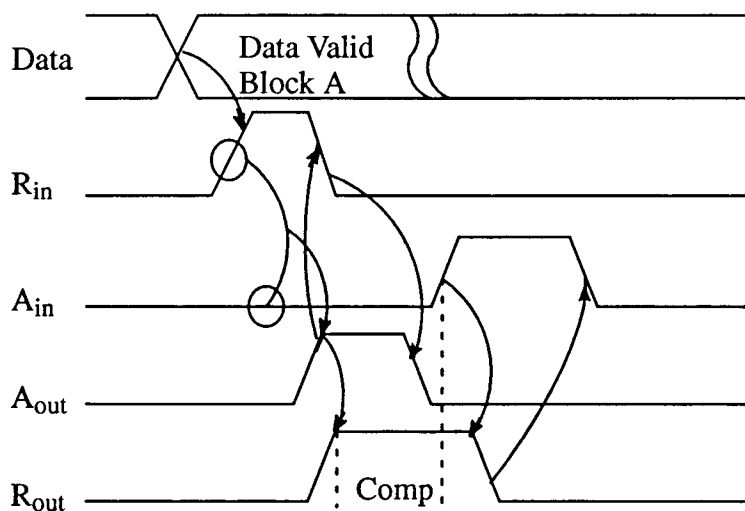


Figure 2.3. Timing Diagram for 4 cycle handshake

Computation in a four-phase scheme is always initiated on rising edge or falling edge transitions and all the signals return to '0' after the computation is complete. Thus the 4-cycle protocol is also called as the return-to-zero(RZ) signalling scheme.

This type of signalling is most compatible with logic families like ECDL[9] and DCVSL[10]. Since these logic families are differential in nature, the imbalance in the output signals when the computation is complete can be used for completion signal generation.

The signals thus generated are inherently four-phase since all logic families work on enabled and disable states.

2.2.2 Two Phase or Transition signalling

Two phase or transition signalling does not depend on the sense of the signals, in that it does not matter if the signal is at a logic high or low. All signalling is handled as events and the interconnect deals with transitions rather than logic levels. For an explanation of this scheme proposed by Ivan Sutherland [1], refer to Fig. 2.2. Initially it is assumed that all the request and acknowledge signals are at a logic '0'. Block A after its computation raises the signal R_{in} . The interconnect checks A_{in} (A_{in} is a logic '0' in this case), to ascertain if Block B is ready to accept data. This causes the signal A_{out} to be raised, acknowledging the acceptance of data. R_{out} is raised, indicating computation in Block B. Block B completes its computation and eventually produces a completion signal. This signal is fed-back in the form of A_{in} . This cycle is repeated with the control signals just toggling in state for the next cycle. It should be noted that the interconnect described above is the non-pipelined interconnect scheme. Figure 2.4. depicts the timing diagram for a transition signalling scheme.

2.2.3 Comparison of the two schemes

One direct advantage of two-phase implementation over its four-phase counterpart is that it involves lesser number of transitions at the handshaking interface. This implies that the power dissipation could be minimized. Another advantage is that, the two cycle implementation has been shown to be faster[11]. However, all the logic families known produce only four phase completion signals. Hence using a standard logic family like ECDL or DCVSL would mean that the the completion signals produced by them must be converted to two-phase. This is a hardware overhead which might be justified in large systems where

the cost of such hardware is extremely small compared to the system's overall hardware complexity.

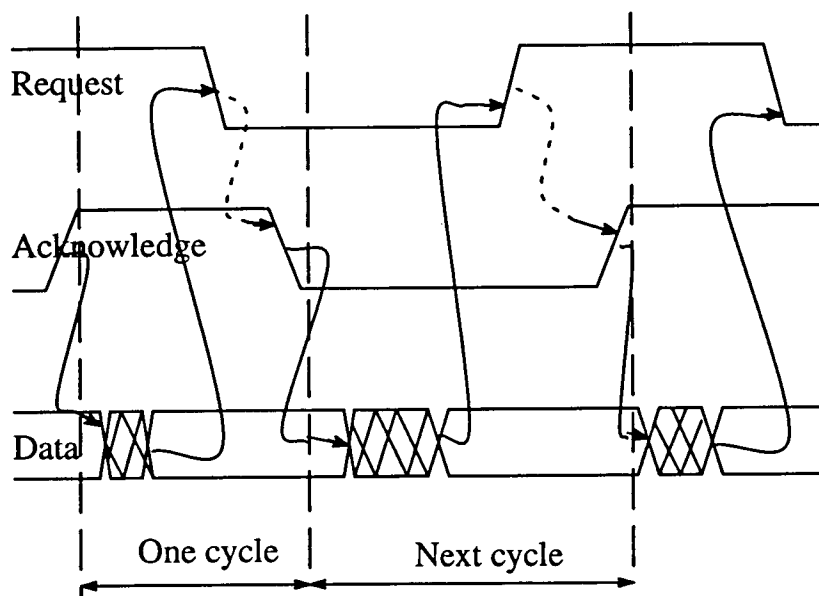


Figure 2.4. Transition Signalling

On the other hand, four-phase schemes initiate computation only on the rising edge or the falling edge. Hence, the signals need to be reset once every cycle. Some time overhead could be encountered in resetting these control signals. This increases the total number of transitions in the handshaking interface and hence increases the power dissipation. The advantage however, is that all logic is four-phase and hence the return-to-zero signalling scheme is directly compatible with logic families. It is also easier to intuitively understand a four-phase scheme as it is closely linked to logic levels. Another advantage of using four-cycle clocking is that simple edge triggered flip-flops and registers can be used for storing data, unlike in a 2 cycle case where special types of registers capable of latching on both edges of a control pulse must be designed.

2.3 Event Control Primitives

In order to handle two-phase control signals as described above, the interconnect needs some primitives capable of handling events. The most important primitives include the *event AND* and the *event OR* elements.

(1) **Event AND:** If the inputs of an *event AND* match state, it copies the state to the output. Otherwise, the previous state is maintained. Thus *ANDing* of two events is possible by using a Muller-C element. The logic symbol of a Muller C-element is shown in Fig. 2.5.

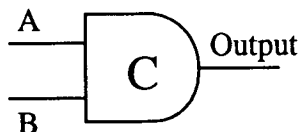


Figure 2.5. A Muller-C element

(2) **Event OR:** The OR function of events is relatively easy to implement. When either input of an XOR gate changes state, its output changes state. Hence an XOR gate can be used directly for *event OR* control.

2.4 Micropipelines

Pipelining is a common paradigm for high speed computation. Pipelining is similar to an assembly line where parts of the assembly are handled by various sections. For example, workers on an automobile assembly line perform small tasks, such as installing seat covers and fixing mirrors. The power of the assembly line comes from the fact that many workers perform small tasks to collectively produce many cars per day. Note that pipelining does not reduce the time to produce one car, but it increases the number of cars being built simultaneously and thus the rate at which cars are produced.

As applied to the ubiquitous microprocessor, pipelining is an implementation technique in which multiple instructions are overlapped in execution. For example, let an *add* instruction be encountered in an instruction mix along with several other instructions. All instructions encountered need to be fetched, decoded, executed and the results thus obtained, written back. By pipelining, it is possible to fetch the next instruction while the previous one is being decoded and so on. This increases the throughput of the system. In order to execute the *add* instruction that was encountered, an adder would invariably be used. If the adder itself could take data from only one instruction and work on it, two *add* instructions encountered one after another in the instruction mix would cause bubbles in the pipeline. The bubble results from the fact that access to the adder is delayed until it has completed the data manipulation on the previous instruction. A common solution to this problem is to design the adder so that it can have data from many instructions resident in it at any given time, and is capable of working on many such pieces of data simultaneously. This is pipelining at a level lower than the instruction level and is hence called *micropipelining*.

If the various stages of a pipeline are forced to execute in lockstep with a global clock, the result would be a synchronous pipeline. On the other hand, if each stage produced its own request and acknowledge signals, an asynchronous pipeline would result. Various asynchronous pipelined implementations exist and have been described in the literature. We recommend [1], [12] and [13] as examples. Sutherland in [1] describes a transition signalling and [12] implements it using ECDL, whereas [13] employs the four-phase scheme. We will examine the transition signalling implementation in detail as it is the strategy used for the work presented here.

What may not be obvious is that all micropipelines should follow what is known as a bundled data convention. The bundled data convention [1] requires that the control and data lines be treated as a single bundle, and data should always be available before the control signal. Fig. 2.6 depicts a micropipeline using the transition signalling framework.

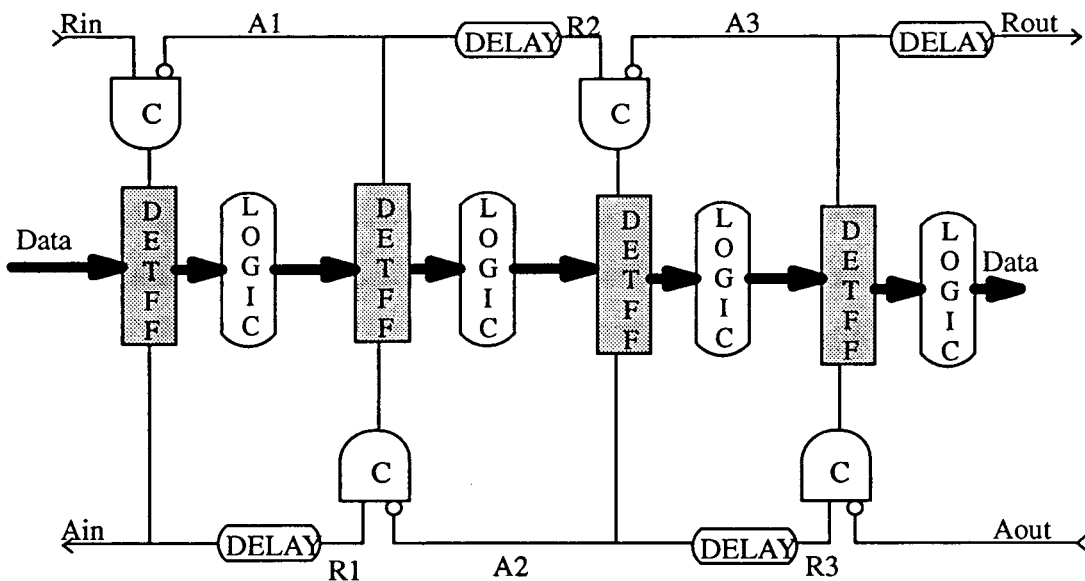


Figure 2.6. A Micropipeline using transition signalling

Before we begin a formal explanation of the signal propagation, a word about the double edge triggered flip-flop (DETFF) is in order. The double edge triggered flip-flop can be considered to be two D flip-flops arranged such that data can be latched at both the rising and falling transitions of the control signal. Efficient implementations of the DETFF have been proposed and tested by Lu [14].

The operation of the micropipeline can be explained as follows. Each loop of the control signals as can be seen from Fig. 2.6, has a single inverter and hence can oscillate. If each of the Muller C-elements is at the same initial state, then a transition in the input R_{in} will propagate all the way to the output as follows. A change in the sense of R_{in} will cause the output of the Muller C-element to change. This transition is used by the DETFF to latch the input data. The signal propagates through the DETFF and is used as the A_{in} signal for the previous stage indicating acceptance of the data. The LOGIC completes the computation. On completion of this computation, a request signal is to be generated for the next

block. The DELAY element comes into play at this time. The DELAY is adjusted in such a way that the A_{in} signal propagates through it and appears as a Request for the next stage after the logic has completed its evaluation. This signal is fed-back through the DETFF belonging to the next stage to be synchronized with the next transition at the R_{in} line.

As can be seen, the DELAY needs to be long enough to account for the LOGIC block delay. In other words the DELAY element is present to ensure that the bundled data convention is satisfied. Traditionally this DELAY element has been a problem. It is extremely difficult to predict the delay time as it depends on the depth of the logic thus complicating synthesis. Furthermore, it is very difficult to predict the behaviour of the DELAY element based on process variations. An easy solution to this problem is to eliminate the DELAY element from circuits altogether. To this end, differential logic families offer a solution. If the DELAY element is still used, it can be made a tunable DELAY. Hence variations in process parameters can be accounted for by just changing the DELAY value by tuning it, may be by changing its operating point by changing a bias voltage. The work presented here makes use of such a tunable DELAY element.

2.5 Completion Signal Generation

One of the best solutions to the meta-stability problem in asynchronous circuits is to derivation of completion signals from the data itself. This can be done if differential logic families are employed for the purpose of computation. We will discuss the process of computation signal generation using DCVSL and ECDL below.

2.5.1 DCVSL(Differential Cascode Voltage Switch Logic)

The operation of DCVSL is illustrated in Fig. 2.6. When the signal I (initialize) is low, the NMOS tree is cut off and the two output nodes get precharged. When I goes high, one of the output nodes gets discharged conditionally, depending on the NMOS logic net-

work. The key feature is the double rail coded nature of this logic. The outputs are initially pulled down to a '0' during the precharge phase. After computation is complete, the outputs take both the rails depending on the logic function evaluated. Thus simply ORing the two outputs would produce a reliable completion signal. A '1' at the output of the OR gate indicates that the logic has completed its evaluation.

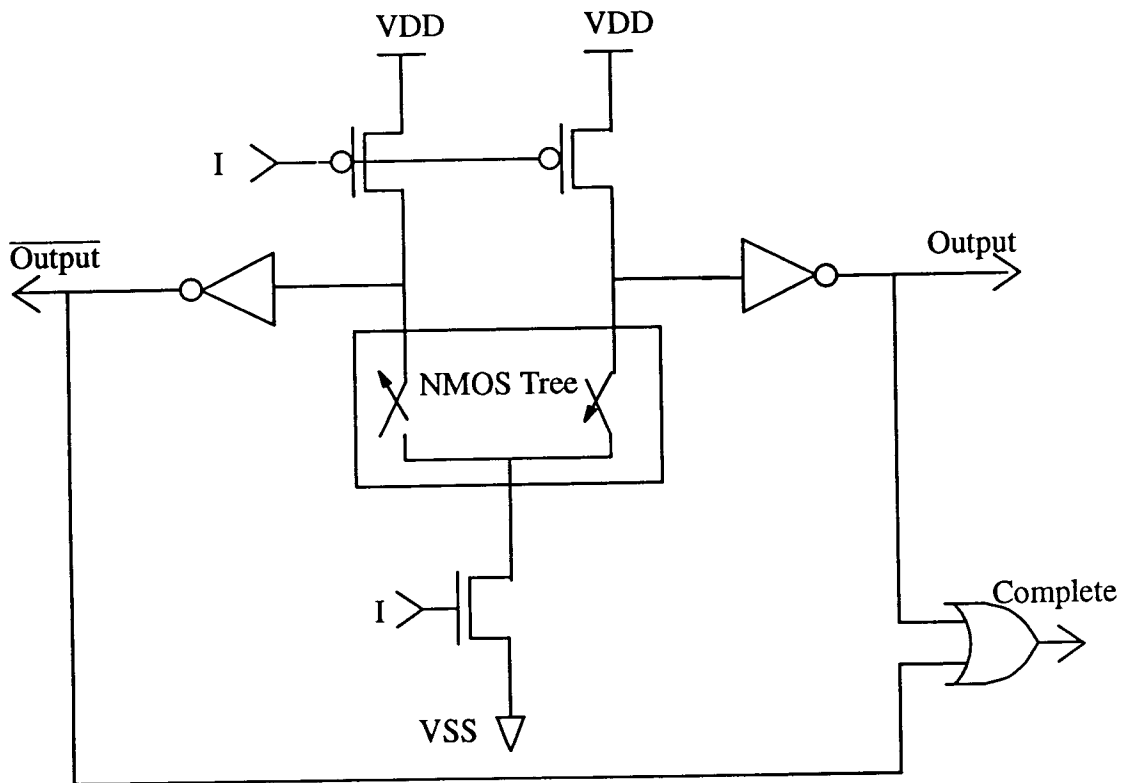


Figure 2.7. DCVSL Gate Structure

When an algorithm is partitioned into a number of cells, each cell typically comprises a number of such DCVSL cells connected in series. Only in the end of the series need there be a connection between the DCVSL gate and the handshaking circuit. Typically the R_{in} signal is connected to the input I, at the interface. Thus the DELAY element depicted in Fig.

2.6 has been effectively substituted by a DCVSL gate, which simulates the exact delay. However this completion signal is a four phase signal. Additional hardware is required to convert this to two-phase and use it in a two-phase system. The usage of the OR gate provides a hidden advantage. A finite delay time will elapse before the data is valid and the OR gate switches to indicate completion. This combined with the wiring delay, can account for the setup time of the register to which this output is connected to, in cases where this signal is used to drive the clock signal of a flipflop.

2.5.2 ECDL (Enable-Disable CMOS Differential Logic)

ECDL circuits are characterized by two distinct states – enabled and disabled, the states being determined by the transition of the control signal I . Figure 2.8 depicts the general structure of an ECDL gate. The signal I , represents the completion signal issued by the preceding stage and $Done$, is the control signal that would be issued by the current stage, after all logic processing connected with it are completed.

During the disabled state, that is, when I is at logic '1', out and \overline{out} are reset to '0'. When I is pulled down to '0' by the previous stage after it has finished its processing, this stage is enabled. Depending on the N network, the outputs get set. Since the outputs are complements of each other there will always be an imbalance between the outputs after evaluation is complete. This condition can be utilized to feed a NOR gate which produces the actual completion signal $Done$. Multiple output functions may be generated using multiple ECDL gates.

Methods to produce AND, OR and invert functions of events using ECDL gates are discussed in [12]. This paper also exemplifies micropipelines using two-phase signalling schemes but a four-phase logic family.

In differential gates such as DCVSL and ECDL, the complexity of the NMOS logic tree is usually less than twice the complexity of its single ended counterpart. This is a critical

factor, as the overhead incurred in using differential logic may not be justified if the NMOS tree is not very complex, i.e., the logic to be implemented is not deep. This is a tradeoff where gate count is traded off to get reliable completion signals.

