SELF-TIMED INTEGRATED CIRCUITS FOR DIGITAL SIGNAL PROCESSING

by

Gordon Merrill Jacobs

Memorandum No. UCB/ERL M89/128

30 November 1989

SELF-TIMED INTEGRATED CIRCUITS FOR DIGITAL SIGNAL PROCESSING

by

Gordon Merrill Jacobs

Memorandum No. UCB/ERL M89/128

30 November 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

SELF-TIMED INTEGRATED CIRCUITS FOR DIGITAL SIGNAL PROCESSING

by

Gordon Merrill Jacobs

Memorandum No. UCB/ERL M89/128

30 November 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Self-Timed Integrated Circuits for Digital Signal Processing

by

Gordon Merrill Jacobs

Abstract

As the clocking rates of large digital Integrated Circuits (ICs) continue to increase, the global synchronization of the circuits becomes an increasingly difficult design problem. While scaling the feature size of an IC technology has traditionally been the method for obtaining increased performance, limitations in the wiring layers can prevent taking full advantage of the faster devices used in the circuit. At the board level, asynchronous interfaces have been introduced in places where synchronization becomes problematic. A natural extension to this idea is to extend the use of asynchronous circuits to within an IC. Self-timed circuits are introduced as a means for implementing asynchronous ICs. Self-timed circuits, in addition to performing computation, generate completion information that can be used by appropriate interconnection blocks to oversee the transfer of data between stages without the use of any global clock signal.

As a design example, a complete micro-processor based digital signal processor (DSP) was designed and fabricated using self-timed circuits following a 4-cycle handshaking protocol to provide fully asynchronous operation without a global clock. The DSP is mask programmable and it was fabricated in $2\mu m$ N-well CMOS in an active area of $6.6mm$ x $4.7mm$. The processor showed reliable operation at power supply voltages between 3.5V and 7V, illustrating how self-timed circuitry adjusts to variations in environment (and processing) while still operating correctly.

This report starts by introducing previous approaches to asynchronous processors followed by an explanation why it is both necessary and feasible to implement them in

IC form. A survey of self-timed circuitry is then presented along with a description of a synthesis method for generating the interconnection or "handshake" circuits required to implement the correct handshaking protocol for data transfers. Details of the DSP design are given along with experimental results. Finally, some conclusions about the work are drawn.

Robert W. Brodersen

Robert W. Brodersen
Chairman of Committee

Acknowledgments

I'd like to express my deepest gratitude to Professor Bob Brodersen who has done so much for me since we met way back in 1976. Throughout both my Masters and Ph.D. research, Bob has been a source for many good ideas, financial support and counseling. Without his friendship, the work would not have been nearly as enjoyable.

The collective group of students under Prof. Brodersen known as "bobsgroup" provided much help to me throughout my research. I thank them for all of the useful discussions. Outside of our immediate group, Professor David Messerschmitt and Teresa Meng contributed many interesting ideas and developed for the most part, the synthesis program used for the handshaking circuits. Thanks to the CAD group for helping me learn programming.

Thanks also to Professors Paul Gray and David Hodges who have made me feel more like a peer than a lowly grad student since my return to school.

The guys at Teknekron CSD also deserve a big thanks for giving me support over the years. It has been a pleasure working with Amine Haoui, Nan-Sheng Lin, and Roger Strauch in parallel with my research at UCB.

Last, but not least, thanks to Mom and Dad for encouraging me to do what I want to do in life, and for absorbing all of the complaints along the way. To the boys in Chicago, and you know who you are, your faith in my abilities on and off campus *beats* all.

Contents

List of Figures

Chapter 1

Introduction

Since the inception of the Integrated Circuit (IC), the drive to shrink feature sizes further has continued at a rapid pace. By reducing the size of the devices, a chip with given complexity can be made smaller and less expensively, or a chip with a given area can achieve greater functionality. The speed of the transistors also increases with scaling so that more processing power is obtained. Accompanying the advances in digital IC processing technology have been products that take full advantage of the greater speed and complexity. The progress had been so rapid that (perhaps to the frustration of computer owners) the life cycle of a digital chip may only be several years.

Currently, as gate lengths are breaking the $1\mu m$ mark, the ability to take advantage of the smaller, faster transistors is becoming endangered by limitations in the means for interconnecting them. The delays associated with the wiring layers of an IC are becoming significant with respect to the circuit delays. Additionally, the scaling process for these layers has reached some fundamental limits. While the delay of a short wire that connects local devices is still insignificant with respect to circuit speeds, a wire that must traverse an entire IC can have appreciable delay. Most digital processors depend on a global synchronizing signal or "clock" to function properly. All operations of the circuit are initiated by the edges of the clock and their duration must be less than one or multiple periods of the clock. For finer resolution in time, the clock is typically split into several phases. Because all circuitry is synchronized with the clock, the clock signal must be distributed over the entire area of the chip. In suitably regular structures such as gate arrays, buffering trees can equalize the delay encountered by the clock signal to different parts of the chip, however, this creates a problem with off-chip communication.

Cell based designs, which are becoming increasingly common, rarely have a regular clock loading scheme and the control over the exact placement of the clock lines is often limited in automated chip assembly tools. Clock *skews*, i.e., a difference in time between the edges of the same clock signal at different locations on the chip, are a result of the timing distribution network and they require that either the logic circuits meet certain latency requirements, or that the time between phases of a multi-phase clocked system be increased. This negatively impacts the time available for computation. The scaling of IC technology tends to aggravate the clock distribution problem. Smaller devices and larger chip areas translates to a larger number of devices per chip, making the capacitive loading on the clock lines the same or *larger*.

Self-timed circuits are introduced in this report as a means of synchronizing the chip at a more local level. This alleviates problems associated with distributing a clock signal over the entire face of the chip for synchronization. Many of the problems associated with this type of design style have been studied and solved in this research. The results as published thus far however, are quite controversial. While most designers agree that asynchronous transfer of information (using some type of self-timed circuits) is required in system design, the level at which it is required is a subject of intense debate. The approach taken in this thesis was to bring self-timing onto the chip and all the way down to the level of the individual gates of a digital circuit. The philosophy behind this approach is based simply on the history of computer design. As speeds have increased, the level at which asynchronous interfaces have been used has become increasingly local. Extending this idea naturally brings self-timing onto the chips that perform the computation. So, it is felt that while the technology of today may not necessitate this design approach, time will compel its widespread usage. The circuits described in the later chapters show the proof of concept in the design approach described. The technique however, is still in its infancy. It is hoped that the reader will see the feasibility of the approach as well as the good reasons for studying it, and then be able to make some decisions regarding its potential for future applications.

Digital Signal Processors (DSPs) represent a specialized form of high-speed computer design. DSPs do not typically contain the high degree of flexibility seen in general purpose micro-processors but the required level of performance is very high if real-time applications are implemented. Because of this high-speed requirement, the interest in this area held by the author, and the tremendous growth in the use of DSPs in recent years, a

DSP design was chosen as the vehicle for testing the self-timed circuit design methodology studied.

1. Definition of self-timed

In the literature, the terms *synchronous*, *asynchronous*, and *self-timed* can refer to different things in different contexts. In this thesis, the term *self-timed* is used to describe circuits that have an underlying method of operation. Formally,

Definition 1.1 **Self-timed** *circuits are circuits which, in addition to performing computation, supply a completion signal to indicate when the computation is finished.*

All of the cell designs described in later chapters are self-timed because they supply a Data Valid (DV) signal which indicates when the signals at the output of the circuit are valid. Depending on the implementation, self-timed circuits often require an Initialization (I) signal to reset the outputs and DV signal. The I signal can either be a dedicated connection to the logic, or a way of encoding the input data. In any case, the model of a self-timed circuit is that of a logic functional block whose DV signal is asserted at a time after initialization that is exactly the amount required to evaluate the function.