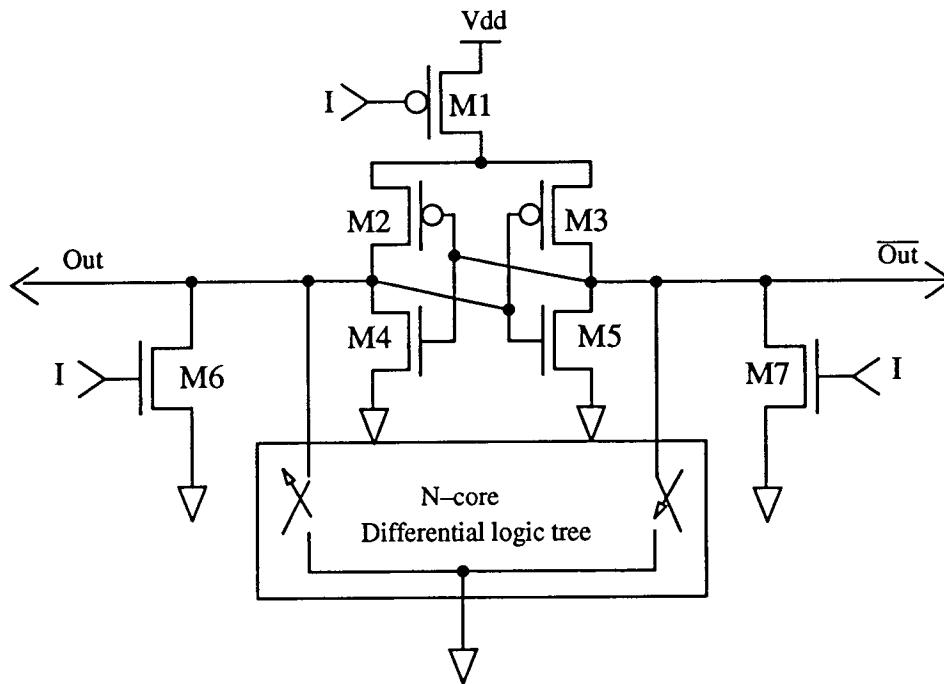


Figure 2.8. Structure of an ECDL gate

CHAPTER 3. ARCHITECTURE OF THE PARALLEL HALFADDER

The addition of two operands is the most frequent operation in any arithmetic unit. A two-operand adder is not only used for addition and subtraction, but also for multiplication and division. Consequently, an efficient two-operand adder is essential. Adders differ in the way they handle carries, and for the most part, the method employed for carry propagation is what distinguishes their performance. The two-operand adder in its simplest form, ripples a carry through out the span of its bits to perform addition. This implementation uses the least amount of hardware. Various other methods exist, which employ ingenious ways to reduce the carry propagation delay and speed up the addition process. However, this increased performance comes with an increased hardware cost.

As of even date, the fastest known adder implementation uses a *carry lookahead scheme*. Unfortunately it is also the most hardware consuming. Hence the best adder would be one, that approaches in hardware complexity, a *Ripple Carry Adder* and approaches in speed, a *carry lookahead* adder. This chapter of the report describes the key features of a *Parallel Halfadder*, which we have implemented. It also describes the features that the PHA borrows from some existing addition schemes, to make it approach the speed of a *Carry Lookahead adder* and posses the hardware complexity of a *Carry Ripple Adder*.

3.1 Addition Schemes

A vast volume of literature can be found on the subject of addition. This section of the report examines the schemes of addition which serve as good limits on the efficiency spectrum. The addition schemes discussed in this section include *Carry Ripple Addition(CRA)*, *Conditional sum addition(CSA)*, *Carry-Completion sensing Additon(CCA)*, and *Carry Look Ahead Addition(CLA)*. The interested reader is referred to [15] and [16] for more addition schemes, like *Carry Skip Addition* and a detailed explanation of what is provided

here. *Parallel Halfadder* addition which uses features from CLA, CRA and CCA will be discussed in the section 3.2.

3.1.1 Ripple Carry Addition

To add two n -bit numbers, the straightforward implementation is to have n *full adders*. A *full adder* accepts as inputs two operand bits say A_i and B_i and an incoming carry bit C_i and produces as outputs a sum, S_i and an outgoing carry C_{i+1} . This outgoing carry from stage i is the incoming carry for stage $i+1$. The corresponding logic equations implemented by the *full adder* are :

$$s_i = x_i \oplus y_i \oplus c_i \quad (3.1)$$

$$c_{i+1} = x_i y_i + c_i(x_i + y_i) \quad (3.2)$$

Figure 3.1 depicts a CRA.

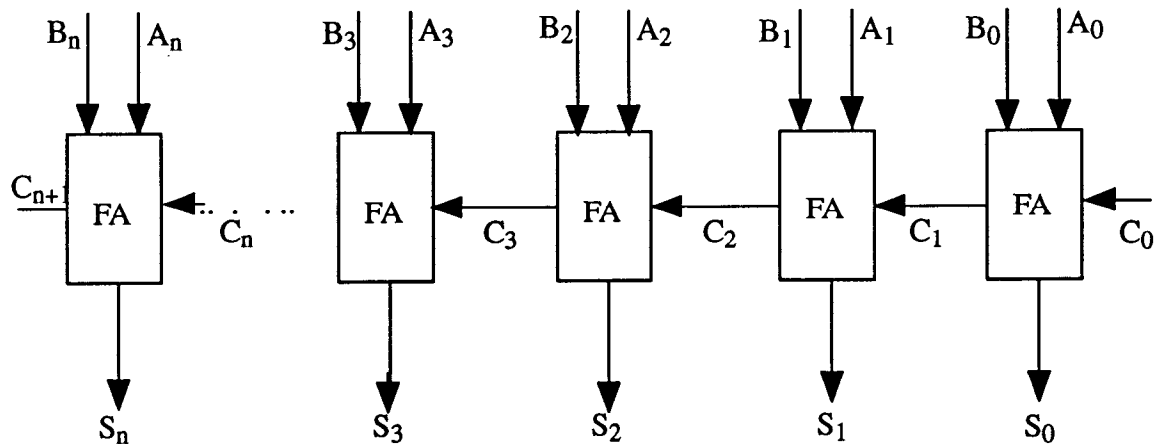


Figure 3.1. Carry Ripple Adder

The *carry-in* C_0 needs to ripple from the least significant (LSB) position to the most significant bit (MSB) position. Thus the name ripple-carry adder. If a term Δ_{FA} be defined

as the operation time or delay of a *full adder*, then to produce an n bit result the time taken by a RCA would be $n\Delta_{FA}$. The term Δ_{FA} comprises a two level delay. Namely the time to compute the term A_iB_i and $A_i + B_i$ in parallel and one gate delay to calculate $C_i(A_i + B_i)$. If one gate delay is defined by the term Δ_g then the total delay of an n bit CRA would be $2n\Delta_g$.

3.1.2 Conditional Sum Adder

Another scheme for fast addition can be attributed to Sklansky[17]. Instead of waiting for the carry to ripple through the n bits of the adder, the CSA splits the bits into groups. Each group produces two sums and carry outs, one assuming an incoming carry of 1 and another assuming an incoming carry of 0. Obviously, by just having all the n bits lumped as a single group results in a full carry propagation time. Hence the most logical way to divide the adder would be to divide the n bits into two groups of $n/2$ bits each and then again divide the $n/2$ bit group into $n/4$ bit groups and so on. This process can continue until the number of bits in the last group is one, provided n is a power of 2. Since repeated division by two is effected, the number of steps required to implement this process is $\log_2 n$. The final sum can now be obtained by just selecting from a number of sum results available based on the carry-in at the lowest level and the intermediate carry-outs produced at each level. The gate count of such a conditional sum adder has been shown to be $3n[2 + \log_2(n + 1)]$ by Sklansky in [18]. The total delay time to compute the sum of n bits has also been shown to be $[2 + 2\log_2(n + 1)]\Delta_g$. The extra hardware cost depends on the number of groups and the multiplexing cost to select one of the two results based on the carry-out in each step of the addition process.

3.1.3 Carry-Completion Sensing Adders

CCAs[16], use n full adders to sum two n -bit numbers just as a normal CRA [16]. The difference is that the *carry-out* of the $i-1$ th stage is not fed as a *carry-in* to the i th stage. Instead the *carry-in* for each stage is independently generated in the form of a carry vector.

Looking at the input vectors for addition in any adder, it is possible to predict the carry out from some stages. If both inputs for any stage are 1s, then irrespective of the *carry-in* the *carry-out* will always be a one. Such a combination is called a *generate* term. Similarly any stage which has both inputs as zeros can only produce a 0 for *carry-out* irrespective of the *carry-in*. Such a bit pattern is called a *carry sink*. Stages which have one input bit '0' and the other '1' will always pass the *carry-in* to the *carry-out*. Such stages are called *propagate* stages. It can be seen that these *propagate* stages introduce a delay in carry rippling. The *carry-outs* for other stages can be generated just from the input patterns.

Once the carry propagation is complete, the sum can be computed in $2\Delta_g$ time units, which is the time it takes for a full adder to calculate its output when all the input bits are available.

Employing the carry completion scheme speeds up the addition process as follows. The carry generate process for all the stages takes place in parallel, and some carry bits are set this way. Carry propagation must be done between any two stages where the carry has been generated. The length of the carry propagating segments depends on the input bit patterns. This yields partial parallelism. Hence the total carry propagate time for all segments will be the time it takes for the longest segment to propagate a carry. Burke, Goldstein and Von Neumann [19], have shown that the upper bound of the longest carry length of two n -bit binary numbers is given by $\log_2 n$. The ratio of the worst case to the average case increases as n becomes large. Consequently the CCA becomes a better alternative as the number of bits to be processed increases.

Such an adder is characterized by a gate count of $17n - 1$ gates. Sklansky [18] predicts that the lower bound on the delay of such an adder is given by $(n + 4)\Delta_g$.

3.1.4 Carry Lookahead adders

CLAs, carry the idea of carry generation and propagation one step further than CCAs[15]. The CCA still ripples the carry within its groups where carry needs to be propagated. The CLA does it all in parallel. We define the *Generate* and *Propagate* terms that were introduced in the previous section formally here.

$$G_i = A_i \cdot B_i \quad (3.3)$$

$$P_i = A_i \oplus B_i \quad (3.4)$$

Hence at any stage with a *carry-in* of C_i the *carry-out* can be determined to be

$$C_{i+1} = A_i B_i + C_i(A_i + B_i) \quad \text{or} \quad (3.5)$$

$$C_{i+1} = G_i + C_i P_i \quad (3.6)$$

By the same token, we have that

$$C_i = G_{i-1} + C_{i-1} P_{i-1} \quad (3.7)$$

Substituting (3.7) in (3.6), we obtain

$$C_{i+1} = G_i + G_{i-1} P_i + C_{i-1} P_{i-1} P_i \quad (3.8)$$

Further substitution yields

$$C_{i+1} = G_i + G_{i-1} P_i + G_{i-2} P_{i-1} P_i + \dots + C_0 P_0 P_1 \dots P_i \quad (3.9)$$

From equation (3.9) it can be seen that all the carries can be calculated in parallel, and hence there is no need for a carry to ripple through all the bits.

One of the main problems in using a CLA which implements Eqn. (3.9) is *fan-in*. If the number of bits becomes large, then a number of high *fan-in* gates are required, and this is a definite problem in VLSI implementation. One method to circumvent this problem is to make use of Block CLA adders(BCLA). Figure 3.2 shows a BCLA for a 16-bit 2 level *carry lookahead* implementation.

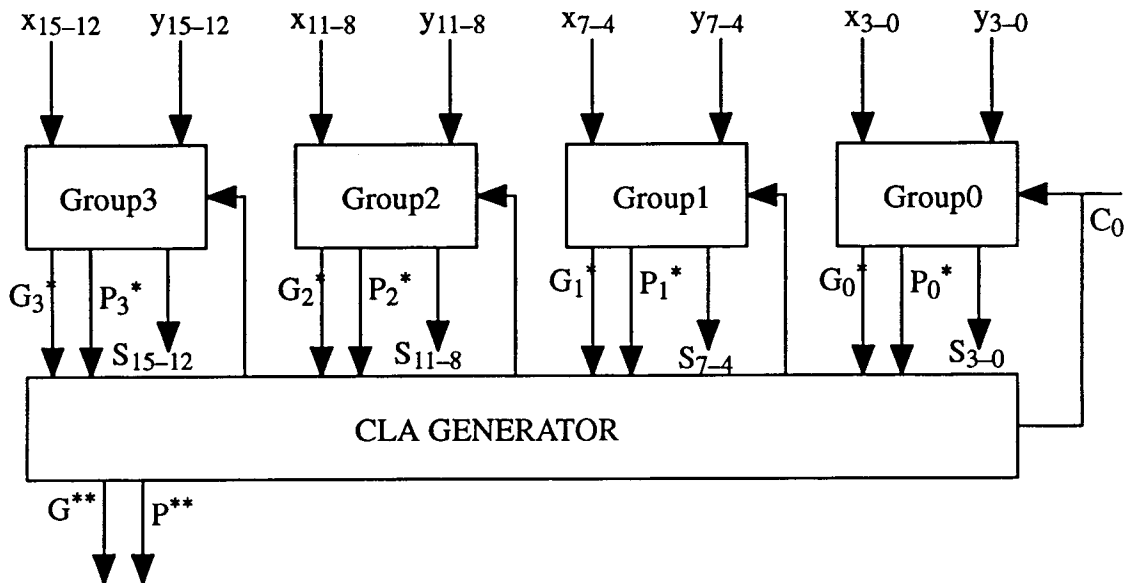


Figure 3.2. A 2 level 16-bit CLA adder

G^* and P^* indicate *group-generate* and *group-propagate* terms. The *group-generate* and *group-propagate* terms are used like *carry-ins* in single input cases. G^{**} and P^{**} indicate the *section-generate* and *section-propagate* terms. These terms can be used, if for example a 64-bit adder is to be constructed using 4 similar adders as shown in Fig. 3.2

The block CLA adder is popular among VLSI designers due to the fact that it is very modular in construction and cascading to any number of bits is easily possible. Modularity saves area in VLSI implementation as well. The speed of a *carry lookahead* adder is traded off for *fan-in* in the case of a large number of input bits. Hence the speed and gate count of a CLA adder are directly proportional to the *fan-in* allowed at the gate level. A straightforward calculation of the gate count and computation time are difficult for a CLA. Sklansky [18], has come up with the following figures for gate count and computation time. All the

calculations are based on the assumption that only 2 input gates are used.

$$\text{Gate Count} = 6n + 1 + \frac{p+1}{p-1}q + \frac{p^2+2p-1}{p} \left[k \left(n + \frac{1}{p-1} \right) - \frac{pq}{(p-1)^2} \right] \quad (3.10)$$

where

$$k = \log_p[1 + n(p-1)] - 1 \quad (3.11)$$

$$\text{and } q = 1 + (n-1)p - n \quad (3.12)$$

The computation time is given by,

$$[4 + k(p+1)]A_s \quad (3.13)$$

Having discussed a few addition schemes, we will now describe the *Parallel Halfadder(PHA)*, and list the features that it borrows from these adders to make it approach the hardware complexity of a CRA and the speed of a CLA.

3.2 The Parallel Halfadder

The original idea of Parallel Halfadder addition was conceived by David B. Swink[20]. We have developed an architecture, evaluated the hardware issues and have characterized the performance of such an adder here. The practical use of such a PHA has been exemplified by using it in a self-timed divider.

On considering the i th bit of any adder, it can be seen that four variables are of importance. These are the two operands A_i and B_i and the *carry-in* and *carry-out*, C_i and C_{i+1} respectively. Once A_i , B_i and C_i are available, the value of sum and C_{i+1} can be calculated. Let two sequences of numbers as shown in Fig. 3.3 be considered. The first bit set needs no carry propagation, and the result is available just by assuming all C_i terms are 0 and applying equation (3.1). Using (3.2) the carry out from the adder can be determined. In the second case, carry propagation must be done to some midway point, the second least significant bit in this case. Tremendous amounts of time are wasted by modelling the worst case carry propagations in both cases, i.e., by waiting, assuming that a carry will propagate from the least

to the most significant bit. The CRA waits for such carry propagation. The CLA on the other hand uses hardware to predict the carries in each stage and consequently faster.

As shown in Fig. 3.3, both the CLA and CRA are inefficient. There is neither a necessity to predict the carries nor wait for carry propagation in the first example, and both these need to be done only for a few bits in the second example. Hence a method which determines the point till which carry propagation is needed is intuitively better. The CCA makes use of this philosophy to some extent. It also uses the idea of generate and propagate terms to determine the carry vector. Since it works on producing a carry vector, it takes a finite amount of time to determine the carry vector even if no carry propagation is needed.

$\begin{array}{r} 1010 \\ + 0101 \\ \hline 1111 \\ \hline \end{array}$	$\begin{array}{r} 1 \\ \quad \downarrow \\ 1011 \\ + 0010 \\ \hline 1101 \\ \hline \end{array}$
a. No carry propagation	b. Carry dies after second bit

Figure 3.3. Carry Propagation for two 4-bit numbers

An examination of the propagate term of Eqn. (3.4) suggests that, it is, in itself a sum without a *carry-in* taken into account. So effectively in generating a propagate term, the summing portion of a half adder has been used. Two half adders can make a *full adder* and the *full adder* can produce a sum after taking the *carry-in* into account. Hence, in principle it is possible to determine the sum of $2-n$ bit numbers by initially XORing the input bits and then iteratively adjusting the final result until all the carry propagation has been accounted

for. Besides, the carry vector generated for adjustment can be continuously monitored and the point where no more carry propagation is needed can be easily determined. The parallel half adder uses the ideas mentioned above.

By using the principle of iteration in time, the most obvious advantage is the hardware saving of one *half adder* per bit of input data. The inquisitive reader might question that iteration necessitates intermediate storage, and the hardware cost of the $2n$ bit registers needed for iteration would exceed the hardware cost of n *half adders*. This is true. But, the hidden advantage of using registers, is in pipelining of the adder.

The PHA accomplishes iterative convergence by using $2n$ -bit registers, one called the SUM register, and the other the CARRY register. The register naming is appropriate, as the sum converges to its final value in the SUM register and the CARRY converges to a '0' in the CARRY register. The steps that are executed to accomplish such a convergence are as follows:

1. The *addend* and the *augend* are loaded into the two registers, SUM and CARRY.
2. The SUM and CARRY registers are simultaneously bit-wise ANDed and XORed.
3. The XORed result is routed back to the SUM register and the ANDed result is left shifted once, padded with a LSB zero and routed back to the CARRY register.
4. The CARRY register is checked. If it is a zero, the process is complete and the SUM is available in the SUM register. Else steps 2 and 3 are repeated until the CARRY register zeroes.

Figure 3.4 lists a C program that simulates this algorithm.

The zero detection signal serves as the signal to flag the operations in the adder. A zero value at the output of the zero detector signals no operation, and a non zero value indicates computation. This makes the scheme self-timed, i.e., the convergence of the adder is dependent on the input bit patterns. Such an adder is very difficult to model synchronously.

```

/*=====*/
/*===== Parallel Half Adder =====*/
/*=====++ Adapted from David. B. Swink[20] +=====*/
/*=====*/

#include <stdio.h>

main(argc, argv)
int argc;
char *argv[];
{
/* Get the command line arguments for the addend and the augend*/

    int sum = atoi(argv[1]); /* First operand */
    char mode = argv[2][0]; /* Add?Subtract */
    int carry = atoi(argv[3]); /* Second operand*/
    int sum_1 = sum;
    int carry_2 = carry;
    int temp; /* Temporary operand save */
    int i = 0; /* Counter */

    printf("Step      SUM      CARRY      RESULT  \n\n");
    while (carry != 0)
        {
            temp = sum ^ carry;
            carry = (sum & carry) << 1;
            sum = temp;
            printf("%2d : %7X + %9X = %9X \n",
                ++i, sum, carry, sum+carry);
        }
}

```

Figure 3.4. C program that simulates the PHA[20]

The operation of the PHA can be understood as follows. The bit wise XORing of the operands produces a sum without taking into account the *carry-in* at any stage. When the *carry-in* to the adder is zero, the only carries that need to be accounted for are the ones that are internally generated. Such internally generated carries per iteration are determined by

the bit wise ANDing of the operands. This internally generated carry is repeatedly adjusted until all the carry propagation is completed. The bit wise XORing of the SUM and left shifted CARRY register adjusts this. The end of adjustment is signified by the zeroing of the CARRY register.

Figures 3.5 and 3.6 show the computation results obtained by compiling and executing the code of Fig. 3.4. Figure 3.5 illustrates a fast case in which no carry propagation is essential and the result converges early. The second example, is one in which the carry needs to propagate for a long distance and hence the number of iterations required for convergence are large.

On an average, the carry propagation for n bit operands has been proved to be equal to $\log_2 n$ [19]. We have run software simulations and determined the average number of iterations required to attain convergence for 16-bits to be equal to 3.89. Hence, if the delays encountered per iteration are small, theoretically it is possible to achieve an average convergence time comparable to a carry lookahead adder. The following section presents the architecture of the PHA. Based on the architecture, it will be possible to calculate the per iteration delay and the per bit hardware cost associated.

Step	SUM		CARRY		RESULT
0 :	3042	+	8704	=	B746
1 :	B746	+	0	=	B746

Figure 3.5. Software Simulation of the PHA: Fast Convergence

Step	SUM		CARRY		RESULT
0:	2E	+	7FD3	=	8001
1:	7FFD	+	4	=	8001
2:	7FF9	+	8	=	8001
3:	7FF1	+	10	=	8001
4:	7FE1	+	20	=	8001
5:	7FC1	+	40	=	8001
6:	7F81	+	80	=	8001
7:	7F01	+	100	=	8001
8:	7E01	+	200	=	8001
9:	7C01	+	400	=	8001
10:	7801	+	800	=	8001
11:	7001	+	1000	=	8001
12:	6001	+	2000	=	8001
13:	4001	+	4000	=	8001
14:	1	+	8000	=	8001
15:	8001	+	0	=	8001

Figure 3.6. Software simulation of the PHA: Worst Case convergence

3.3 The Proposed Architecture

The asynchronous PHA design, when looked at from a high level, needs 5 control signals in addition to the data lines. These are R_{in1} the request signal that comes in bundled with the data line containing the addend, R_{in2} the control signal bundled with the augend data, A_{out} the acknowledge issued by the succeeding stage indicating that the previous result produced by the adder has been consumed, R_{out} the signal produced by the adder to the next stage indicating that that the sum bits are available and A_{in} the signal produced by the adder indicating that the input operands have been consumed.

Figure. 3.7 depicts the architecture of the adder. The two registers, SUM and CARRY, are $n+1$ bit wide for 2 n -bit operands. The $n+1$ Th. bit is used for producing the final *carry-out* from the adder. The SYNC block is used to detect if all inputs are available and the previous output produced by the adder has been consumed by the adder's succeeding

stage. The zero detector is used to determine the state of the CARRY register. The same signal can be used to select if new data needs to be latched, or if the intermediate iteration results are to be recirculated. The CONTROL block generates the internal clocking and control signals for the SUM and CARRY registers using the zero detection signal. The COMP block, which comprises a plane of XOR and AND gates, does the actual computation. The result of the addition is available from the SUM register after the carry register zeroes. The $n+1$ th bit of the SUM register holds the carry of the addition operation. Complete circuits to implement the adder are provided in Sec. 5.4.

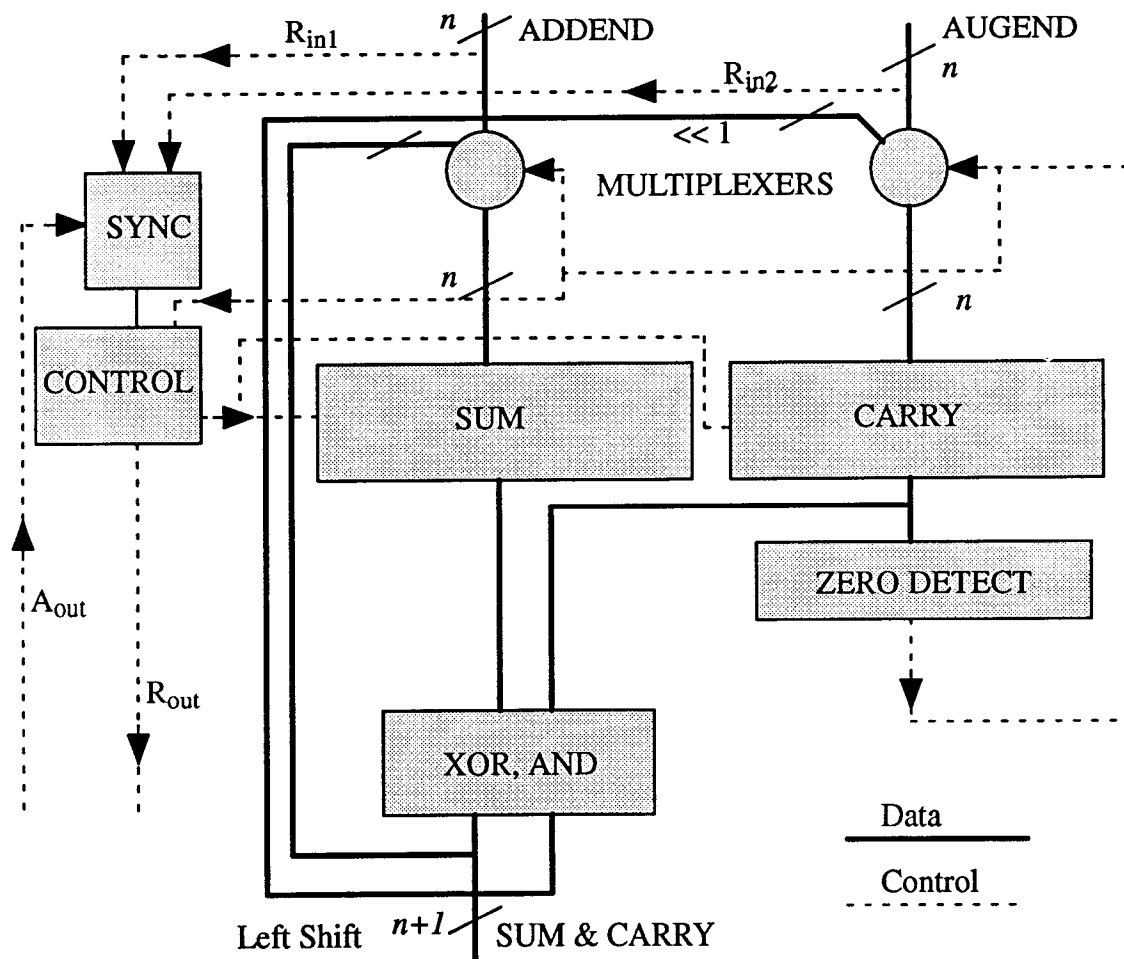


Figure 3.7. The architecture of a PHA

With the block level description as provided above, it is possible to calculate the hardware cost required to implement the adder. An approximate figure for per iteration time delay can also be estimated.

3.4 Including Subtraction

The PHA implements 2's complement addition. Hence, to subtract one operand from the other, the subtrahend needs to be complemented, and a '1' added to the LSB. To allow this enhancement, a new one bit register, namely the MODE register needs to be added. The peculiar problem surrounding subtraction here is the fact that the PHA does not support a *carry-in* along with the two operands. If such a *carry-in* were present, the problem of subtraction would be reduced to complementing one of the operands, forcing a *carry-in* and performing a normal subtraction. Since this is not possible in the case of the PHA, it is necessary to introduce the *carry-in* in the second iteration. This is done by introducing a '1' during the carry shifting in the second cycle. This poses two problems. The first is the introduction of the carry exactly in the second cycle, this being an asynchronous system. We have solved the problem by using a state machine which produces the selection signals for a MUX, which selects either a '0' or a '1' for shifting in depending on the MODE register and the output of the said state machine. The second problem is the more crucial of the two. Since the carry introduction is delayed, in the case of some bit patterns it is possible that the CARRY register zeroes in the first iteration. The zero detector would then detect the process as complete and the SUM register would actually not have converged to the right result. We have solved this problem by modifying the zero detector and the control logic. These are shown in the chapter on implementation in Figs. 5.2 and 5.5. The rest of the process remains the same. This will complete the operation of subtraction.

However, the final *carry-out* produced by the $n+1$ Th. bit of the adder must be inverted to determine the *carry-out* to the next stage. This can be done by selectively inverting the *carry-out* depending on the value contained in the MODE register. Translated into hard-

ware, this warrants the inclusion of an XOR gate controlled by the MODE register and the $n+1$ Th. bit of the sum output.

Inclusion of subtraction is particularly relevant here, since division of two numbers is performed by repeated subtractions. Figure 3.8 shows the software simulation of the adder with this subtraction capability included. We chose this example as it demonstrates the problem of zeroing before actual convergence as discussed above.

Step	SUM	CARRY	RESULT
0 :	5BA0	- FFFC	= 15B9C
1 :	FFFF5BA2	- 2	= FFFF5BA4
2 :	FFFF5BA0	- 4	= FFFF5BA4
3 :	FFFF5BA4	- 0	= FFFF5BA4

Figure 3.8. Subtraction of 65532 from 23456

3.5 Overflow detection

Since the PHA does not propagate carries horizontally, the problem of detecting an overflow is noticed. The overflow detection mechanism usually uses an XOR gate connected to the carry-in of the last stage and the carry-out produced from the last stage. Hence the problem of overflow detection is reduced to a problem of determining the carry-in at the last stage in the case of the PHA, since the carry-out is already available.

Since the final sum is available after the adder has converged, and the original input bits are known, determining the carry-in is reduced to a simple combinational logic solution.

3.6 Hardware consumption estimation

In the architecture of Fig. 3.7, the only blocks that are black boxes are the SYNC and the CONTROL blocks. The rest of the blocks are self explanatory. Hence, $2n$ multiplexers, $2n + 2$ registers, $n + 1$ XOR gates and $n + 1$ AND gates are essential. The zero detector is usually a NOR circuit and until a more efficient method is presented in this report, it can be assumed to be composed of a tree of NOR and NAND gates to compose a high *fan-in* NOR gate. The corresponding hardware investment would be $n - 1$ gates. The control logic in our implementation uses 22 discrete gates and its construction is presented in Fig. 5.5. The gate count for CONTROL block remains a constant irrespective of *fan-in*. This gate count is only for the adder and the gates needed to control subtraction have not been included. This is a reasonable assumption as we are providing comparison data for adders and we have indicated the methods to complete subtraction only to present a complete analysis of all cases.

Table 3.1 summarizes and totals the hardware cost associated with the implementation of the PHA.

The main advantage of using the adder in a micropipeline is that the storage for every pipestage is available for free. The SUM and CARRY registers provide the needed intermediate storage. Since this adder is designed for operation in a micropipeline, where storage always alternates processing, the hardware cost associated with the registers can be ignored as the registers needed to implement the micropipeline can be eliminated.

For an n -bit CRA, the hardware cost associated is $7n$ gates[16]. From Table 3.1, it can be seen that the number of gates necessary to implement a PHA are only marginally higher than that of a CRA. This matches our objective of approaching the CRA in terms of hardware complexity. The other half of the objective that still needs to be met is to rival the CLA in terms of speed. The following section estimates the speed of the PHA.

3.7 Computational speed of the PHA

From Fig. 3.7, it can be seen that the following blocks form a part of the PHAs critical path.

- The Data Multiplexers
- The Registers
- The Zero Detector
- The Computational Plane comprising XOR and AND gates
- The Control unit generated delay

Figure 3.7 also shows that the computation plane and the zero detector work together. Since the zero detector is the slower of the two, the computational delay can be left out from the calculation of the critical path. Assuming a *fan-in* of 6 for each gate in the zero-detector, the propagation time associated with a n bit zero detector is $\log_6 n$. It should be noted that this value is different than the value used for hardware complexity calculation. The worst case hardware was assumed. As will be seen later in Chapter 5, the CONTROL unit uses a multiplexer similar to the data multiplexer used in the datapath. It will also be seen that both the multiplexers are controlled by the same control signal. Hence, the design is datapath limited, and it is enough to account for the logic delay of one multiplexer in the critical path calculation.

Description	No. Of Gates
Computation (XOR and AND gates)	$2n + 2$
Zero Detect (Worst Case assuming a binary tree implementation of Zero Detection NOR gate)	$n - 1$
Multiplexers to select Data	$4n$
Control Logic (One time Overhead)	21
Total Number of Gates for a n bit adder	$7n + 22$

Table 3.1: Hardware consumption estimation of a n bit PHA

Assuming that the operand bits are uniformly distributed over their range, it has been pointed out by Burks et. al. [19], that the average longest chain carry propagation is $\log_2 n$.

The critical path of the PHA includes the following delays:

Multiplexing	: $2\Delta_g$
Latching using a Double edge triggered flip-flop	: $1ns^1$
Zero Detection using 6 input NOR gates	: $\log_6 n$

Hence the total critical path is given by

$$(2 + \log_6 n)\Delta_g + 1) \log_2 n. \quad (3.14)$$

The computation time of the CLA, as indicated in section 3.1.4, is of the order of $\log_2 n$. Equation (3.14) suggests that the PHA is higher by a factor of $\log_6 n$. Hence a cursory analysis would suggest that the PHA will progressively lag the CLA in speed as *fan-in* increases. Since the hardware cost and computation time of the PHA are known at this time, and similar values are available for all the adders discussed earlier in this chapter, a comparison of these parameters can be made.

Characterization of any adder should at the minimum, include the area investment, its computation speed and its processing power. The other factors that need consideration include the modularity or the ease with which the adder can be implemented in VLSI, its power dissipation and the number of transitions that are necessary to converge to a result. In chapter 4 we will present efficiency equations based on the minimum considerations stated here. Using these equations, we will further try to compare the PHA with all the adders presented in this chapter.

¹ For a 1.2μ Technology

CHAPTER 4. THE PHA: A PERFORMANCE EVALUATION

Having calculated the hardware complexity and the computation time of the PHA in the previous chapter, we will introduce an efficiency model here to analyze its performance. We will also compare the PHA to the other adders presented in the previous chapter.

All efficiency and hardware complexity estimation will be done assuming that the adders are used in a pipeline. At this point pipelining implies two things. On the one hand, each bit of the adder may be a stage of a pipeline and each stage comprising a bit may produce completion signals after processing. This is the lowest level of pipelining. On the other hand, all the bits of the adder are lumped and the adder is treated as a single pipestage with no pipelining internal to its implementation. It is these types of adders to which we direct our attention to in this study. The structure of an adder in a micropipeline is shown in Fig. 4.1.

4.1 An Efficiency model

The computational efficiency of an adder is directly dependant on two factors, namely the investment I on the adder and its computing power n [18]. The computing power determines the operating range of the adder. The investment I is a function of the total hardware complexity ϕ and the total addition time ρ .

$$\eta = \frac{n}{I} \quad (4.1)$$

Calculation of the investment on the adder constitutes an interesting problem. The two terms in the function for investment, namely the total addition time ρ and the hardware complexity ϕ differ from each other by several orders of magnitude. Typically, ρ is in the order of a few nano seconds and the gate count is of the order of several hundreds. Hence a function to calculate the investment on the adder needs to be determined.

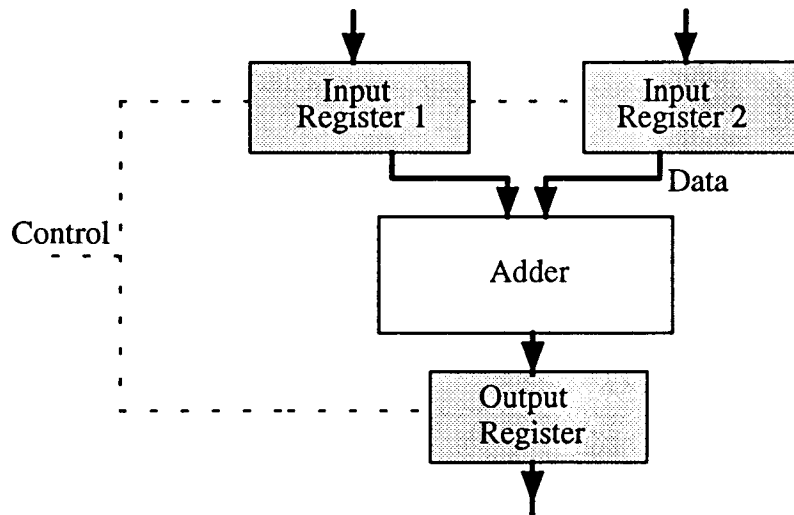


Figure 4.1. An Adder in a pipeline

4.2 Investment on adders

Several different methods exist to calculate the investment term I in the efficiency equation of the adder. The best function to calculate the investment is yet to be determined and only approximate functions are available. We will try to indicate the various methods of calculating I described in the literature, and calculate the efficiency of the PHA using all the methods that can produce indicative results for efficiency calculations.

4.2.1 The Area time model

The area time model due to Sklansky[18] is perhaps the simplest function that can be used to determine investment, and is the product of area and time. Unfortunately this is also the most inaccurate measure. Since the gate count is several thousand times greater than the addition time, the product of area and time when determined seems to place undue bias on the gate count. Consequently the *Ripple Carry Adder* seems to perform best, when effi-

iciencies are calculated using this model. This is untrue. Many VLSI designers are willing to trade off area for speed recently. This necessitates the investigation of other models to determine investment.