Synchronous and *asynchronous* will be used to refer to the timing of a system:

Definition 1.2 *A* **Synchronous** *system is one where the transfer of information between combinatorial blocks is performed in synchrony with a global clock signal.*

Definition 1.3 *An* **Asynchronous** *system is one where the transfer of information between combinatorial blocks is not performed in synchrony with a global clock signal, but rather at times determined by the latencies of the blocks themselves.*

Therefore, the reliance of a synchronous system on a clock signal eliminates the need for self-timed circuitry within. Conversely, asynchronous systems typically depend on the existence of self-timed circuits for determining the correct times for communication between stages. In the DSP design presented, interconnection or *handshaking* circuits make use of the completion signals of the self-timed circuits. By enforcing a handshaking protocol, the transfer of information is ensured to be error free and at the correct times. In this thesis, the entire chip design utilizes self-timed cells, so the (asynchronous) system is also called self-timed.

2. Background

Asynchronous processing has been studied for a long time as a way of designing general purpose computers. The idea is simple and logical in that each task to be undertaken is started at the moment the preceding task completes rather than starting only at the edges of a global synchronization signal. The completion times of each task are often dependent on both the type of task being performed and the data on which is being operated. In the asynchronous scheme, the processing time approaches an *average* for all tasks rather than being set by a clock period that is the worst-case completion time of all tasks. This can provide a tremendous benefit in terms of processing times. Also, given the local nature of the timing in an asynchronous system, extensibility is enhanced. There is a large volume of literature describing asynchronous processing and related topics. In this section, developments leading up to this work will be compared and contrasted.

2.1 Early Work

The design of asynchronous logic circuits, while studied in the past, has not been very fruitful in terms of the actual circuitry being used. Often referred to as the Huffman model[1], asynchronous circuits with bounded delay elements were first published in 1954. The discussion was limited to combinatorial networks. If more than one signal in the network was changed, an erroneous output or *hazard*[2, 5] could be generated.

Hazards

For an asynchronous combinatorial network to function properly, it must be transient-free. In other words, a signal should not change temporarily when it is required to remain fixed, or change more than once when it is required to change only once. A circuit contains a hazard when, for some transient-free input change, there exists some combination of stray delays for which the output contains a transient. For a combinatorial function f, suppose I_1 and I_2 are two input states that are applied in succession. The circuit has a *static hazard* if $f(I_2) = f(I_1)$ and the input sequence $I_1 I_2$ can generate the output sequence $f(I_1)\bar{f}(I_1)f(I_1)$. The circuit contains a *dynamic hazard* if $f(I_2) \neq f(I_1)$ and the input sequence $I_1 I_2$ can produce the output sequence $f(I_1)f(I_2)f(I_1)f(I_2)$ or a longer sequence. A different type of hazard defined in [2] can occur in circuits with unbounded delays and is denoted *delay hazard.* A circuit has a delay hazard if, for the input sequence $I_1 I_2 I_3$, one of the following output sequences is produced:

1. $f(I_1)f(I_2)\bar{f}(I_2)f(I_3)$, *where* $f(I_2) = f(I_3)$
2. $f(I_1)f(I_2)f(I_3)f(I_2)f(I_3)$, *where* $f(I_2) \neq f(I_3)$

Delay hazards are usually caused by a mismatch in delays of different gates in the network. For a certain input sequence, the outputs of the mismatching gates get in a "race" to the network output causing an unpredictable result (if the delays are not known exactly in advance). This phenomenon occurs in improperly designed synchronous circuits also and it is called a *race condition* in contemporary literature.

By assuming bounded delays in the circuit elements, the hazards can be eliminated by adding more gates in the circuit and restricting the inputs so that no more than a single one changes at a time. For a sequential circuit[5, 6] or finite state machine, race conditions can lead to steady state hazards due to the existence of feedback. While some techniques were worked out to eliminate this type of problem, the design procedure is error-prone and restricting the movement of inputs also restricts the applications of the circuits themselves.

In the Huffman model, line and gate delays are assumed to be bounded and no restrictions are placed on their relative magnitudes. In the Muller model[4], gate delays are assumed to be unbounded but line delays are assumed to be zero. Using the Muller model, a method of using data detectors or "spacers" to encode data lines for the purpose of indicating valid outputs was published[2]. However, the process of encoding the data lines in the design phase is cumbersome and it involves a large overhead in the required hardware. In a pipelined architecture, the use of spacers reduces the hardware efficiency to less than 25% [64]. As with many of the earlier schemes, the overhead in hardware, difficulty of design and the relatively slower speed of logic circuits, where clock distribution was not such a problem, caused asynchronous circuits to be impractical.

In the model for employing self-timed circuits in the processor design presented in this work, the computation circuits are separated from the timing or handshaking circuitry. The sequencing of operations is entirely determined by the handshaking circuits and the computation stages, implemented as self-timed circuits, simply add latencies to the handshaking signals. This differs from the early work in asynchronous design in that the circuits are actually *delay-insensitive.* Properly designed handshaking circuitry displays no hazards and it is shown that an algorithm can be used for automatically synthesizing the handshaking logic, making the system design straightforward.

2.2 Arbiters and Metastability

Another problem that is often associated with asynchronous circuits is the *metastability* phenomena[19]. A bistable circuit can go into an unstable state in which the output is not at a normal logic level, and stays that way for an indeterminable amount of time. It is the setting of the inputs in a very particular way that causes the metastable state. *Arbiters* or circuits which arbitrate multiple requests for a resource may suffer from the metastability problem if requests can occur at any time such as in an asynchronous system. There has been a great effort in characterizing the problems associated with arbiters[14, 15]. In circuits where arbiters are used however, the inevitability of a metastable state is usually accepted and it must be considered in the overall operation.

In the signal processor circuits described in this work, the use of arbiters is avoided completely. It is not the self-timed nature of the circuit operation that implicitly causes metastable problems. Rather, it is the architecture in which the circuits are employed. If the architecture (or possible architectures) can be well defined early in the design, requests for resources come from pre-defined places, eliminating non-deterministic configurations of the circuitry that might require an arbiter. For example, in a datapath containing three pipeline stages, each stage receives requests only from the preceding stage. Further, the configuration of the circuit at the finest level of detail is known by the instruction control signals coming from the program ROM. Thus, even when the configuration changes slightly (say, changing the state of a MUX in the datapath), the information is available beforehand.

2.3 Data Flow Computers

Data Flow architectures offer a possible solution to the problem of exploiting concurrency of computation in a program or algorithm[7, 8, 9, 10]. While a data flow computer can benefit from an asynchronous timing scheme, the overall architecture greatly differs from the Von Neumann model often followed for computing. Instruction execution is fundamentally different than traditional sequential execution. In a data flow computer, an instruction is ready for execution when its operands have arrived. A consequence of this is that many instructions of a data flow program may be available for execution at the same time. Hence, concurrency of computation is natural. The operation of a data flow computer follows that of a *data flow graph* describing the algorithm. A data flow graph may be drawn directly from an algorithm or derived from a sequential programming language using methods similar to those used in optimizing compilers for analyzing the paths of data dependency. The data flow graph is made up of *actors* connected by *arcs*. The arcs define paths over which values from one actor are conveyed by tokens to other actors. An actor is *enabled* when it has tokens present on each input arc, and there must be no token on any output arc of the actor. Any enabled actor may *fire* i.e., perform its computation, which removes one token from each of the inputs and places a token with the result values on each of the outputs. This is sometimes referred to as "data-driven" computation. It is evident that an interconnection of actors as specified in a data flow graph requires a much different hardware architecture for implementation than a traditional computer. In fact, the term data flow refers to these differences. The temporal aspects of the the phrase "data-driven" and the description of an actor "firing" make "self-timed" or "asynchronous" come to mind. The idea of timing the firing of actors at exactly the moment the inputs are ready certainly fits the data flow model. However, "ready" is abstracted in data flow to mean that time within the constraints of the hardware realization. For a clocked system, this may be one or several clock cycles. In an asynchronous system, it may be closer to the actual time. A self-timed circuit as defined above is similar to a hardware realization of a data flow actor. Beyond that, there is nothing else that the Digital Signal Processor described in this work has in common with a data flow computer. The *techniques* presented are more general. So, while they were applied to the design of a DSP here, there is nothing to prevent their use for designing a data flow machine.

To add to the confusion, the term *Synchronous Data Flow*[11] is often used to describe a a data flow graph where the nodes consume and produce a pre-determined amount of data at each input or output arc. In signal processing terminology, systems with fixed or integer related sample rates can be specified as synchronous data flow graphs. The "synchronous" part of synchronous data flow only refers to the relative rates of tokens flowing between actors. It has no implication about the hardware implementation, which could again be done with clocked or non-clocked circuitry.

2.4 Systolic Arrays

Systolic Arrays[13] consist of an array of modular processing elements with regular and (spatially) local interconnections. The data in the array are rhythmically computed - as timed by a global clock - and passed through the network. The network configuration and the numerous processing elements exploit the concurrency in an algorithm the same way as a data flow model. The systolic nature of the timing i.e., the regular clocking and the requirement for a global synchronization signal, prevents the operation from being data-driven. Thus, with its synchronous timing and multi-processor architecture, it in no way resembles resembles the DSP in this work.

2.5 Wavefront Arrays

The *Wavefront Array*[13] applies self-timing to a systolic array eliminating the global clocking scheme and allowing its operation to be data-driven. If pipeline registers are placed between elements, in the terminology of this work, it realizes an n-dimensional self-timed pipeline. Depending on the implementation of the self-timing and communication between elements, a wavefront array might resemble the DSP described here. In other words, the techniques for interconnecting processing elements in the DSP are directly applicable to the wavefront array. The system architecture is just different.

In the coming chapters, the reader will become familiar with the differences between the approach taken in this work and previous efforts. In the model used, the handshaking circuits are both critical to the operation, and an addition over what is seen in synchronous design. The overhead on an integrated circuit for the required handshaking circuitry however, is considered negligible.

3. The Self-timed Model

The model that is followed in this report for a self-timed system, that is, an asynchronous system made from self-timed circuits, is shown in Figure 1.1. The self-timed circuit, which accepts an initialization signal (I) and generates a completion signal when data is valid at its outputs (DV) is used in conjunction with a storage register and interconnection or handshake circuit to form a complete stage. The storage register holds input data while the computation proceeds. The handshake circuit is responsible for enforcing a protocol on the communication signals between stages and the protocol ensures that transfers only occur at the correct times. The DSP design shown in later chapters can be abstracted as an interconnected group of the stages shown in the figure. For bit-slice datapaths, a completion signal is often generated for each bit in the data word. These individual completion signals must be used to form a single **DV** signal for the entire stage in order to maintain data alignment.

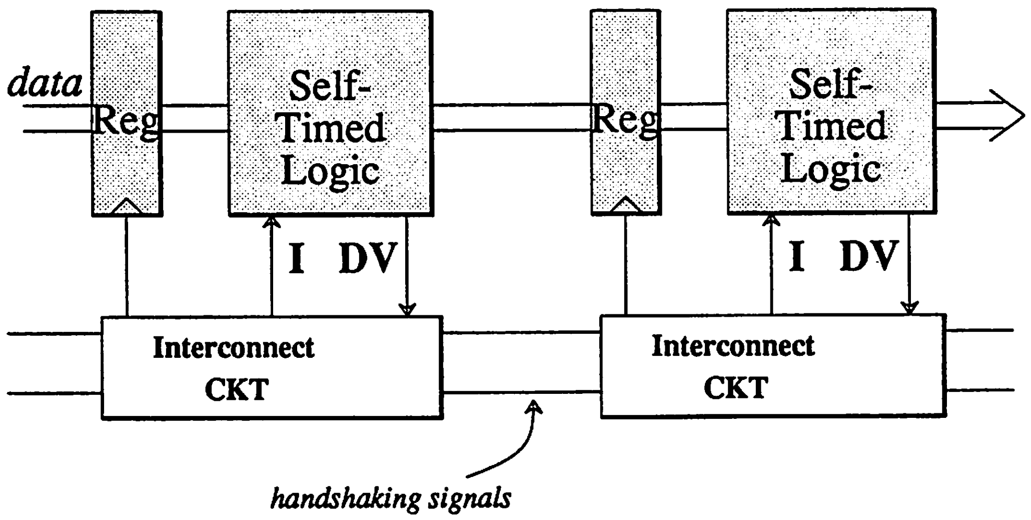


Figure 1.1: The model for a self-timed system.

4. Scope

Chapter 2 examines in detail some of the reasons for which the research of self-timed integrated circuits was undertaken. These reasons include both technological and design issues for future IC designs. In Chapter 3, the actual realization of integrated self-timed circuits is discussed. One logic family, which provides both true and complement outputs, allows by a simple extension of the gate design, the generation of a reliable completion signal. There are several alternative styles for this logic family which, while not used on the chip design presented here, are feasible for some applications. These alternatives are surveyed. The synthesis of handshaking circuits which make use of the completion signals to effect correct data transfers between stages is discussed in Chapter 4. Some of the most common handshake circuits are presented. Since the c-element is a component which is required in most handshaking circuits, and its design can affect the efficiency of operation of an asynchronous system, Chapter 5 was devoted to a survey of c-element designs and performance. Chapter 6 shows the designs of some DSP macrocells that are self-timed. Next, the design of a fully self-timed and programmable digital signal processing chip is presented in Chapter 7. Test results from the DSP prototypes are given in Chapter 8. Finally, in Chapter 9, a set of conclusions are drawn from the research.

Chapter 2

Motivations for using Self-Timed Techniques

As mentioned in Chapter 1, the main motivation behind employing self-timed techniques in a high speed digital processor is to eliminate the requirement for a global synchronization signal or "clock". In this chapter, a more detailed analysis of the clock distribution problem is presented in order to fortify the argument behind researching the self-timed approach.

Motivation for developing a circuit technique that removes the need for a global clock signal centers in two areas. The first has to do with the way the design process of VLSI chips has evolved and how this affects the chip designer's ability to control the way the clock is physically distributed over the chip. The second is concerned with the trends in scaling of the digital IC process and the physical effects they have on clock distribution and skews.

1. Design Issues

Take it for now that limitations in the wiring layers of future IC processes cause a difficulty in limiting clock skews in VLSI circuits. What does this do to the design process? The designer, either of the circuit, or of the software that helps assemble the chip, is faced with a global concern. The placement of clock wires in the layout now affects the operation of the circuit in a critical way. The convenience of a hierarchical approach to the circuit design is not adequate because the evaluation of the performance is not really possible until the layout is completed. This adds a tremendous burden on the designer, who may not find out about fatal clock skews until the chip is fully designed, making changes more difficult. Thus, the first reason for investigating a self-timed approach is that by bringing the clocking signal generation to a local level (which is effectively what self-timed circuitry accomplishes), the chip design of complex systems can be significantly simplified since concerns over global synchronization are eliminated. This fits extremely well in the hierarchical chip design paradigm that is the basis of many CAD methodologies today. The designer is allowed to work at higher levels of description for a circuit and the design cycle time is substantially reduced.