4.2.2 The log of Gate count model

Sklansky [18] has discussed the gate count and time delay of a number of popular implementations of adders and we summarized the same in the preceding chapter. Assuming only two input gates, it has been proposed that the area factor be described by $\log_2 G$, where G is the raw gate count required to implement the adder. Thus, using Sklansky's model to calculate the adder efficiency, equation (4.1) becomes:

$$\eta = \frac{n}{\rho \log_2 G} \quad (4.2)$$

Sklansky claims that the figures of efficiency obtained using this model seems to compare favorably with the proclivity of VLSI designers to invest.

As the *fan-in* of any adder increases, so does its gate count. Since a logarithmic scaling of gate count is effected, this model seems to be biased towards speed and attributes lesser importance to gate count. Hence, we feel that this is not a precise estimate for investment.

4.2.3 The Area-time-squared Model

Another meaningful estimate of the investment on the adder can be obtained by the product of the hardware complexity and the square of the computation time. The area-time-squared bound is based on the information flow within a chip[21]. The application of this bound modifies equation (4.1) as follows:

$$\eta = \frac{n}{\rho^2 G} \quad (4.3)$$

The Area-time-squared bounds have been extensively used for the comparison of complexity of algorithms for VLSI implementations[21].

All the models presented above require the computation time and the hardware complexity of adders to determine efficiency. The following section delves into the details of calculations of these parameters.

4.3 Determination of Hardware complexity and computation time

The time that an adder takes to compute a result is the time difference between the instant when both the data inputs are available and the instant when the calculated result is available. For asynchronous implementations this represents the time lag between the instant when both R_{in1} and R_{in2} are available and the instant when R_{out} is issued. For a synchronous system this is measured as a fixed number of clock cycles.

However, there exists no hard and fast rule which can be used to determine the hardware complexity of adders. The usual method to determine this parameter has been the gate count. Gate count is not an accurate indicator of VLSI real estate since it ignores wiring areas. Besides, modularity of the design is also important to come up with a dense implementation with more gates per unit area. In fact, it is possible that a smaller area results from an implementation with more number of gates, but with simpler interconnections and more modularity, than one with a fewer number of gates but more complex interconnections. Assuming a design implemented using a standard cell library, typically the routing area would be as much as the cell area itself. This is a very valid assumption as the standard cell library philosophy is followed by most CAD tools for auto routing. Hence it is reasonable to multiply the gate count in any design by 2 to arrive at the approximate hardware complexity. The multiplied gate count shall be denoted by G .

Lu [9] has proposed a technique which takes into account the transistor count as an indicator of the hardware complexity. An additional parameter called the wiring equivalence number has been used as an indicator of the the number of non-local signals that need to be

connected to a particular logic cell. All the calculations for CRA, CLA, CSA and CCA have been done for ECDL.

However, our discussion will be limited to the gate level, and the gate count and wiring area will be used as indicators of hardware complexity.

Table 4.1. indicates the functions used to calculate the gate count and efficiencies of various adders. The pure gate count shown in table 4.1 will be multiplied by 2 to get an adjusted value to account for the wiring areas.

Type Of Adder	2 input gate count	Computational Delay Time
Ripple Carry Adder	$7n$	$2n\Delta_g$
Carry Completion Adder	$17n - 1$	$(n + 4)\Delta_g$
Conditional Sum Adder	$3n[2 + \log_2(n + 1)]$	$[2 + 2\log_2(n + 1)]\Delta_g$
Parallel Half Adder	$17n + 22$	$\log_2(n) (1 + \log_6(n) \cdot \Delta_g)$
Carry Lookahead Adder	$6n + 1 + \frac{p + 1}{p - 1}q + \frac{p^2 + 2p - 1}{p}$ $\cdot \left[k \left(n + \frac{1}{p - 1} \right) - \frac{pq}{(p - 1)^2} \right]$ <p>where,</p> $k = \log_p[1 + n(p - 1)] - 1$ <p>and</p> $q = 1 + (n - 1)p - n$	$[4 + k(p + 1)]\Delta_g$

Table 4.1: Gate counts and speeds of different adders

The variables in all the calculations represent the following:

n : The number of bits processed by the adder

Δ_g : The propagation delay time of a single gate

All formulae assume two input gates. This simplifying assumption has been made so that a common platform for comparison may be established and the figure of efficiency does not

get distorted due to unbalanced improvements due to different implementations. Figure 4.2 depicts these relations graphically.

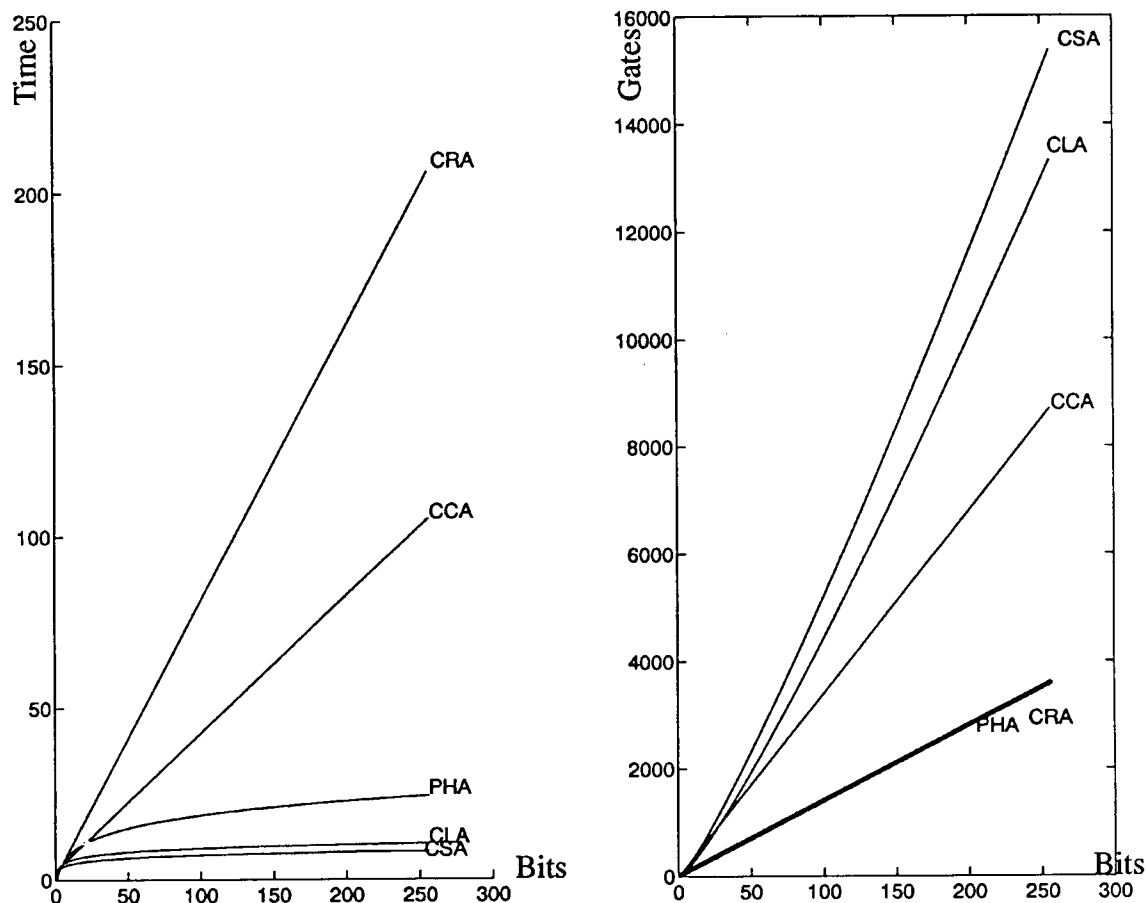


Figure 4.2. Computation time and gate count of adders

4.4 Efficiency calculations

The confusion surrounding the accurate judgement of the investment on adders, has led us to use more than one model to calculate efficiency. None of the models presented in this section are perfect and they serve only as general indicators of the efficiency trend.

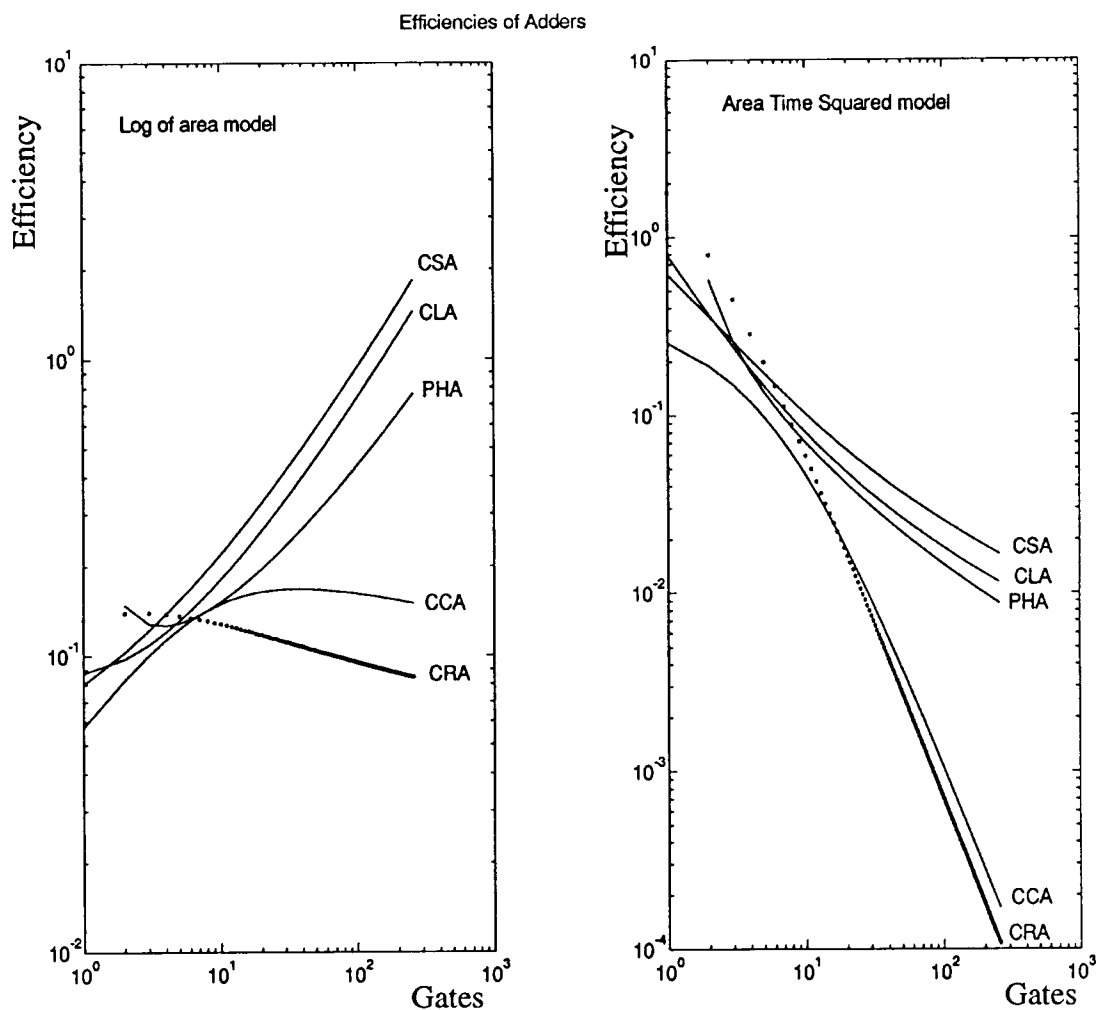


Figure 4.3. Efficiency estimation of adders

Using the above values of gate counts after adjustments, Fig. 4.3 shows the curve for efficiency of various adders with inputs varying from 1 to 256 bits using the area time squared model and the log of gates model. The range from 1 to 128 bits is the most critical, since we do not foresee a need for a *fan-in* more than that in the near future. The rest of the curve is just provided to indicate the general trend.

Using both the models the standings are the same. The CSA seems to be the most efficient and the CRA the least efficient. The PHA is positioned in the center of the spectrum

in both cases. The factor ruining the efficiency of the PHA is the fact that its speed is of the order of $(\log_2 n) \cdot (\log_6 n)$, the second part of the term, $\log_6 n$, being contributed by the zero detection network. However the efficiency models are not perfect and have their drawbacks, some of which are listed in the following section of the report.

4.5 Limitations of the efficiency model

One fact that stands out from Fig. 4.3 is that the CSA enjoys the number one position in the efficiency spectrum using both the models of efficiency calculations. Yet it is not the most popular adder that is used by VLSI designers. This highlights one of the drawbacks of both the efficiency models used here. The efficiency models are not sensitive to layout issues like modularity.

The model for efficiency used here does not take into account special architectural features like implementation using various logic families and using high efficiency and high *fan-in* gates. Different logic families produce different types of improvements or degradation of performance. The PHA will gain considerable speed by using a logic family like ECDL for its zero completions and the CLA will gain some speed if high *fan-in* gates are used. These effects cannot be accounted for in our efficiency calculations.

Intrinsic gate delays have been used and the wiring delays have been ignored. This fact is susceptible of a ready explanation. Wiring delays become important only in cases where a global signal needs to be routed to different parts of the system. This would typically be the case only for control signals which run over long spans within the chip. Since this discussion centers around the logic processing throughput, it is reasonable to ignore the wiring delays which usually are a small fraction of the logic delay. It is still an approximation.

4.6 Improving the Efficiency of the PHA

Table 4.2 shows the values of computation time and gate count for 8,16, 32, 64, 128 and 256 bit adders. For lower values of *fan-in* the speeds of the CLA and PHA are comparable and the efficiency of the PHA seems to be aided by a smaller gate count compared to the CLA and a comparable computation time. But as the *fan-in* increases the efficiency seems to depend solely on the computational speeds. The computational speed of the PHA is given by:

$$\log_2 n (1 + (\log_6 n + 2)\Delta_p) \quad (4.4)$$

n	Computation Time		Gate Count	
	CLA	PHA	CLA	PHA
8	5.5175	4.7408	98	78
16	6.7811	7.0948	236	134
32	8.0587	9.8357	559	246
64	9.3435	12.9634	1297	470
128	10.6319	16.4779	2960	918
256	11.9921	20.3793	6663	1814

Table 4.2: Gate count and computational speeds of the CLA and the PHA

The first component of this delay, $\log_2 n$, is contributed by the number of iterations that need to be performed by the adder on an average, to attain convergence and the term $\log_6 n$, is contributed by the zero-detection delay. This is implementation sensitive and the assumption underlying Eqn. (4.4) is that a zero-detector uses gates of *fan-in* equal to 6. Hence, methods to speed up the adder should take both these facts into consideration and reduce either or both of them to implement a faster adder. The next section of the report is dedicated to methods which can improve the speed of the adder and hence its efficiency.

4.6.1 The Tandem Adder

The convergence of the PHA depends on the bit patterns of the addend and the augend. Fig. 4.5(a) shows severe carry propagation where the carry is propagated through out the span of the bits. Fig. 4.5(b) shows the addition of the same numbers, but in their one's complement form. The result is the same, but the advantage of using the technique of Fig. 4.5(b) is that the carry propagation is reduced. This technique can be applied to the PHA to reduce long carry propagations. If the carry propagations are reduced, the average number of iterations required for the sums to converge would be reduced and hence the adder would become faster.

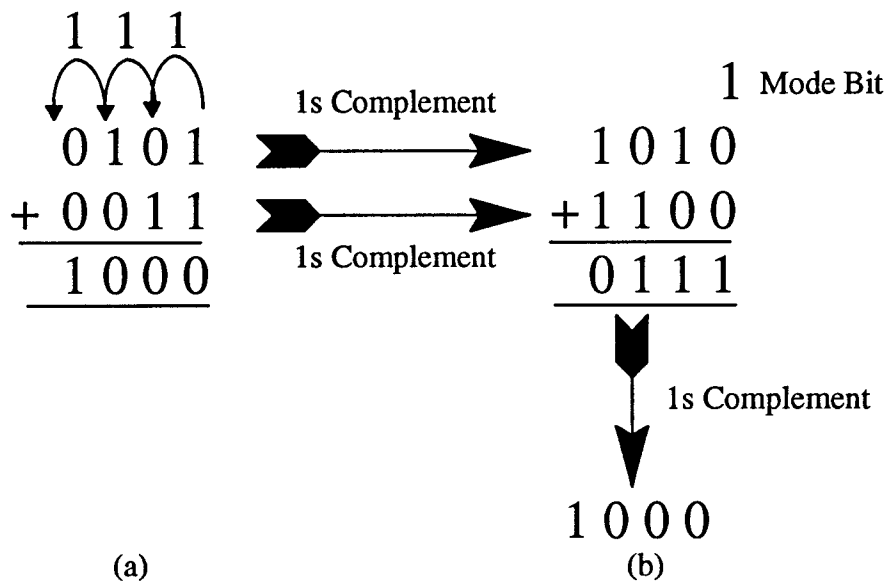


Figure 4.4. Alternative addition of two numbers

By using two adders, one working on normal numbers and the other working on complements, it is possible to attain fast convergence[20]. It should be noted that the worst case carry propagation for one adder is the best case for the other. Hence one adder working on complements can aid the normal adder and the result available at the SUM register of the

adder that converges faster can be used. Thus, it is possible to reduce the number of iterations necessary to ensure convergence.

As an example the addition of the numbers $65535 + 1$ is a worst case for the normal PHA. If the same thing were to be implemented using a tandem technique, the convergence takes place in two cycles as shown in Fig. 4.5.

As can be seen, the tandem adder will always perform better than a single adder. However, this improved performance comes at a very large hardware cost. The hardware required to implement this type of an adder would be at least twice that of a simple PHA. Besides it would be necessary to implement complementing hardware and hardware to detect which of the adders converged first and route the result accordingly. This would typically require n multiplexers, n being the number of bits, and a more complex control logic to control the select lines of these multiplexers. Hence this option is not favourable considering the hardware standpoint, though it is left to be seen if the improved speed justifies this increased hardware investment.

Step	SUM	CARRY	RESULT
0 :	FFFF	+ 1	= 10000
1 :	10000	+ FFFFFFFF	= 10000

Figure 4.5. Tandem addition

4.6.2 Hiding the Zero-Detect delay

From Eqn. 4.4, it can be seen that two terms affect the speed of the adder. The first being the number of iterations, $\log_2 n$, and the second, the zero-detect delay denoted by the term $\log_6 n$. The method indicated in the previous section can be used to reduce the total num-

ber of iterations that the adder takes to converge. This section describes methods to improve the zero-detect delay.