Currently, there is a strong trend towards increased use of application specific ICs or *ASICs* which rely on the use of automated design tools. The tools characteristically deal with a cell library designed to meet a wide variety of applications. While the designer can easily manipulate basic logic cells to build up a circuit rapidly, there are some sacrifices with this process however, because the underlying cell design, chip layout, and routing are not entirely under the control of the designer. Therefore, the luxury of a finely tuned, hand laid-out clock distribution system is usually not possible. Hence, clock skew problems may not appear until the final circuit layout is complete.

In current synchronous chip designs, the sub-systems of the chip are made to work during the appropriate clock phase periods and the designer depends on the edges of those clock signals to be synchronized with the same clock signals fed to other parts of the chip. Self-timed circuits restore this level of hierarchy to a chip design well after clock skews become a problem in synchronous circuits since each sub-system on a chip may operate in its own time frame. As long as the timing signals within a block supervised by a handshaking circuit are aligned enough to ensure the correct operation of that block, the overall chip will operate correctly since those signals need not be synchronized with other signals outside the block.

2. Scaling and Technological Reasons

In the last section, the reader was asked to accept that the wiring delays of future IC processes limit the ability to distribute a global clock accurately. In this section, the physical reasons behind this assumption are presented in detail.

2.1 Scaling Laws

At the 2μ to 6μ level of design rules, scaling of the process could be performed without significant consideration about exceeding the physical limits of the materials involved. As scaling continues beyond the 1μ level, there must be more concern for materials limitations. For example, the higher electric field associated with a much thinner gate oxide and a given power supply voltage can cause problems with pinholes in the oxide [17]. As an analytical tool, several scaling strategies are often discussed in the literature. These strategies follow a set of simple rules on the scaling of dimensions and electrical parameters and they allow for comparisons. The most common VLSI scaling laws are shown in Table I. The scale factor α used in all entries in the table is assumed to be > 1.

TABLE I

Parameter	**Scaling Law**		
	Constant Field	Constant Voltage	Quasi-Constant Voltage
Vdd	Vdd/α	Vdd	$Vdd/\sqrt{\alpha}$
Horiz. Dimensions	$1/\alpha$	$1/\alpha$	$1/\alpha$
Gate Oxide	$1/\alpha$	$1/\sqrt{\alpha}$	$1/\alpha$
Doping	α	α	α
C_{ox}	αC_{ox}	$\sqrt{\alpha}C_{ox}$	αC_{ox}
C	C/α	$C/\sqrt{\alpha}$	C/α
R	$R\alpha$	$R\alpha$	$R\alpha$
RC	RC	$\alpha RC/\sqrt{\alpha}$	RC

In Constant Field (CE) scaling, all vertical and horizontal dimensions and the power supply voltage are scaled by the same factor α. Since the voltage and dimensions are scaled together, the internal electric fields remain unchanged. The scaling of the power supply limits the size of α if compatibility with existing TTL circuits is a consideration. CE scaling also significantly decreases the drain current of devices with channels shorter than $1\mu m$ due to impurity scattering effects.

Constant Voltage (CV) scaling is done by scaling horizontal dimensions but leaving the power supply voltage unchanged. The vertical dimension is scaled by a smaller factor of $\sqrt{\alpha}$. This results in better current drive capability and compatibility with TTL circuits. However, problems with shorter channel devices still exist due to the saturation of drift velocity. It is especially troublesome for sub-micron devices [17].

Quasi-Constant Voltage scaling (QCV) still shrinks horizontal and vertical dimensions by the scale factor α but the power supply is scaled by a smaller factor of $\sqrt{\alpha}$. In QCV, the drain current can increase down to sub-micron levels.

2.2 A real scaled process

The scaling laws above give some insight into future processes but it is rare that a real IC process would be scaled according to such simple rules. Therefore, some attempts were made to gather information about processes that are in design for the future. Estimates from this information for a 0.3μ design rule process are presented alongside parameters for some current processes in Table II [18]. Assumptions behind the figures in the table will be discussed in this section.

The power supply voltage is not likely to scale with each process for backward compatibility reasons. It is also highly desirable to maintain the ability to interface with standard TTL devices. Therefore, it is more likely that a new standardized supply voltage will be used by many different manufacturers for their future digital processes. The choice of 3.3V is logical since it can still drive TTL and it reduces the electric fields to a certain degree.

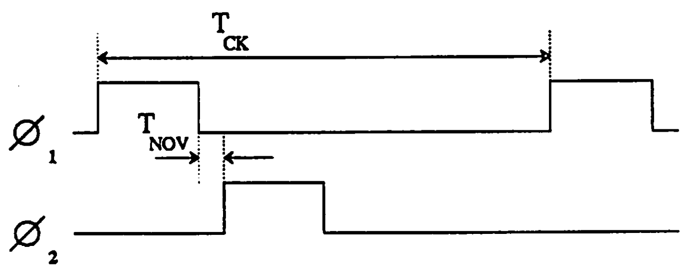
The goal of scaling is usually twofold in that besides just obtaining smaller area devices, the devices can be made faster. The g_m of the transistors will scale with the increase in C_{ox} of the gate oxide. Since the area of the device shrinks by the square of the scale factor, an increase in speed can be obtained. This is shown in the entries for t_{ox} and C_{ox} in the table. The problem of a scaled process then becomes ensuring that one can take advantage of the added speed. This is where the characteristics of the interconnect layers come into play.

2.3 Interconnect

The effect of scaling on interconnect wires has been studied extensively[16, 26, 27, 28, 30, 32, 33]. If the horizontal and vertical dimensions of a process are scaled by the factor α, the length, width, and cross sectional area of a conductor scale by α, α, and α^2 respectively. The smaller cross sectional area combined with the shorter length of a given

TABLE II

	1984	1986	1993	(est.)
↓Parameter/Process→	3μ	1.2μ	0.3μ	Notes:
L_{eff} (μ)	2.4	1.0	0.25	
Vdd	5V	5V	3.3V	
t_{ox} (A°)	500	250	85-100	
C_{ox} (fF/μ^2)	0.68	1.40	3.90	
$t_{Fox-POLY}$ (μ)	0.8	0.6	0.4	
t_{Fox-M1} (μ)	1.5	1.0	0.9	
m_1 pitch (μ)	4.5	1.5	0.7	(width/spacing)
m_2 pitch (μ)	6.0	2.0	1.0	(width/spacing)
m_1-substr cap (fF/μ^2)	.023	.035	.040	
m_2-substr cap (fF/μ^2)	.014	.020	.020	
m_1-m_2 cap (fF/μ^2)	.034	.040	.040	
m_i-m_{i+1} cap	NA	NA	.040	(All upper layers)
m_1 resistance $(\Omega/\square)$	0.038	0.17	.07-.17	(W - moly)
m_2 resistance $(\Omega/\square)$	0.026	0.025	0.075	
gate resistance $(\Omega/\square)$	50	40	2.5	(silicide)
τ_d	1.2nec	.32nsec	85psec	(FI=FO=3)
τ_{10pF} (ideal)	$5.6\tau_d$	$6.7\tau_d$	$7.8\tau_d$	
$T_{10pF-min}$	6.7nsec	2.2nsec	0.66nsec	
i_{CK}	50mA	125mA	320mA	$\tau_R \approx 2\tau_g$
T_{CK} (est.) of DSP	150nsec	30nsec	7.5nsec	
T_{NOV} 4ϕ	10nsec	2.8nsec	1.4nsec	(+ interconnect)
$4T_{NOV}/T_{CK}$	26%	37%	75%	



wire in the circuit will cause the resistance of the wire to increase by α as shown in Table I under QCV scaling. Similarly, the wire's capacitance to the substrate will decrease by α due to the smaller area of the wire combined with the larger unit area capacitance from the smaller vertical dimensions. The combined RC time constant of a wire therefore remains unchanged as listed in Table I. This has dire consequences for the speed of the entire circuit. While transistors are able to switch faster, the wires carrying signals between logic gates tend to have a more constant delay. In other words, the performance will be limited by the wires themselves ultimately.