One way to reduce the zero-detect delay is to implement a high-speed zero detector. But the speed up that can be gained this way is limited, since a finite delay contribution will always be present to detect a zero, and this finite contribution gets multiplied by the number of iterations necessary to ensure convergence. Our implementation of the zero-detect circuit uses a high *fan-in* grounded pullup NOR gate. This implementation trades-off power consumption for speed. The advantage however is the near fan-in independant behaviour of such a gate. This is true since the pulldown is effected by using n N-transistors in parallel and the pull-up effected by a single grounded gate P-transistor, where n is the number of bits. Since the primary interest in our zero detector is the transition from a logic '0' to a logic '1', the P-transistor size has been optimized to take this fact into account.

A more efficient implementation is one that eliminates this delay altogether. Since it is physically impossible for the system to work without a zero-detector, and all zero-detectors have a finite delay, the only possible solution to this problem is to hide this delay as much as possible. To do this, the algorithm for PHA addition presented in Chapter 3 needs to be slightly modified.

The algorithm uses a "*predict complete*" strategy. It is assumed that the adder has converged to the right result after each iteration has completed and further processing is done only after it has been ascertained otherwise. Considering our implementation and the architecture of the PHA presented in Fig. 3.7, this delay is seen when the control unit issues a select signal to the multiplexers only after the zero-detect circuit has resolved itself. The probability of this occurrence is only once per addition process. Since the addition takes $\log_2 n$ iterations to complete on the average case, $\log_2 n - 1$ times an unnecessary wait on the zero-detector resolution is noticed. Hence if a different strategy is involved, namely a "*pre-*

dict incomplete” strategy, the prediction would be correct $\log_2 n - 1$ times and wrong only once, on the average.

To implement such a *“predict incomplete”* strategy, the extra hardware required would be a new register, say SUM1, equal in size to the SUM register. Irrespective of the result of the zero-detector output, the control can issue a signal to the MUX to latch internal data after the AND/XOR plane has calculated the new values. After the zero-detector has resolved, and should the assumption prove incorrect, the contents of the SUM register should be rolled back one iteration. The SUM1 register helps this. The contents of the SUM1 register should always lag the contents of the SUM register by one clock edge so that such a roll-back is possible. The constraints that are imposed on the clocking now are that the delay between clock edges should be at least greater than the sum of the XOR/AND plane delay and the MUX delay. The other constraint is that the zero detector delay should be lesser than the sum of the XOR/AND plane delay, MUX delay and the latching delay of the DETFF.

The changes needed in the original architecture of the PHA to include such a capability are shown in Fig. 4.6.

The roll-back needs to be effected every time the adder completes computation. Instead of re-latching the data from SUM1 to SUM to effect this roll-back an easier method would be to use the SUM1 register for all external purposes and the SUM register for all the internal calculations. Thus after the sum has converged, the result is actually available in the SUM1 register instead of the SUM register in this modified version of the adder.

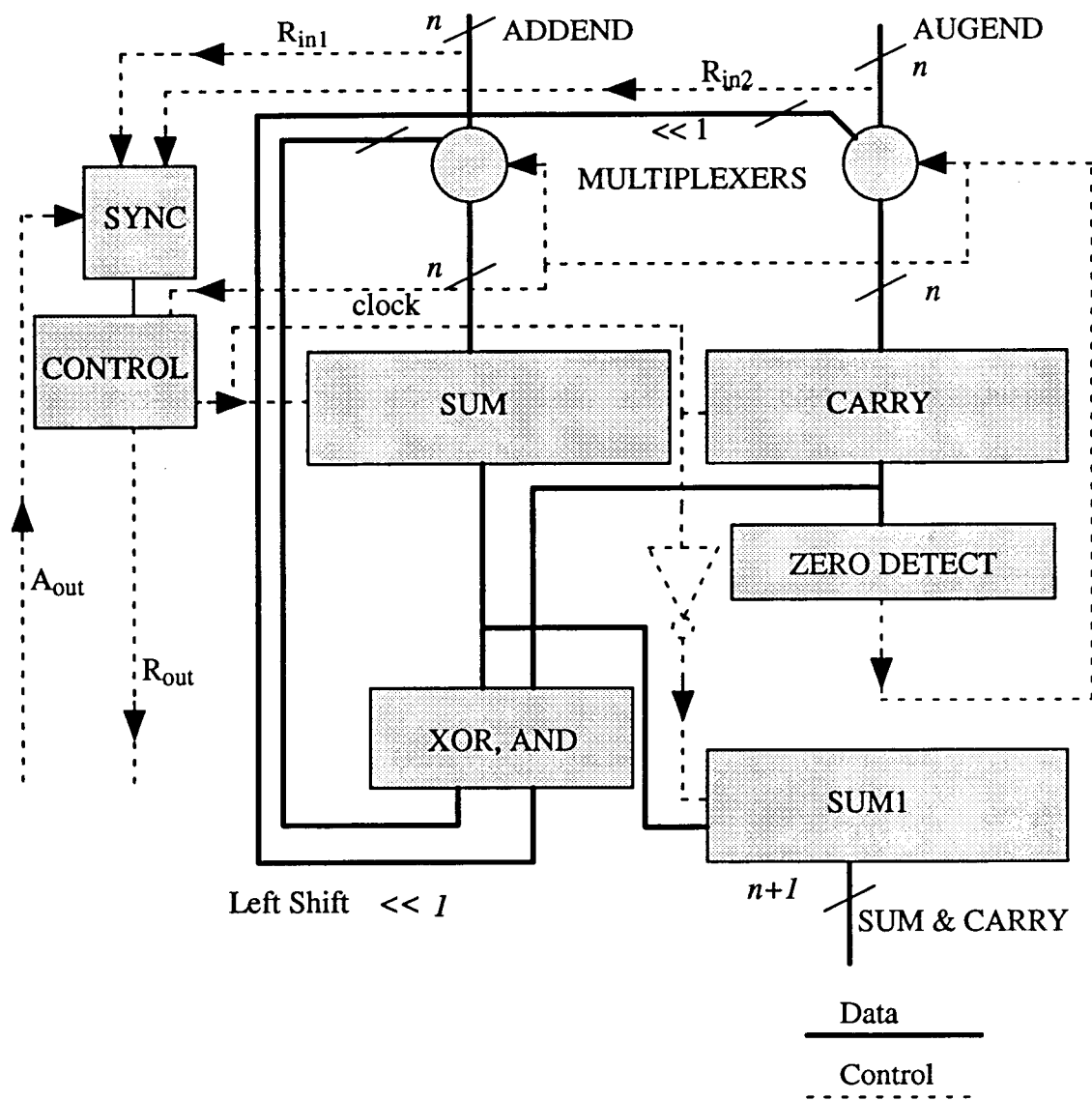


Figure 4.6. Modified Adder to support "predict incomplete"

Using this architecture, it is possible to hide the zero-detect delay almost fully. A very small part of it will show up since the constraints that need to be met here for the clock generation are stricter than in the normal case. Waiting will still have to be done when the *fan-in* of the adder is extremely large, as the zero-detect delays in such cases, would exceed

the sum of the delay of the XOR/AND plane and the MUXing delay. Using the values presented in Chapter 4, the sum of the XOR/AND and the MUXing delay adds up to $3\Delta_g$, where Δ_g denotes the unit gate delay. The delay of the zero detector is given by $\log_f n \Delta_g$ where f is the fan-in of the gates used to implement the zero detector (6 in our implementation) and n is the number of bits in the adder. Hence the zero-detect delay will be fully hidden for cases when $\log_f n < 3$ and it will start showing, by a value equal to $\Delta_g(\log_f n - 3)$, when $\log_f n > 3$.

In our first cut implementation, the adder is capable of supporting addition and subtraction using a MODE control bit. The improvement strategies discussed above have not been implemented. We propose to improve the design and incorporate these improvements and implement an adder possessing a higher efficiency.

This chapter and the previous explained the operation of the adder, evaluated it and suggested methods to speed it up. The facts that actually remain to be discussed are the strategies to implement the adder in silicon and its use in a real application. The next chapter discusses ways to partition the adder such that it can be implemented in silicon with minimum effort.

CHAPTER 5. IMPLEMENTATION OF THE PHA

Once the design and the system level simulation of a system are completed, a number of options exist to implement it in silicon. The solutions range from a full custom implementation to an implementation synthesized directly from a hardware description language like VHDL or Verilog[®]. With the advent of ASIC CAD tools, the latter implementation is rapidly gaining popularity with VLSI designers. This chapter discusses some of the implementation options available and describes the methodology adopted by us to layout the PHA.

5.1 Implementation Strategies

Given the available tools, some options that presented themselves were full custom layout or employing a place and route tool, VPNR (Vanilla Place and Route). VPNR is a tool capable of routing a design captured using VIEWlogic[®] Powerview, provided all the necessary components are available in a standard cell library. It employs the Magic layout editor to produce the final design.

A full custom layout allows a number of optimizations and yields the most compact layout. The disadvantages are that it is time consuming and that any change in the design after layout is started may be hard to implement and might involve a waste in terms of work done. Using a place and route tool like VPNR, on the other hand simplifies the process of layout considerably and produces a layout which is reasonably dense. A change in the design at any point in the design cycle can be accommodated. The disadvantage however, is that the user is completely at the whim of the tool and has little control over the way the layout is completed. Only the most recent advancement in CAD tools facilitate the control of critical path behavior by adopting a timing driven layout methodology. The layout in such cases is optimized based on the timing criterion rather than placement issues as has been done

traditionally. VPNR, being a primitive tool, provided no such capability. Though the number of rows and columns in the final layout can be selected for an optimal aspect ratio, the results obtained are far less impressive in comparison to a full custom layout. This is justifiable since the problem of placement and routing is NP complete.

One possible way to improve the final result obtained using a place and route tool is to partition the design optimally. The design would have to be divided into a number of blocks and each block routed separately and the blocks placed optimally and routed. Since VPNR does not support a block routing capability, we have implemented a semi-custom layout in which parts of the layout were generated using VPNR and the rest were laid out by hand.

5.2 Partitioning the adder

It was emphasized earlier that one of the desired characteristics of a design to be implemented in silicon is modularity. Hence it is necessary to split the adder into modules which can be easily duplicated so that an adder of any *fan-in* can be built. The best way to ensure this is to bit-slice the adder. Fig. 5.1 shows a possible way to split the adder into cells. Each cell of Fig. 5.1 represents one bit of the adder and the cell needs to be duplicated $n+1$ times for an n -bit adder.

The control unit generates all the signals, namely, Select, Clock and Mode. The Carry-in of the current stage is the Carry-out of the previous stage. This takes care of the left shifting that needs to be implemented to complete each iteration of the PHA. The Carry-out bits of all such stages need to be connected to a zero detector to determine convergence. The inset of Fig. 5.1 shows the cell from a high level with all the signals that need to be cascaded. The external input A is cascaded downwards so that a cell placed diagonally below the cell can also get the value of External A. Though this feature is not essential for the imple-

mentation of the adder, it is extremely important for implementing arithmetic operations like division, where the divisor bits need to be cascaded downwards to all the stages.

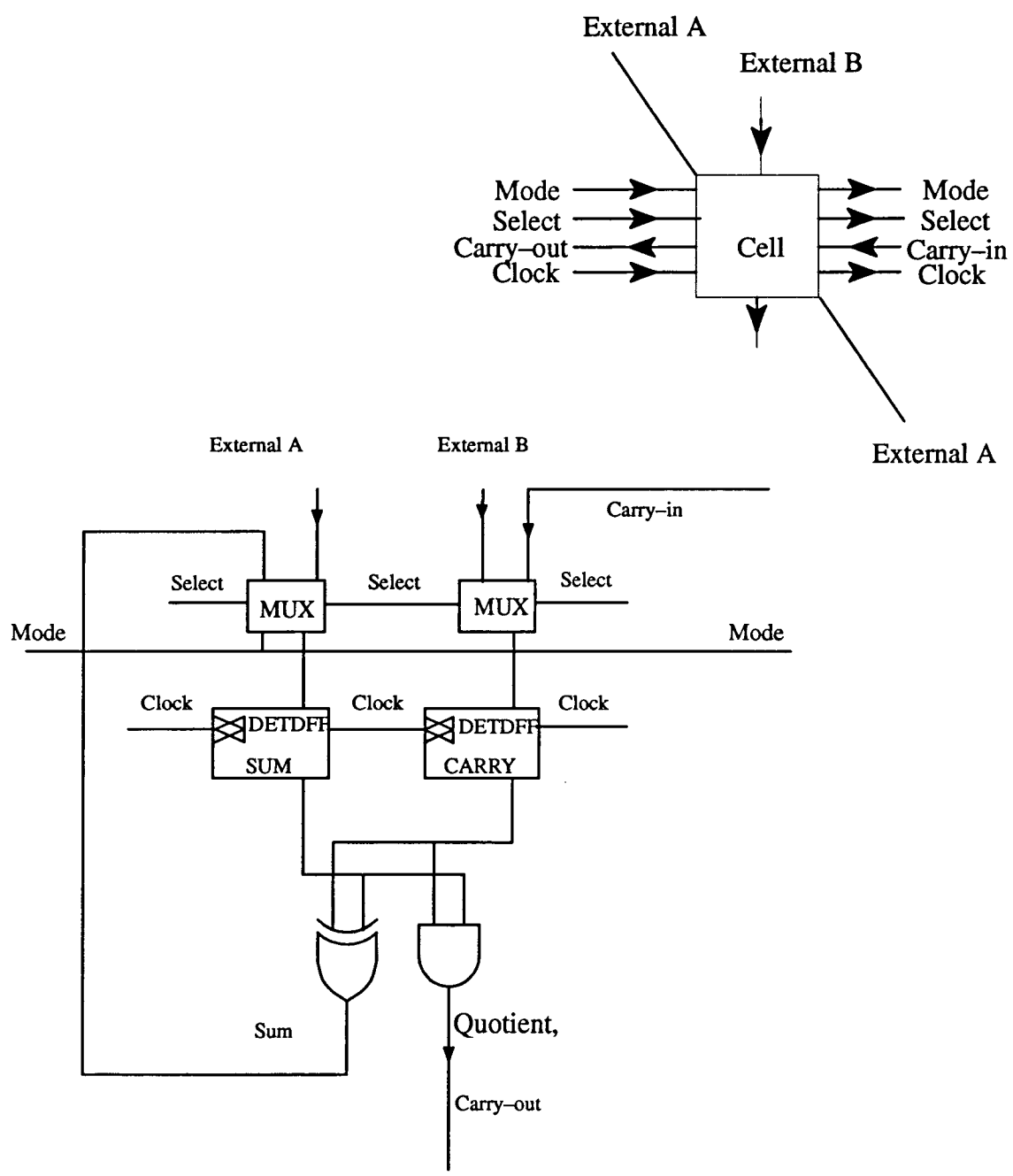


Figure 5.1. Partitioning the PHA into cells

5.3 The Final Implementation

The cell as described in the previous section was implemented full-custom, in the form of a bitslice. To implement a n -bit adder, $n+1$ such bitslices would have to be butted against each other. The $n+1$ Th. cell is necessary since there is no other way in the algorithm to determine a carry-out. The control unit is fan-in independent. Since the control unit is the most likely to change down a design cycle, this was implemented using VPNR after capturing the design in VIEWlogic[®] Powerview. The bitslices were implemented using the cells from a standard cell library. The zero detecting NOR gate and the CONTROL unit laid out using VPNR were all connected together to produce the final layout. The adder that we have implemented follows Fig. 3.7 and does not take advantage of the speed up techniques described in the previous chapter. The simulation results and detailed circuit diagrams describing the various blocks of the adder are presented in the following sections.

5.4 Circuit diagrams

Figure 5.2 depicts the architecture of the PHA as captured using VIEWdraw[®]. The architecture closely follows the one described in Fig. 3.8. The register, Controlled_Rx, is capable of complementing the input data depending on the MODE bit. The multiplexers are present inside the controlled registers. The SUM[16:0] bus produces the actual computation results and SUM[16] is inverted depending on the MODE register to produce the carry-out. Mode '0' implements normal addition and Mode '1' implements subtraction. The Shifter block is just a heap of wires shifting the bits once to the left. The least significant bit of the shifter is determined by the carry select block, which provides control signals to a MUX to select either a '0' or a '1' in the second cycle to implement a two's complement subtraction. The expanded views of the Controlled register, The computation plane and the the CONTROL unit are shown in Figures 5.3, 5.4 and 5.5 respectively.

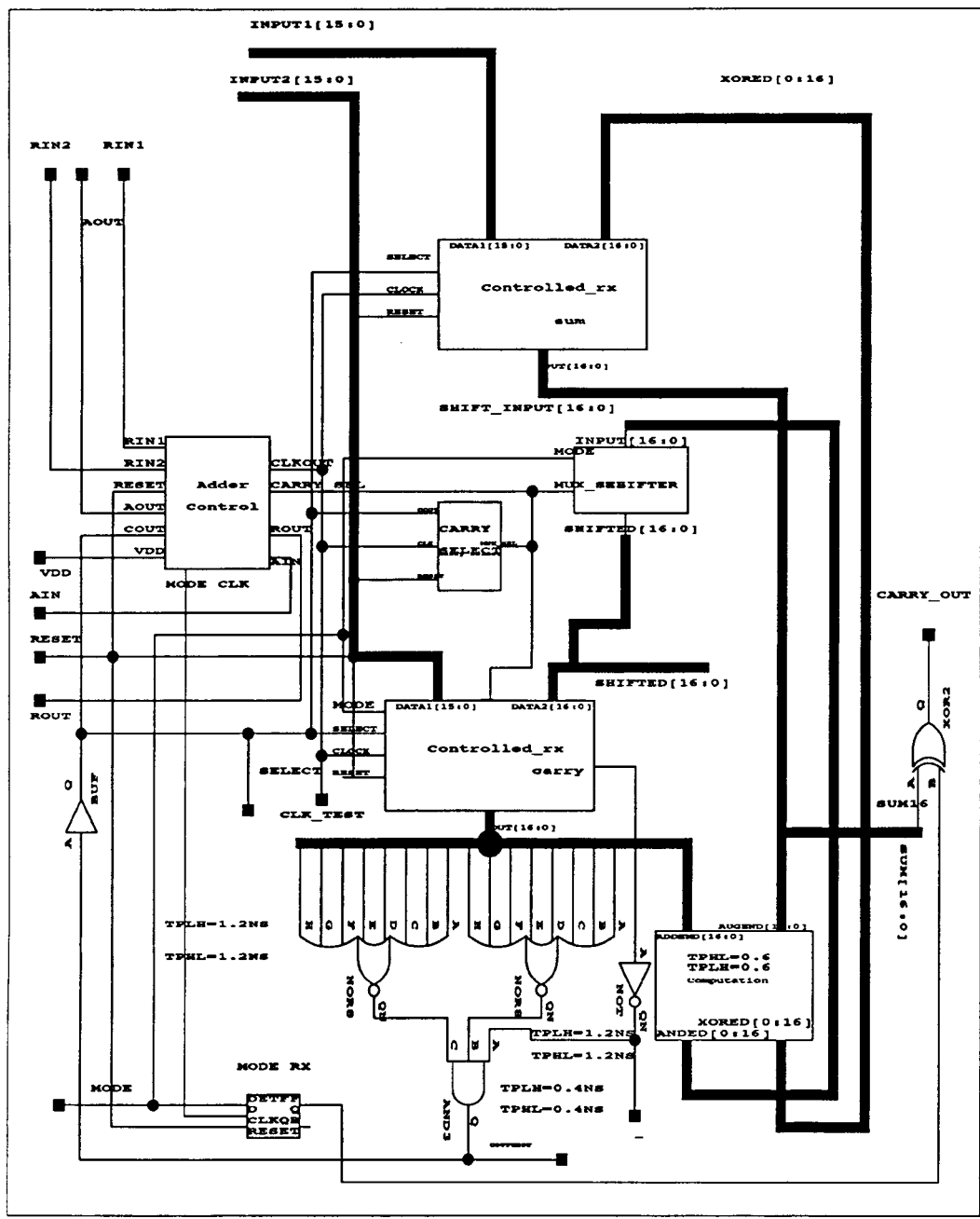


Figure 5.2. The PHA captured using VIEWlogic[®]

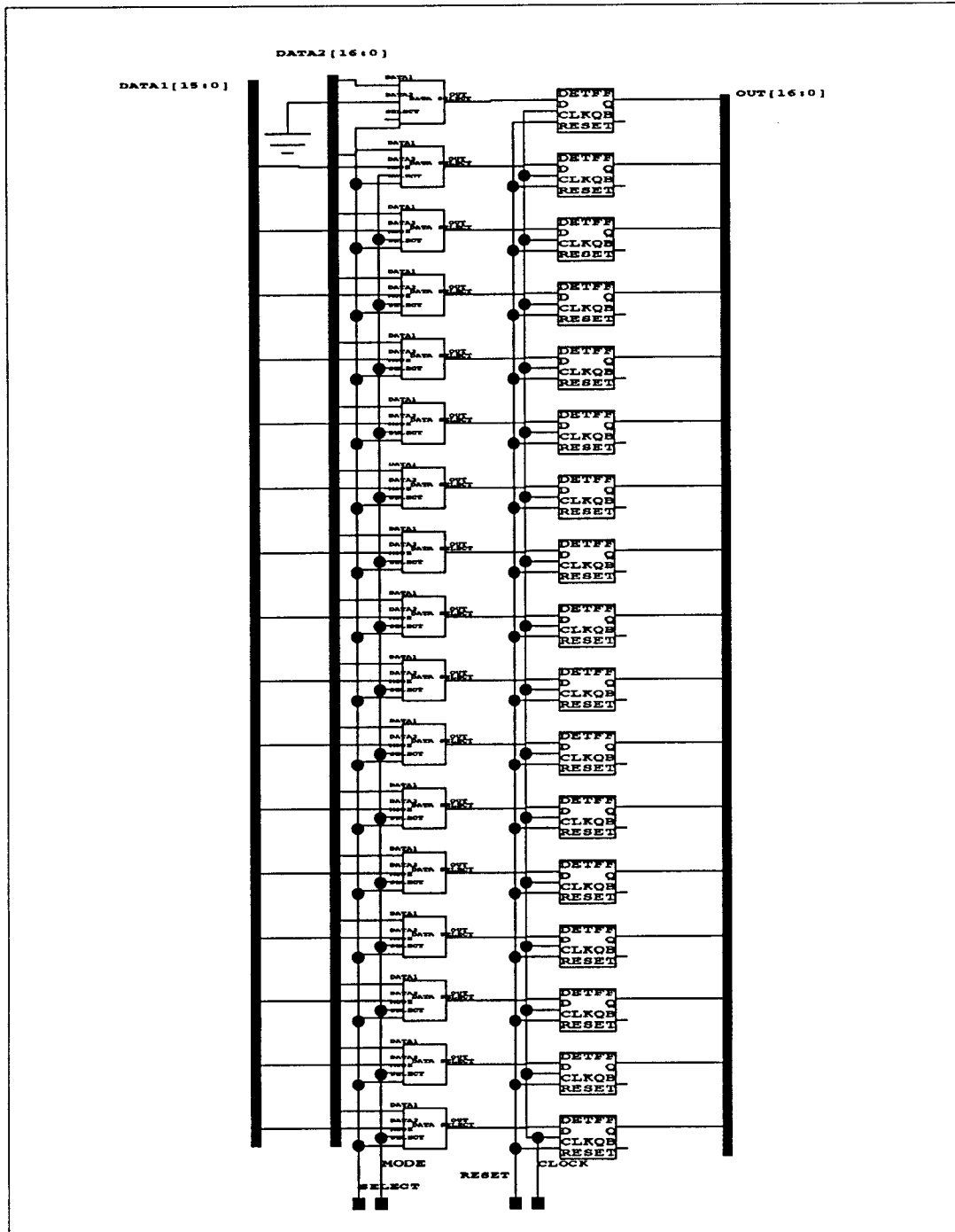


Figure 5.3. The insides of the Controlled Register

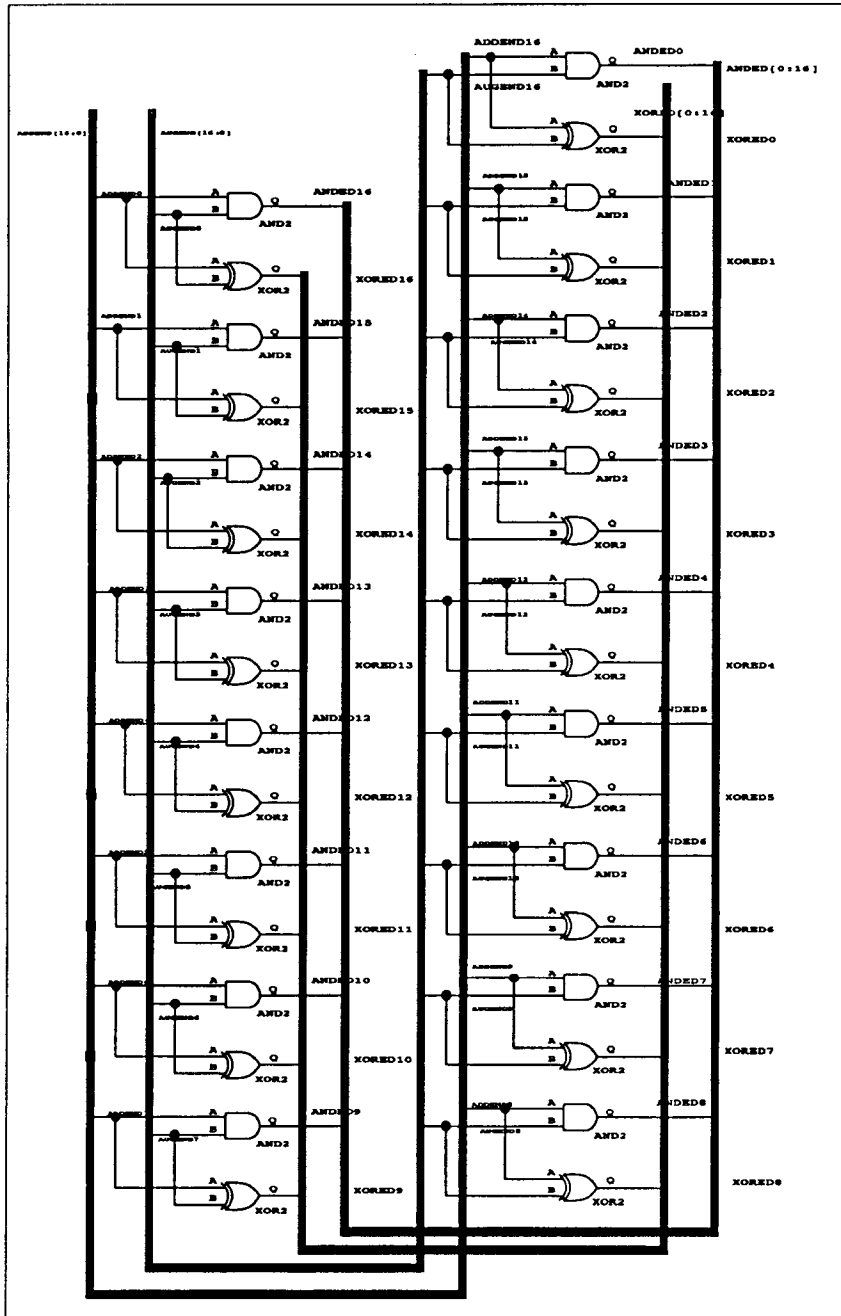


Figure 5.4. The computation plane

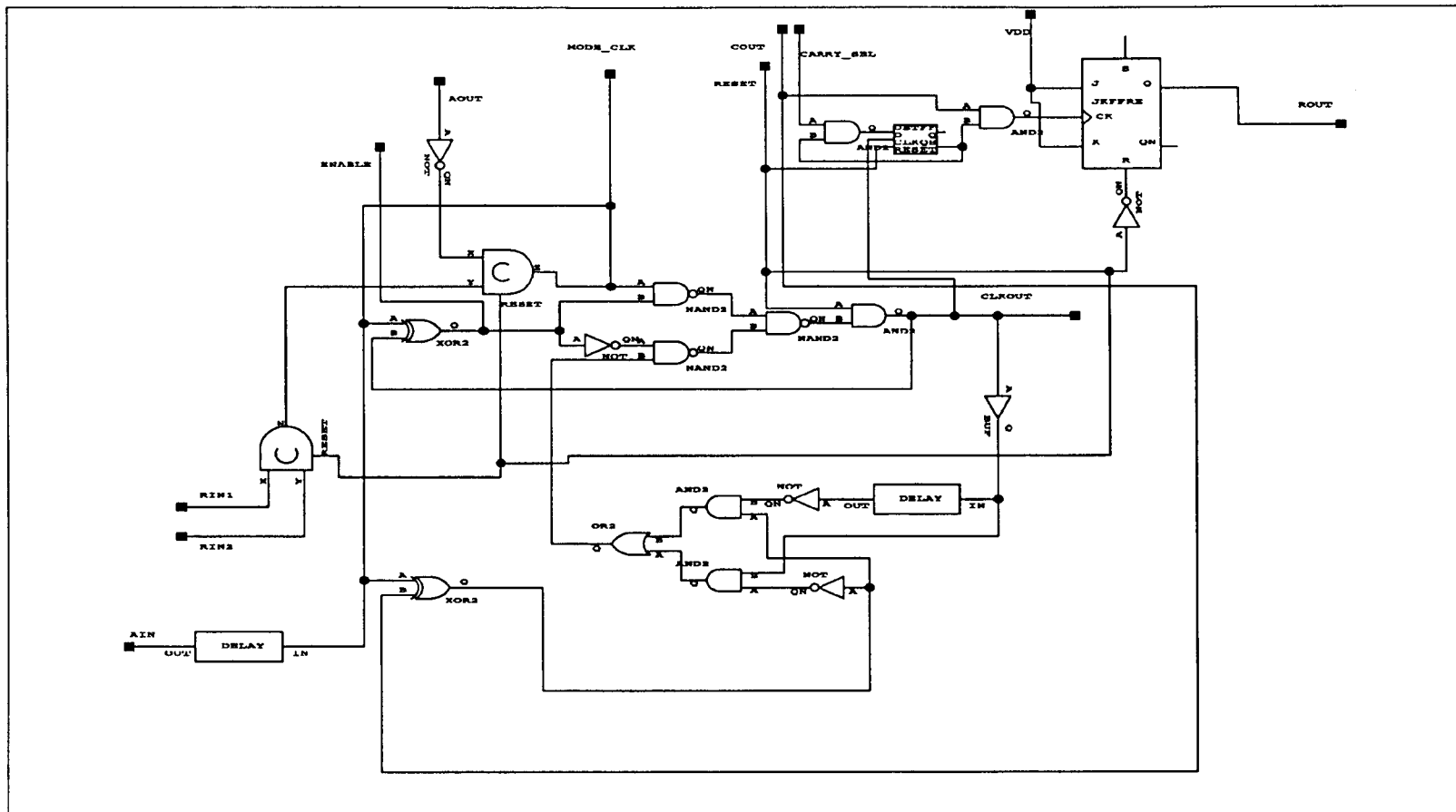


Figure 5.5 CONTROL unit for the PHA

Figure 5.5 shows the presence of two Muller C-elements in the CONTROL circuit. These are used along with the R_{in} and the A_{in} signals to determine synchronization. Since the generation of the A_{in} signal in the figure depends on the A_{out} signal from the succeeding block, this is a non pipelined implementation of the interconnect.

As indicated in Section 3.4 subtraction needs the introduction of a '1' in the second cycle. Since the specific signal names have been introduced earlier, the state diagram and the circuit required to implement the state diagram are shown in Fig. 5.6.

5.5 Simulation Results

The design presented above was simulated using VIEWlogic Viewsim. The same examples that were simulated using the C program in Sec. 3.2 were chosen. Figures 5.7 and 5.8 present the simulation results for addition.

Figure 5.9 presents an example of subtraction of two numbers. The numbers used here are the same as those used in Sec. 3.4.

Figures 5.10 and 5.11 present the SPICE simulations of the layout of the CONTROL unit and one bit of the data path respectively.