The design of the interconnect becomes more complicated as dimensions are reduced for several other reasons that preclude the use of the simple scaling laws presented above. Smaller wires have some other associated problems. If a two-dimensional model is used to analyze the capacitance of a wire, it can be shown that the wire looks like a cylindrical wire plus a flat wire with no fringing effects (see Figure 2.1) [29]. As the wire is made thinner, the capacitance per length becomes asymptotic to that of the cylinder. A graph of the relationship is shown in Figure 2.2 for a wire and field oxide thickness of $1\mu m$. Therefore, despite the insulating layer thickness, scaling a wire size to reduce its capacitance has limitations. The asymptote of the capacitance curve is roughly $1.5pf/cm$.

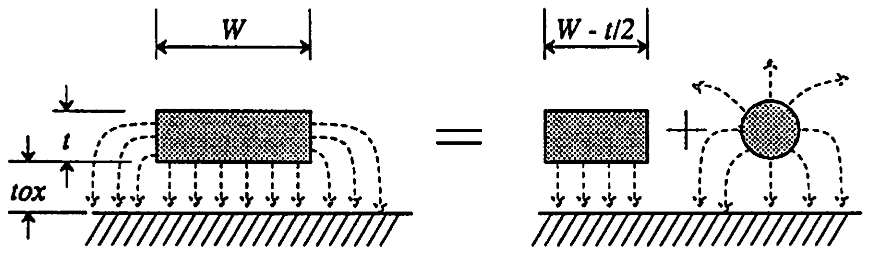


Figure 2.1: Two-dimensional model of wire capacitance.

Electro-migration

Aluminum conductors which have a high current density exhibit a phenomenon called *electro-migration* in which they tend to open over time because the molecules of Aluminum actually migrate away from their original position[28, 16]. It is generally accepted that a current density value of $J = 1mA/\mu^2$ should not be exceeded in the interests

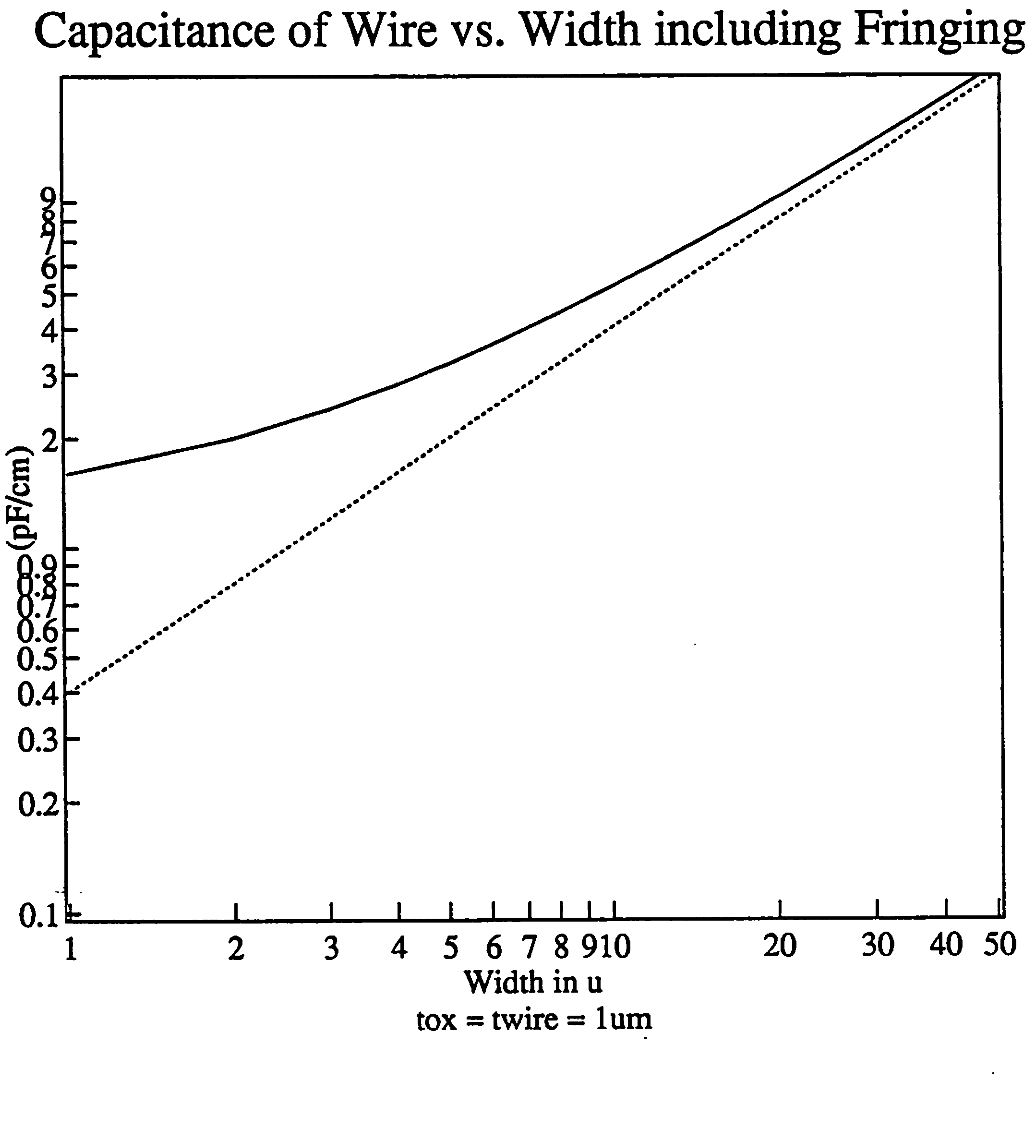


Figure 2.2: Capacitance of wire/unit length when two-dimensional effects are included.

of reliability[28]. Usually, the conductors carrying the power supply over the surface of the chip are made wider to both avoid voltage drops and electro-migration. How does scaling affect this problem? As the transistors are scaled down in size, the speed of the circuits increases. For a signal such as a clock therefore, the slopes of the clock edges must increase as the period of the signal decreases. Since the load on a clock line can be quite large, the peak charging current in the line scales up with decreasing device sizes. Figure 2.3 shows the charging current in a wire versus the capacitance of the wire. The charging model used is a simple linear charging $i = CV/T$ and the voltage and charging time are the values of Vdd and $\approx 2\tau_d$ taken from Table II. Using this conservative model, it can be seen in the figure that a charging current i_{CK} of hundreds of mA is required for the capacitance on a typical clock line, for example $10pF$.

Aluminum wires are usually made about $1\mu m$ thick and the rule followed to avoid electro-migration is that the wire should be $1\mu/mA$ wide. Figure 2.4 shows the capacitance of AL wires versus their length when the rule to avoid electro-migration is followed. Clearly, AL is inappropriate for dynamic signal lines that have any appreciable capacitance. Even when made wide enough to avoid electro-migration (using excessive area), the wires then add so much capacitance that the delay time would be prohibitive.

The problems with AL conductors can be avoided by using different materials for large current carrying wires. Some current processes employ molybdinem (moly) as one of the metal layers. Tungsten is another likely choice for future processes. Both of these materials avoid the electro-migration phenomena at the expense of a higher sheet resistance. Where Aluminum has a typical sheet resistance of $0.025\Omega/\square$, Tungsten and moly are roughly $0.07\Omega/\square$ and $0.17\Omega/\square$ respectively[30]. The higher resistance of Tungsten and moly adversely affect the RC time constant of the wires and while electro-migration is not a problem, wires with large peak currents still must be made wider to avoid severe voltage drops (causing further delays) on signal edges. Figure 2.5 illustrates this. For a Tungsten wire, the minimum width required to maintain a voltage drop of less than $1.5V$ is shown for charging currents of $10mA, 50mA$, and $100mA$. For a longer wire traversing the surface of the chip, the width must still be large for Tungsten - again adding more parasitic capacitance.

A more likely strategy for advanced IC processes will be to use a greater number of interconnect layers as opposed to further scaling of the layers currently in use. The design rules for the most compact interconnect layer will tend to stabilize at about $1\mu m$

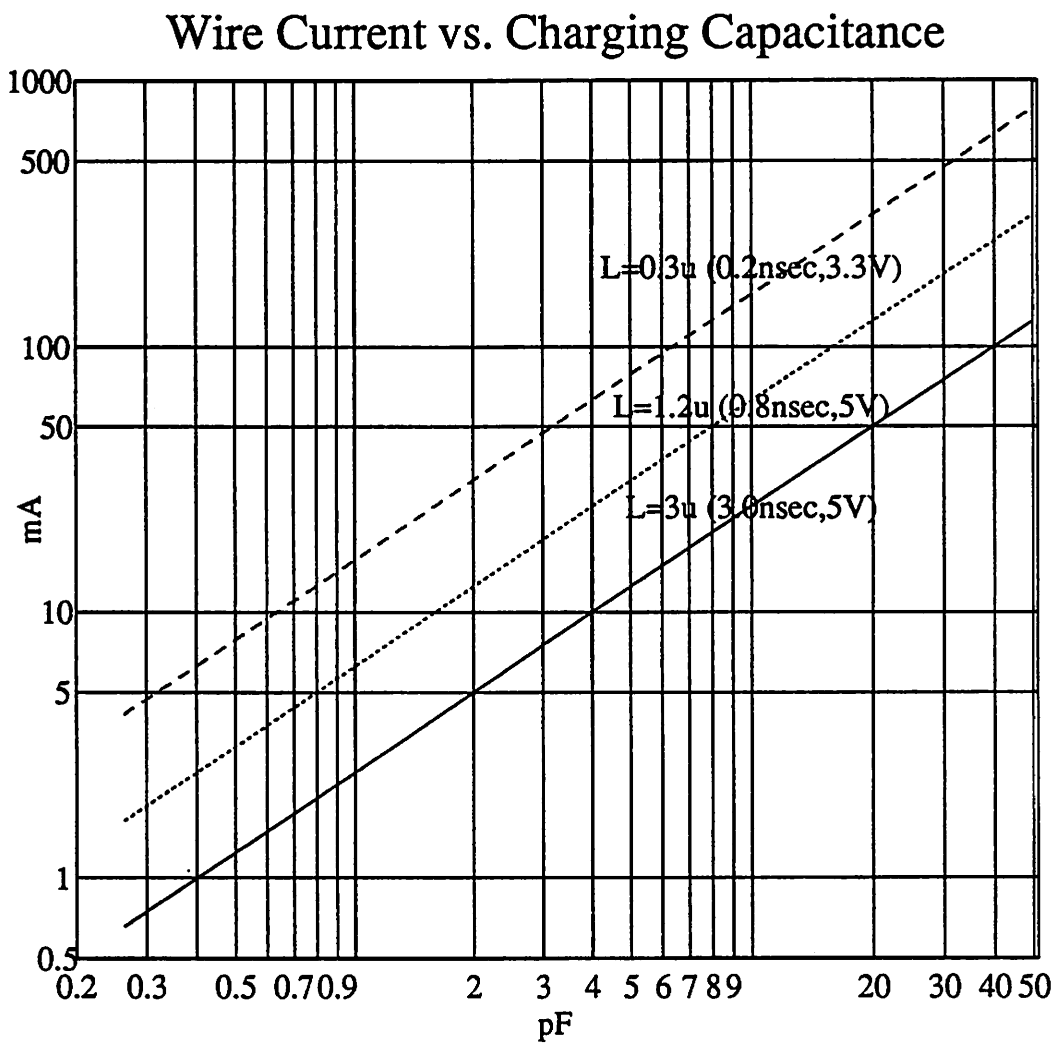


Figure 2.3: Charging current versus charging capacitance using a linear model.

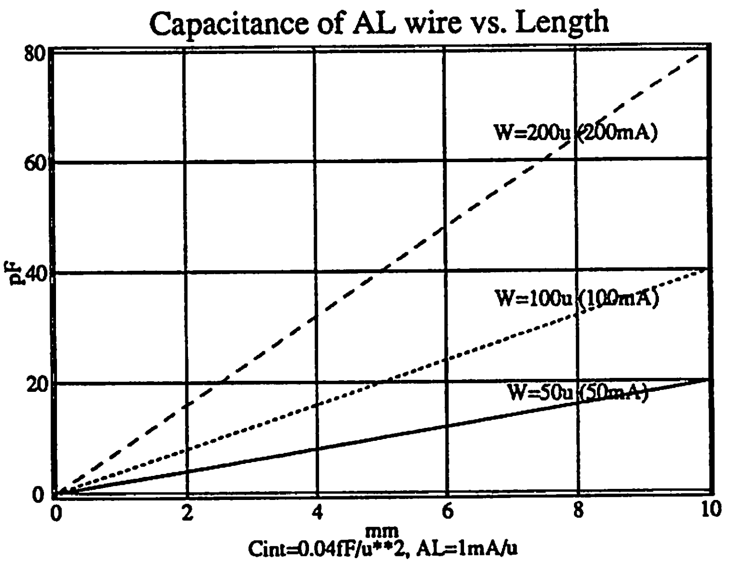


Figure 2.4: Capacitance of AL wire versus length when $1\mu/mA$ rule is followed. Charging currents are shown for each line.