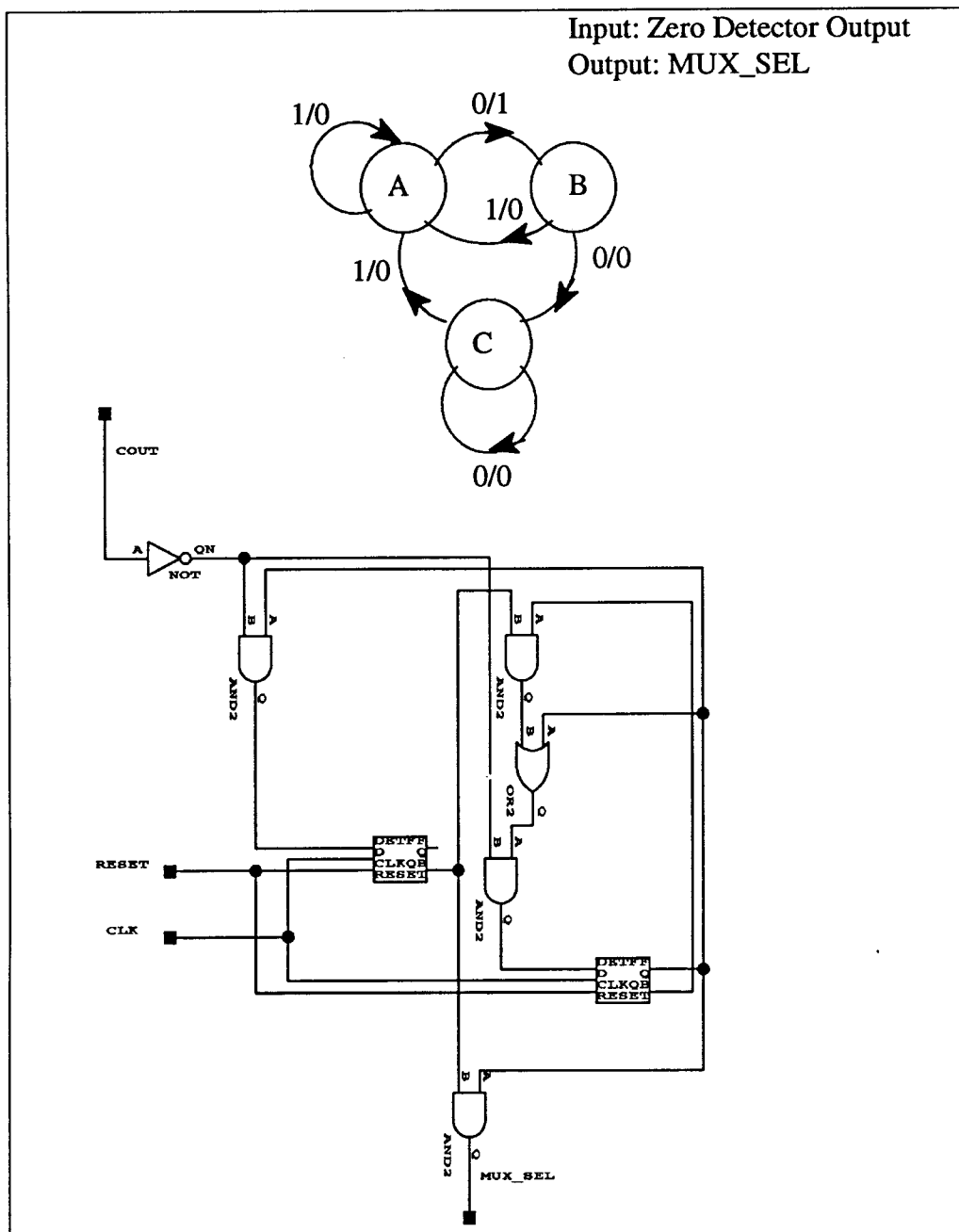


Figure 5.6. Including Subtraction: Carry Select

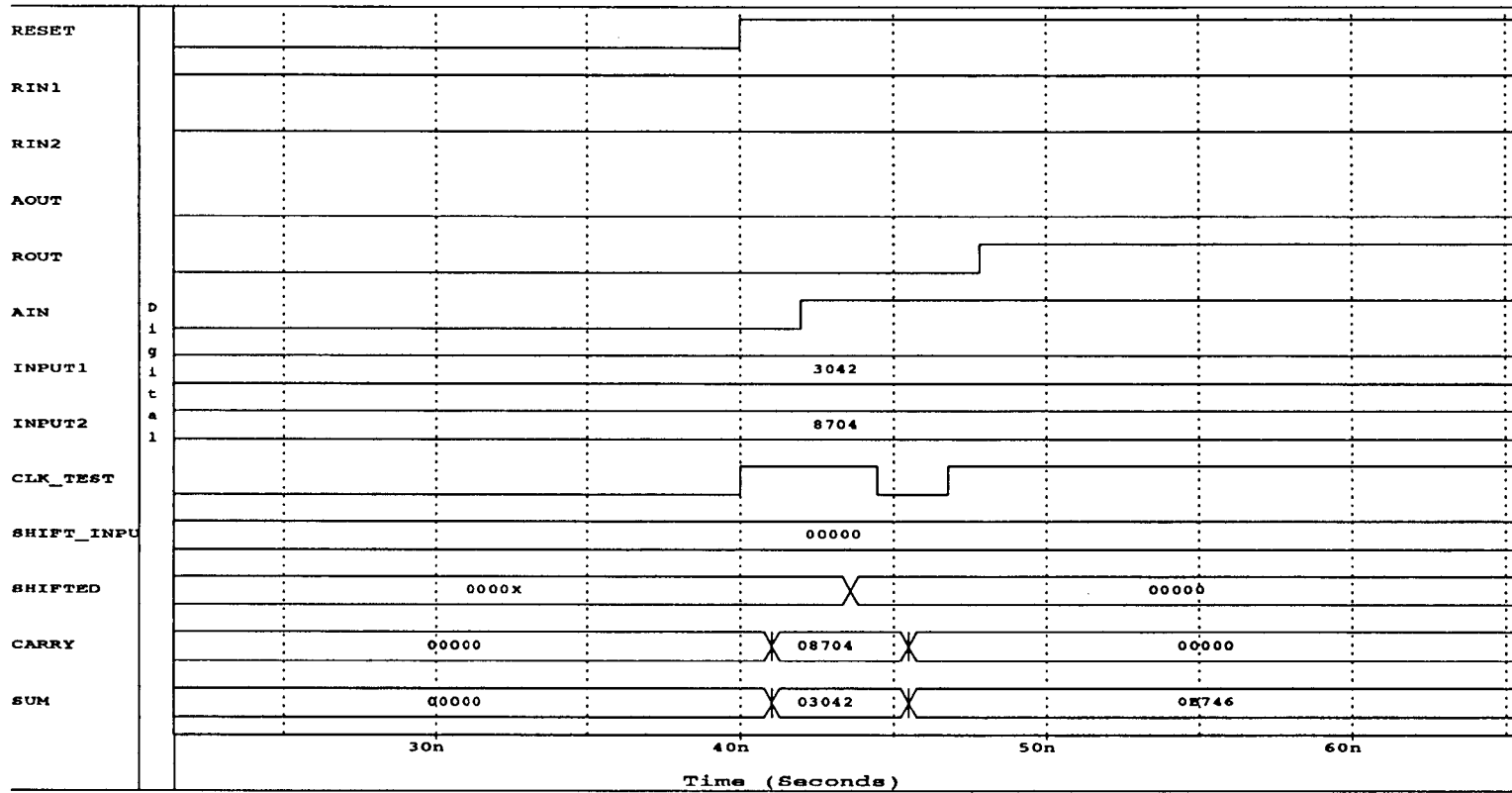


Figure 5.7. Simulation of the PHA using VIEWsim

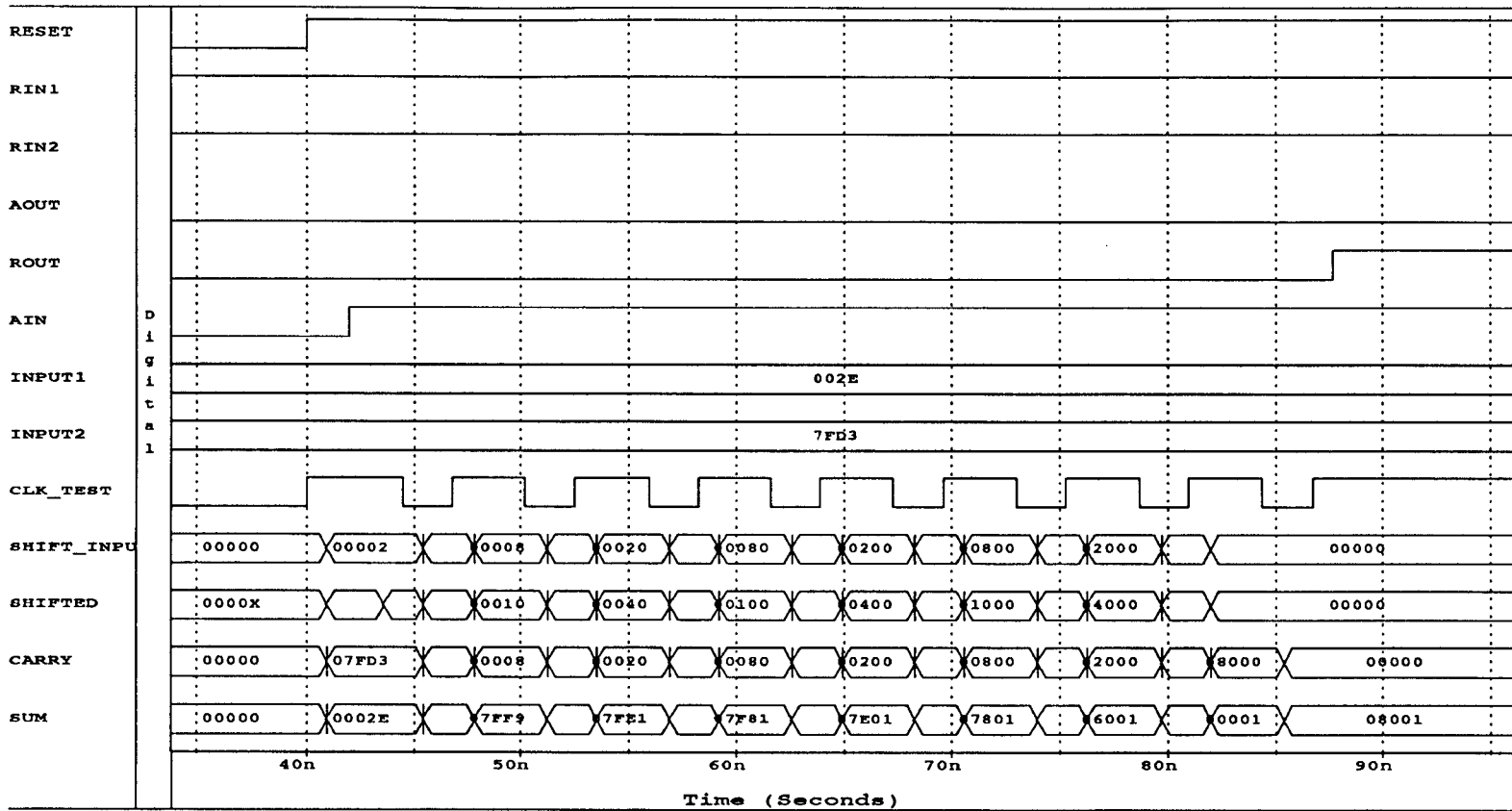


Figure 5.8. Simulation of the PHA(Worst Case)

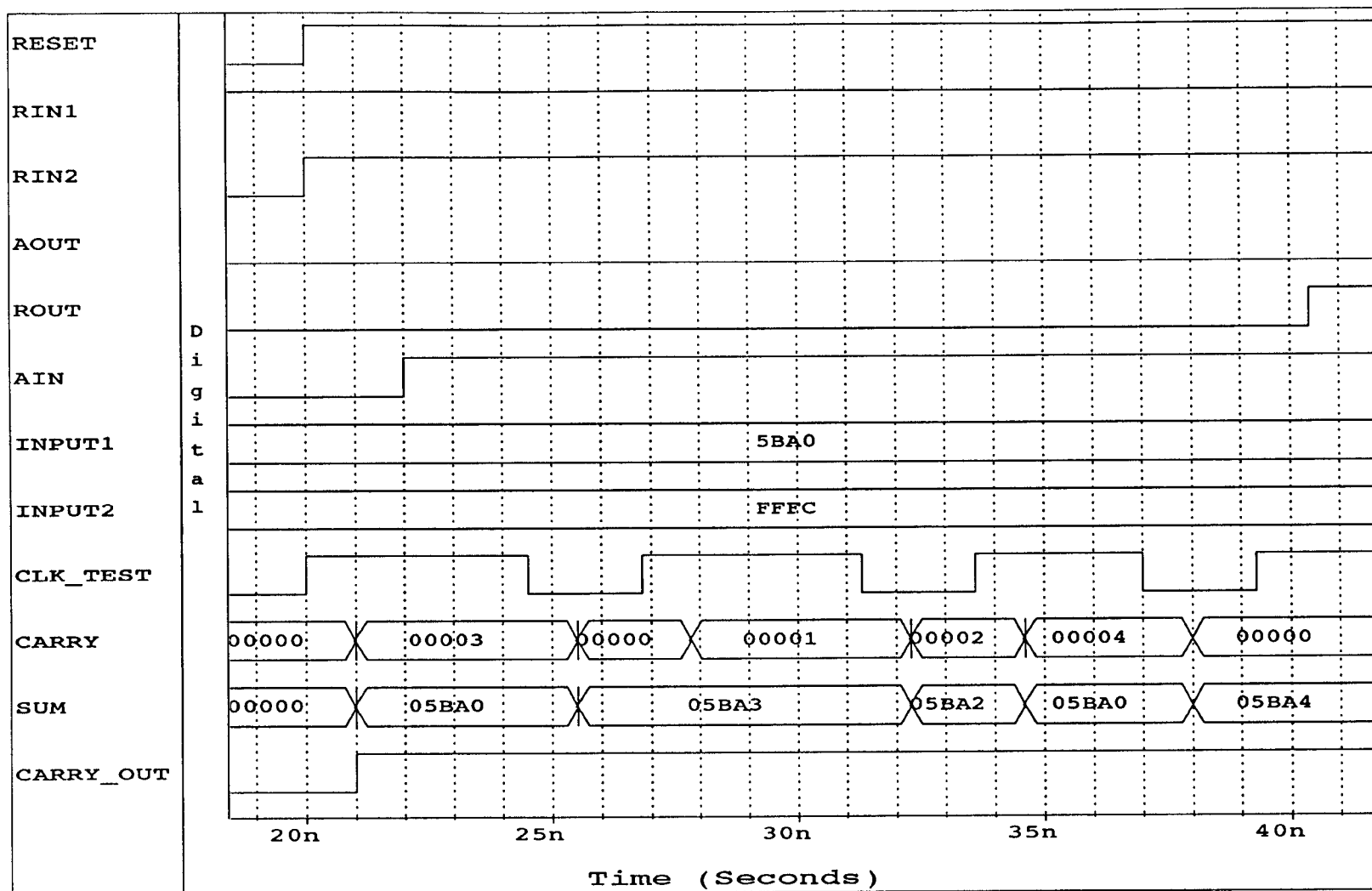


Figure 5.9. Subtraction using the PHA

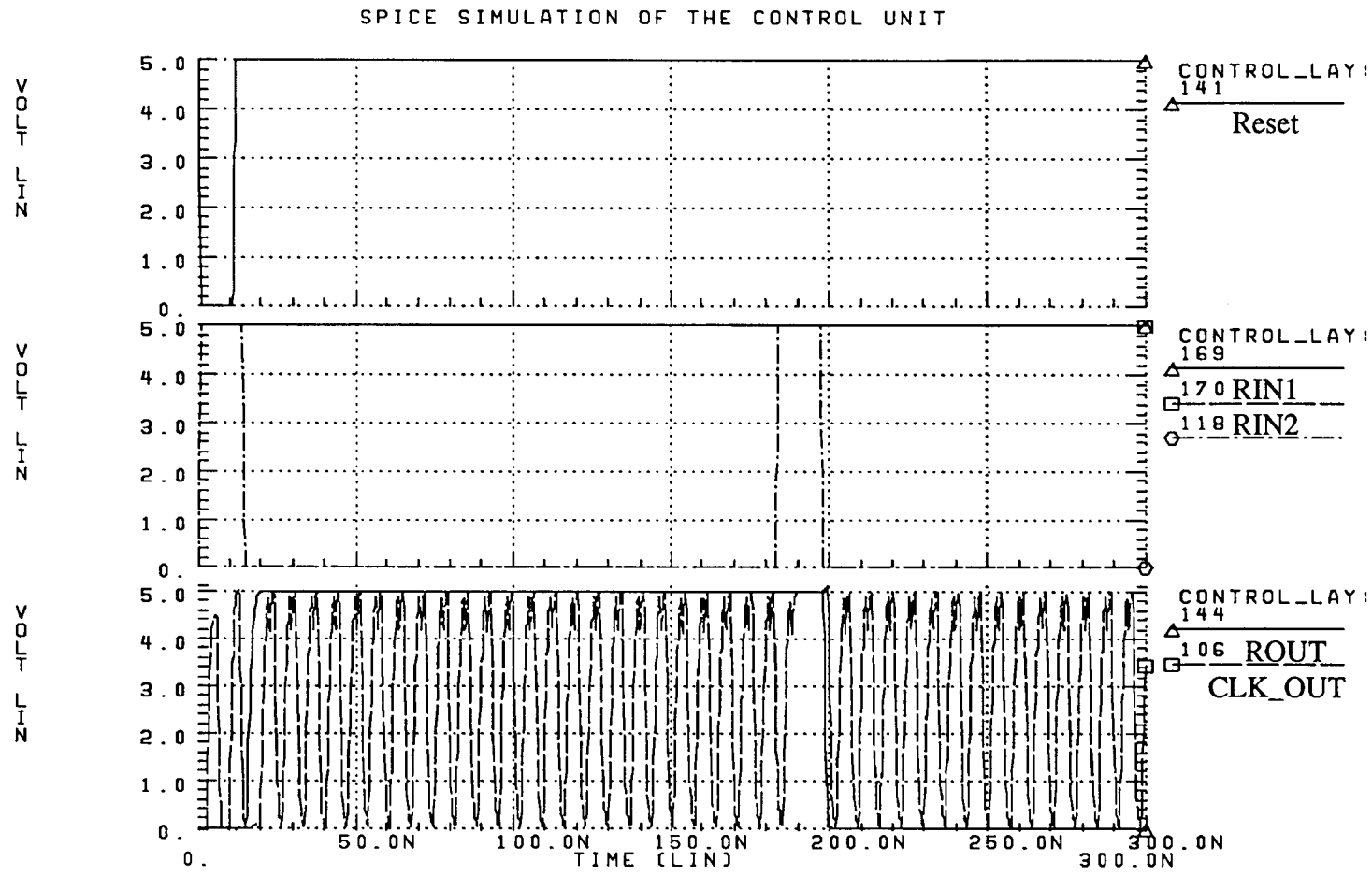


Figure 5.10. SPICE simulation of the CONTROL circuit

** SPICE FILE CREATED FOR CIRCUIT MULLERCRES
94/06/24 13:03:51

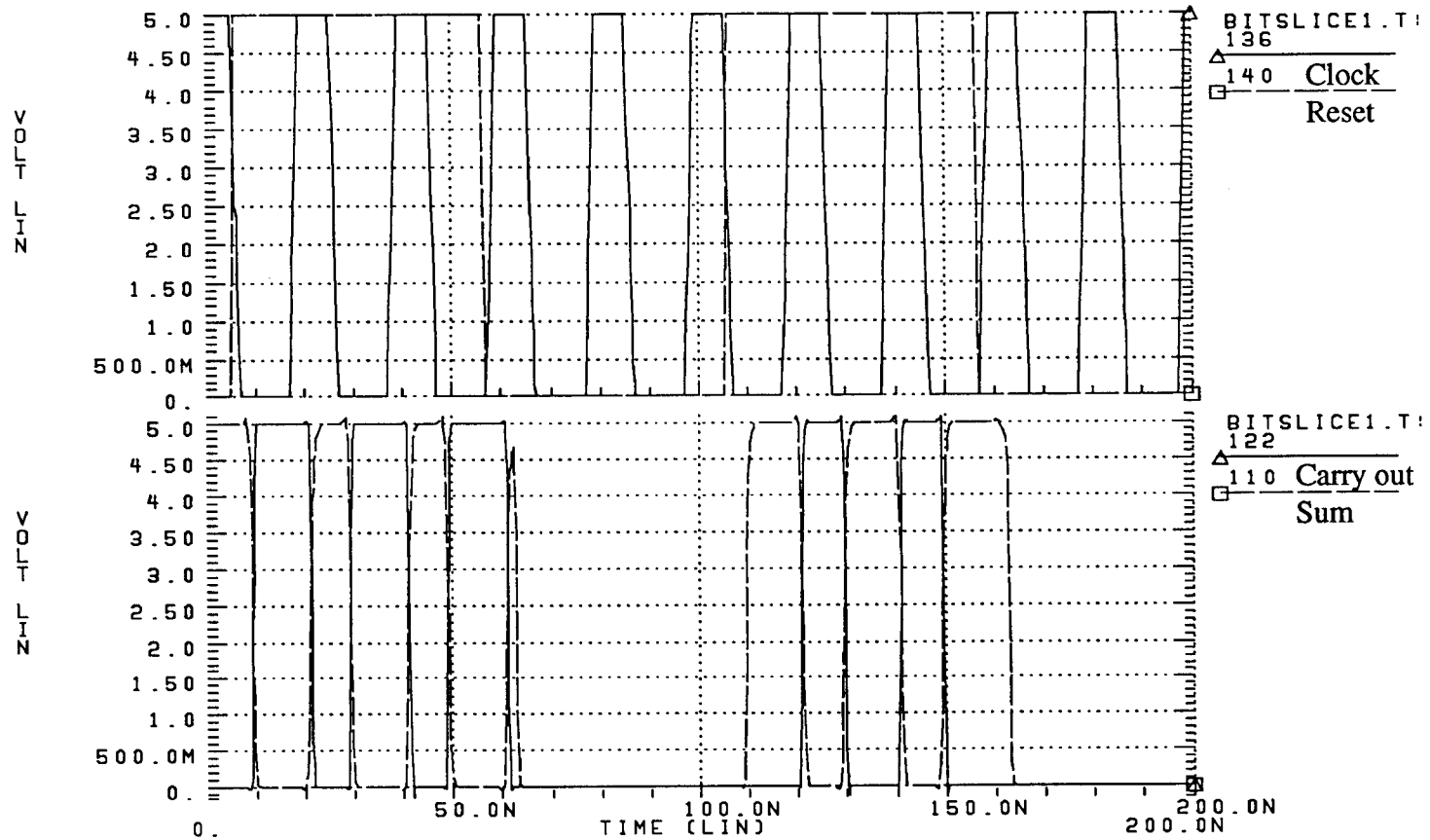


Figure 5.11. SPICE simulation of the bitslice

CHAPTER 6. A SELF-TIMED 16 BIT DIVIDER USING THE PHA

Of the four basic arithmetic operations, division is the most complex and is consequently the most time consuming. Given a dividend X and a divisor Y , the outputs that are of importance in a division operation are the quotient Q and the remainder R , satisfying the condition:

$$X = Q \cdot Y + R \text{ where } R < Y \quad (6.1)$$

The asynchronous PHA described in chapter 3, evaluated in chapter 4, implemented, improved and simulated in Chapter 5 can be used in the construction of a self-timed array divider. On the average, the divider would produce a quotient bit after the average computation time of the n -bit adder. This chapter presents the design of an array divider using cells from the PHA.

6.1 Array Division

All algorithms for division can be implemented using an array of cells where each step of the algorithm is executed by a separate row of cells[15]. Thus, n rows of cells with n cells per row will be needed to implement a radix-2 division algorithm. Division can be performed using either a restoring or non-restoring algorithm. In restoring division, a difference between the previous partial remainder and the divisor is formed and the quotient bit is determined based on the sign of the result. Should the sign prove negative, the partial remainder is restored. This operation is unnecessary. Just by repeatedly shifting and choosing an addition or subtraction operation depending on the sign of the previous result the same result can be accomplished. The advantage of using a non-restoring array is the simplicity with which it handles negative partial remainders. Figure 6.1 depicts a 16-bit array divider.

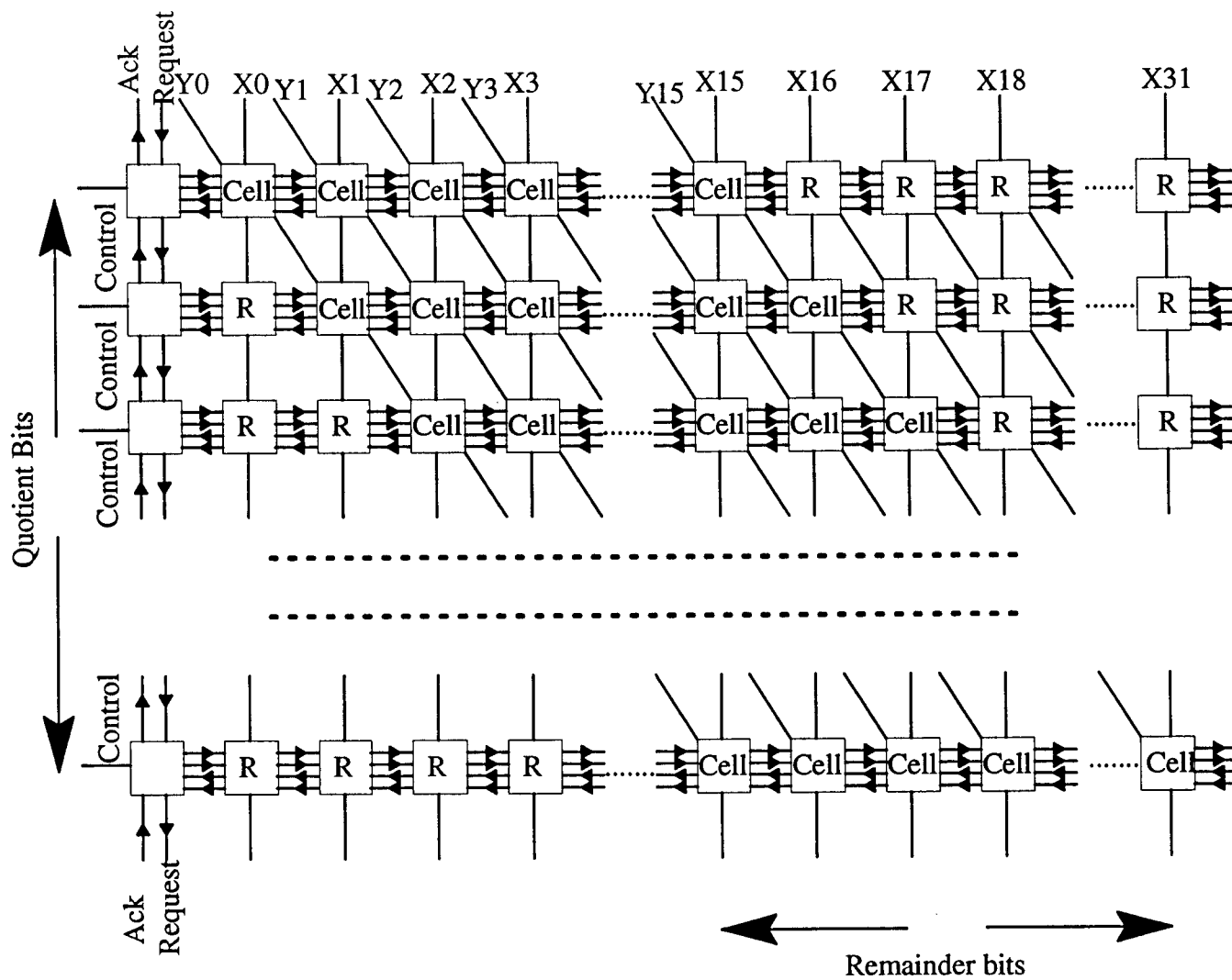


Figure 6.1 A 16 bit array Divider

Each Cell of the divider is one cell of the PHA. The Blocks labelled R, are simple double edge triggered flip-flops. These flip-flops facilitate pipelining. These are necessary as one of the operands is 32-bits long, and only the first 16 bits are used in the first row. The other bits are consumed one by one as the execution proceeds downwards along rows. When a A_{in} signal is issued to the preceding block, the data on this 32-bit input can be changed any-time by the preceding block as the receipt of data has been acknowledged. This could potentially result in a loss in data. On the other hand, not issuing the A_{in} signal to the previous block until the process is complete, will result in a non-pipelined implementation thus resulting in a drastic reduction in the usage of the rows and hence the throughput of the system.

The CONTROL blocks form the left periphery of Fig. 6.1 and generate the internal clocking and synchronizing signals.

6.2 The design of a 16-Bit Divider

As described in the previous section, a cellular construction of the adder can be used to implement an array divider. Since our implementation of the adder in silicon is in the form of bit-slices and the the control unit is also available, it is relatively straightforward to implement an array divider. To capture the design of the divider, and perform a system level simulation, we have used the adder captured in Chapter 5 directly without any alteration. Hence, each row of the array divider in our design looks like a single block. Any attempt to implement the divider in silicon with the cells and bitslices that we have developed would follow Fig 6.1 more closely.

Figure 6.2 depicts the design capture of the divider. The pipelined nature of the divider is clearly seen from Fig. 6.2, with all the PHAs connected to one another and data flowing through each stage of the pipeline. Also, the absence of the global clock and locally generated control signals connecting one block to another can be noticed. Each PHA is a stage of the pipeline, and there exist 16 such stages.

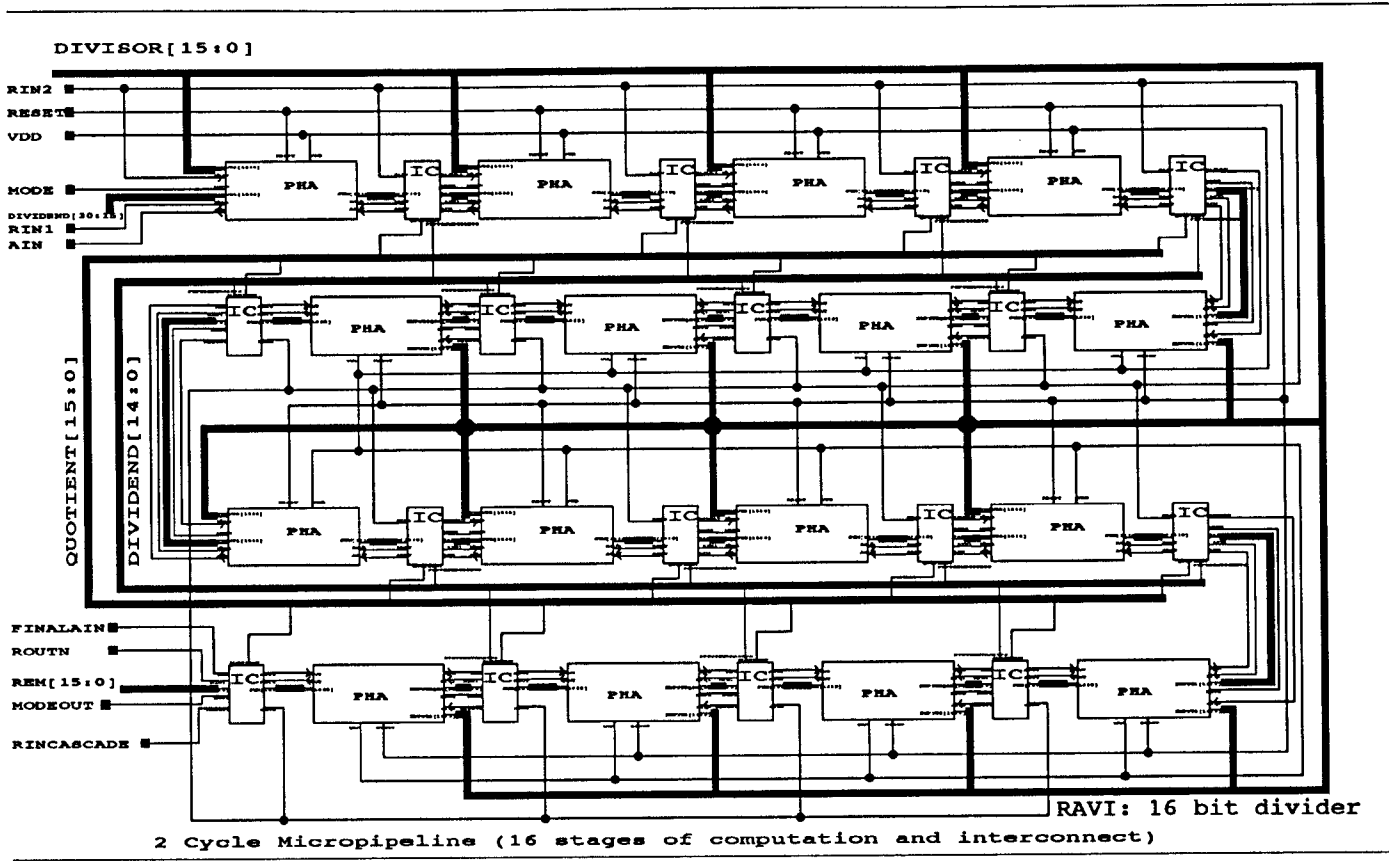


Figure. 6.2. Design capture of the 16-bit divider

A peculiar problem has been noticed in simulating the divider captured above. From the circuit diagram for the CONTROL unit of Fig. 5.5, it can be seen that a RESET should introduce a '0' at the clock output line. The rest of the actions taking place in the CONTROL unit depend on this initial '0' token circulation. It can also be seen that this '0' was introduced by us by using a redundant AND gate at the clock output and connecting it to the RESET line. The event driven nature of the simulator constrains that the RESET signal should be the one that changes last. If this condition is not met, the system goes into an unknown state. We ensured this in the simulation of the adder by writing simulation impulses keeping this in mind. However in simulating the full divider, this condition is impossible to ensure using the simulation impulse file. Since the R_{in1} of all the stages except the first one are undetermined until the previous stage produces a R_{out} , the global RESET signal is the first to change for all the PHAs except the first one. Thus even before all the values of R_{in} are available the RESET changes and the whole system goes into an unknown state. However, we have verified the functionality of the design by splitting the reset line into many parts like RESET1, RESET2 and so on and changing the value on each RESET line only after all the other changes in the circuit were stable. Since it is impractical to split the RESET line into 16 different lines, we verified the functionality of the first few stages only, and found the system to be functionally correct. The SPICE simulation of the CONTROL unit of Fig. 5.10 shows that, in practice any signal can change first and the system is still functional, and the problem noticed here with the RESET line change is only due to the simulator and not due to the functionality of the circuit itself. Thus the divider works even though we are unable to provide any formal simulation results here.

Division is the most time consuming operation of all the four basic arithmetic operations. This is due to the fact that the computation of any row in the array for division is dependent upon the results produced in the previous row. Hence an array divider is used for the sole purpose of its regularity and not due to any special speed up possible. Since the array division process consumes large quantities of hardware, the usual scheme employed for

carry propagation is the *Ripple Carry Scheme*. Very rarely are the schemes like CLA used to speed up the carry propagation due to their prohibitive hardware consumption.

The PHA offers an effective solution to this problem by speeding up the addition process, but causing an extremely small hardware increase. Thus the PHA is ideally suited for operations like array division.

CHAPTER 7. CONCLUSIONS AND FUTURE WORK

The preceding chapters of this thesis presented the design and development of a Parallel Half Adder and its application in a non-restoring array divider. This chapter summarizes the work done and the conclusions. Directions for future research in this area have also been provided.

6.3 Conclusions

We have designed and implemented a Parallel Half Adder in silicon. The adder has been designed for fabrication using a 1.2μ process.

Since iteration in time is used, the hardware complexity of the PHA approaches the hardware complexity of a simple CRA. The PHA compares reasonably well with a CLA in terms of speed and proves to be faster than a CCA and CRA. In terms of overall efficiency of the adder, the CSA proves to be the most efficient. The CLA is posited next and is closely followed by the PHA.

The number of iterations that are necessary for the PHA to attain convergence depend on the input data patterns. Modelling such a system synchronously would be extremely difficult as the number of clock cycles necessary for convergence varies each time. Thus the PHA represents a case where an asynchronous algorithm is mapped to an asynchronous system. Since the results obtained in terms of speed are comparable to existing adders and the hardware complexity lower, the PHA is a viable alternative to using existing adders for arithmetic in self-timed circuits. It presents a reasonable balance between hardware complexity and speed, making it a good choice for designs where some speed is necessary but the area on-chip is not sufficient to incorporate a very high speed adder design.

The adder is particularly attractive for array division, since the result of each row of the array divider is used for processing by the row immediately below. Thus the speed with

which the addition or subtraction in each row is performed becomes a limiting factor in terms of speed. Since the hardware complexity of the divider is already high, methods to speed up carry propagation like CLA are not viable. Such a dependency does not exist in other arithmetic operations like multiplication where all the partial products can be generated in parallel.

The overhead incurred due to handshaking can be more than offset, if the algorithm is partitioned appropriately, and the thread chosen to process between synchronization points is sufficiently large. This is demonstrated by the pipelined implementation of the PHA, where the cost of synchronization is a small fraction of the computation time. Thus if a system is appropriately partitioned, it is possible to utilize the host of advantages brought out by self-timing and not feel the cost paid for hand-shaking.

6.4 Future Work

Our implementation of the PHA uses the simplest architecture and does not take advantage of the possible speed up techniques proposed. Future work should be directed in this area to improve the adder.

The adder has been characterized here only partially. Attempts to find the probability of occurrence of the worst case and a more detailed study of the carry propagation patterns should be undertaken and accurate mathematical models need to be developed.

As a result of implementing this project, we have updated a standard cell library to incorporate some of the basic primitives like Muller-C elements and Double Edge Triggered flip-flops. A partial effort to provide entries pointing to such a library, in a design capture tool has also been made. The library available as of now can just be used for design capture and auto routing, and cannot be used for simulation. VHDL models need to be attached to the components in this library so that they can be used for design capture and simulation as

well. This is an important step, as a place and route tool is needed to complete full synthesis of a design, and ongoing research in our group is directed at synthesis of self-timed circuits.

Another direction for active research should be the design of an interconnect to hook up an asynchronous building block with a synchronous system. Such a hybrid architecture can potentially utilize the advantages of both synchronous and asynchronous systems.

Since self-timing proves to be well suited for DSP, the use of this adder in DSP algorithms should be studied. We believe that the adder could be best suited for such applications since DSP IC chips are usually core limited, and area is usually a limiting factor prohibiting costly investments like CLA adders. Since our adder provides a balance between speed and area, its use in DSP chips could prove beneficial.

BIBLIOGRAPHY

- [1] I.E. Sutherland, "Micropipelines," *Communications of the ACM*, Vol.32 No.6, pp.720–738, June 1989.
- [2] D. Pountain, "Computing Without Clocks," *BYTE*, pp.145–150, January 1993.
- [3] W. Wolf, *Modern VLSI Design – A System Approach*, Prentice Hall, pp108–112, 1994
- [4] N. Weste and K. Eshragian, "*Principles of CMOS VLSI Design, A Systems Perspective*," Addison Wesley Publishing Company, 1993
- [5] E. J. McCluskey, "*Logic Design Principles*," Prentice Hall, pp.82–84, 1986
- [6] D.B. Armstrong, A. D. Friedman and P. R. Manon, "Design of Asynchronous Circuits Assuming Unbounded Gate Delays," *IEEE Transactions on Computers*, C-18, Dec. 1969
- [7] F. U. Rosenberger, C. E. Molnar, T. J. Chaney and T. P. Fang, "Q-Modules: Internally Clocked Delay-Insensitive Modules," Technical Report 4706, 4707, 4708, Sutherland, Sproull and Associates, 1986.
- [8] T. H. Meng, "*Synchronization Design for Digital Systems*," Kluwer Academic Publishers, 1991, ch. 2, pp. 35.
- [9] S.L. Lu, "Self-timed Arithmetic Structures in CMOS Differential Logic," Ph.D Thesis, Computer Science Department, UCLA, 1991.
- [10] L. G. Heller et al., "Cascode Voltage Switch Logic: A Differential CMOS Logic Family," *ISSCC Digest of Tech. Papers*, 1984, pp.16–17.
- [11] L. Merani, "Micro Dataflow – A Dataflow Approach to Self-timing" Master's Thesis, Electrical and Computer Engineering Department, OSU, 1993.
- [12] S. L. Lu, "Implementation of Micropipelines in ECDL," submitted to *IEEE Trans. on VLSI*, 1993.
- [13] G. M. Jacobs and R. W. Brodersen, "Self-Timed Integrated Circuits for Digital Signal Processing Applications," *VLSI Signal Processing III*, IEEE Press, 1988.
- [14] S. L. Lu, and M. Ercegovac, "A Novel CMOS Implementation of Double-Edge-Triggered Flip-Flops," *IEEE Journal of Solid-State Circuits*, Vol. 25, No. 4, August 1990, pp. 1008–1010.
- [15] I. Koren, "*Computer Arithmetic Algorithms*," Prentice Hall, 1993, Ch. 5, pp.71–92.

- [16] K. Hwang, "Computer Arithmetic," John Wiley and Sons, 1979, Ch. 3, pp.69–93.
- [17] J. Sklansky, "Conditional Sum Addition Logic," IRE Trans. EC-9, No.2, June 1960, pp.226–231.
- [18] J. Sklansky, "An Evaluation of Several Two-Summand Binary Adders," EC-9, No.2, June 1960, pp.213–226.
- [19] A. Burks, H. H. Goldstine and J. Von Neumann, "Preliminary Discussion of the Logic Design of an Electronic Computing Instrument," Instit. Adv. Study, Princeton, NJ, 1946.
- [20] D. B. Swink, "Parallel Halfadder Additon," private communication, April 1993
- [21] J. D. Ullman, "Computational Aspects of VLSI," Principles of Computer Science Series, Ch.2, 1984