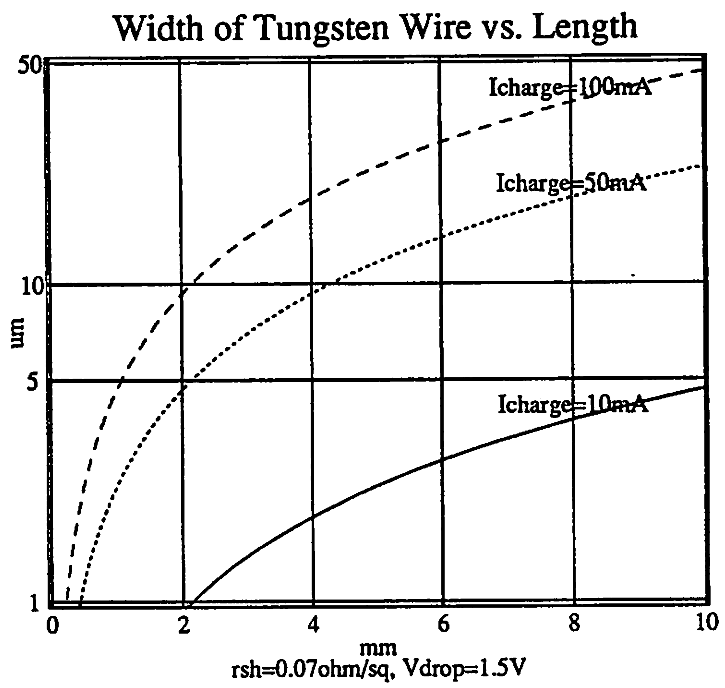


Figure 2.5: Minimum width of a Tungsten wire required to keep a voltage drop of less than 1.5V versus length and charging current.

width and spacing. The upper layers may increase in size and thickness in order to handle power distribution more effectively [23]. (Capacitance of layers distributing the power supply is not really a factor. In fact, it actually helps because it acts as a supply bypass capacitance) This scenario is shown in Table II, where the $metal_1$ width and spacing are not scaled beyond $0.7\mu m$. An inter-layer capacitance of $0.04 fF/\mu^2$ is shown for all layers above $metal_1$. This represents layers of field oxide that are approximately 100 times greater in thickness that the gate oxide.

2.4 Clock Distribution

Now that we have a model for the interconnect layers of future processes, let's examine the effects that the interconnect has on clock distribution. The responsibility of a clock distribution network on a synchronous IC is to send the timing signals to all parts of the circuit without introducing a time difference between the signals at their destination. The goal after all is to synchronize all operations in the system with the master clock. A skew between clocks at different portions of the circuit can cause errors to occur in the operation of the system. Typically, in a two or four-phase clocking scheme, the skews are compensated for by adding a non-overlap or "dead" time between the clock phases. If the non-overlap time is greater than the maximum clock skew time, then errors are prevented by ensuring that all clock signals go LOW before the next clock phase signal goes HIGH . The non-overlap time directly subtracts from the time available to perform useful computation during each clock period, therefore it is desirable to limit it to the minimum amount needed. Figure 2.6 shows a model for a register based system. It is denoted "Full Cycle Synchronous Pipeline" because data is transferred once per full cycle of the clock. The lower drawing in the figure expands the registers themselves. Definitions for the clock waveforms themselves are given in Figure 2.7. In a two-phase system, the two clock lines ϕ_1 and ϕ_2 are separated the non-overlap time T_{NOV}. In some systems, only a single phase clock is distributed. In this case, the two phases used in the registers become ϕ and $\bar{\phi}$ and the inversion is done locally. Figure 2.7 shows a skewed ϕ_2 waveform also and the skew time T_{SKEW} subtracts from the nominal T_{NOV} time. Note that it is the skew *between* stages that is important. For a single clock system, the skew between ϕ^i and $\bar{\phi}^i$ in stage i is well controlled by the designer. However, it is the skew between ϕ^i and $\bar{\phi}^j$ that can cause problems when stage i communicates with stage j. Thus, the equations

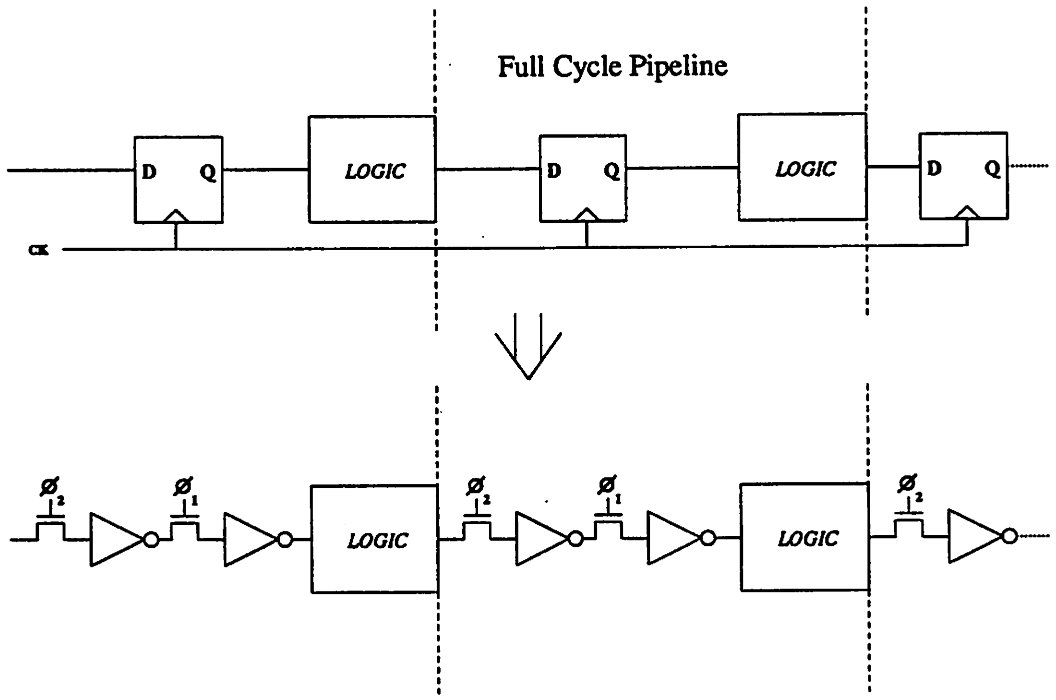


Figure 2.6: Full Cycle Synchronous Pipeline model.

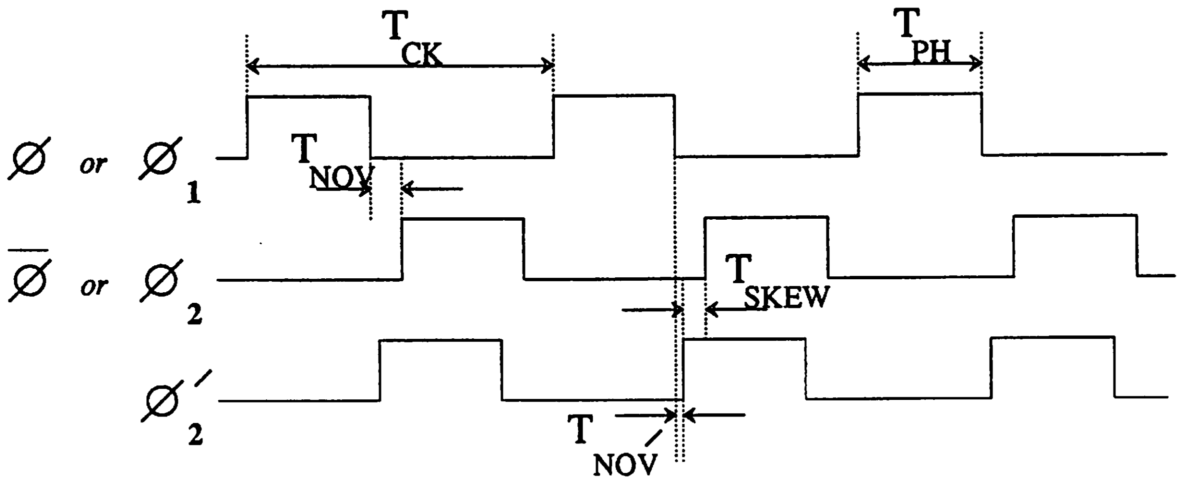


Figure 2.7: Clock waveform definitions.

below apply to both a two-phase and single-phase system, the single-phase system being the special case when T_{NOV} is exactly equal to 0. For the skewed waveforms in Figure 2.8, correct operation requires that

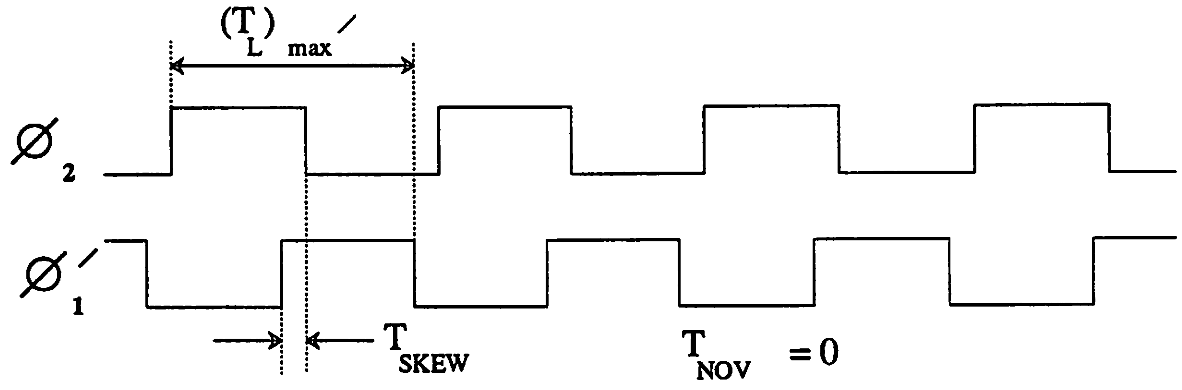


Figure 2.8: A two-phase clock with skew.

$$T_{SKEW} < (T_L)_{min} + T_{NOV} \ or \ (T_L)_{min} > T_{SKEW} - T_{NOV} \tag{2.1}$$

where T_L is the time for a logic block output to reach the threshold voltage of a gate after the input changes (typically less than τ_p, the propagation time). $(T_L)_{min}$ is the minimum T_L in the system. Referring to Figure 2.6 again, Equation 2.1 is intuitive. During ϕ_2, the master latch of the registers is active reading the output of the preceding stage. If there is skew such that ϕ_1 of the preceding stage rises before ϕ_2 drops as in Figure 2.8, then the output of the preceding could change and wipe out the correct value that was in the latch. Adding more T_{NOV} alleviates the problem, but from Equation 2.1,

$$(T_L)_{max} < 2T_{PH} + T_{NOV} - T_{SKEW} \tag{2.2}$$

In other words, the maximum time for computation in a stage is similarly reduced. Some systems contain a finer partitioning of the logic blocks as shown in Figure 2.9 and denoted "Half Cycle Synchronous Pipeline". In this case, the constraint on T_{SKEW} is identical to the full cycle case as shown in Equation 2.1. The constraint on $(T_L)_{max}$ is a bit more complicated in the half cycle. A stage that computes during ϕ_1 can actually take longer than a single clock phase to complete. As long as its output becomes valid before ϕ_2 falls, the next stage will acquire the correct data. However, the next stage may not have a

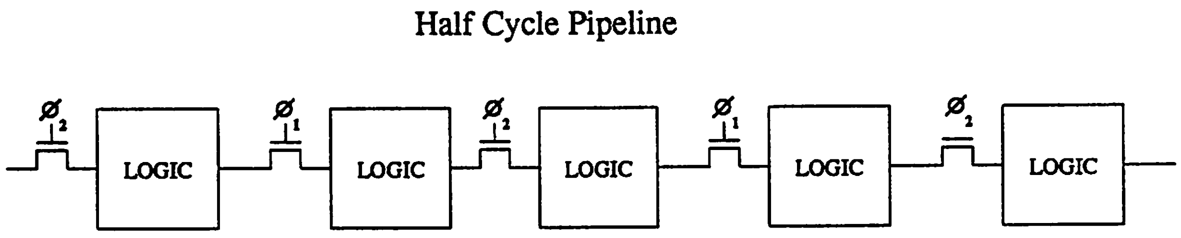


Figure 2.9: Half Cycle Synchronous Pipeline model.

similarly long delay time since it does not have valid inputs until the end of ϕ_2. Thus, the aggregate delay times of adjacent stages must not exceed a clock period. When a pipeline is dominated by a single "slowest" stage which is usually the case, the $(T_L)_{max}$ of that stage must be less than shown in Equation 2.2.

Clock distribution is aggravated by such things as growing chip size, shrinking feature sizes, interconnect time delays and any irregularity in the loading scheme. If the time to get a signal from point "a" to point "b" on a chip is proportional to the distance, then growing chip size implies more time for global signal distribution. The clock distribution problem tends to be less important in certain regular structures. For example, in gate array or sea-of-gates chips, the structure of the chip is very regular and clock drivers can be designed in a tree fashion to equalize delays to all parts of the circuit. [21] Also, in certain pipelined subsystems, where the clock skews can essentially follow the data in the pipe, the clock distribution is less of a problem [75, 76] In both of these cases however, synchronization with off-chip devices is still a serious problem. One system approach to the synchronization problem that is more circuit intensive is to construct phase-lock loops at the clock inputs to each stage (chip) in order to re-establish synchronization at each boundary[77]. When a cell-based DSP design is employed, and a general structure allowing feedback is allowed, then the loading and the layout structure can easily become quite irregular causing the design of the clock distribution to be much more difficult and dependent on global constraints.

The simplest form of clock distribution is to take the clock signal (generated on-chip or supplied from an off-chip source) and buffer it so that it can supply the load of the entire chip's clock signal inputs. All the clock lines of the chip are tied together on a single electrical net and connected to the large buffer. The clock skew will be dependent solely on the delays associated with the interconnect that ties all the clock inputs to the clock buffer. Other forms of clock distribution depend on both wiring delays and internal buffer delays. These will be discussed below after the delays due to wiring alone are quantified.

Wiring Delays

An analytical model for approximating wire delays in MOS integrated circuits was published by Sakurai [31]. The basic model has a distributed RC line for the interconnect wire with a driving MOSFET at one end and a capacitive load C_L at the other end as shown in Figure 2.10. Inductances are not considered a significant factor for *on-chip* conductors compared to the other circuit parameters which dominate[1]. The wire length is L and R is the total resistance of the wire ($= r \cdot L$), where r is resistance per unit length. Similarly, C is the total capacitance of the wire ($= c \cdot L$), where c is the capacitance per unit length. The driving transistor can be replaced by an equivalent resistance R_{tr} and the circuit is driven by a step for the analysis. Sakurai states that a good choice for R_{tr} turns out to be 1/(maximum drain conductance) of the driving MOSFET. While this is only an approximation, it is a good representation of a typical clock distribution circuit. The driving resistance is the output impedance of the clock buffer and the load capacitance is the sum of the capacitances which the clock buffer feeds. For this discussion, an exact solution is not as important as getting a feel for what the basic limitations are when sending a signal through wire on an IC.

Some other symbols used in the analysis are $C_T = C_L/C$, $R_T = R_{tr}/R$, and $t' = t/CR$, all normalized values to the interconnect time constant. Also, s' is the Laplace transformed variable for t'.

When a voltage step is applied to the gate of the drive transistor, the response at the load capacitance C_L, $V_2(s')$, is written as

$$V_2(s') = \frac{V_{dd}}{s'} T_D(s') \tag{2.3}$$

where $T_D(s')$ is the transfer function of the distributed RC interconnect wire. Denoting the poles of Equation 2.3 as $0, \sigma_1, \sigma_2, \cdots, \sigma_k$, the response in the time domain $v_2(t')$ can

[1]Of course, bonding wire inductances are quite significant for clock and power connections to the outside world.

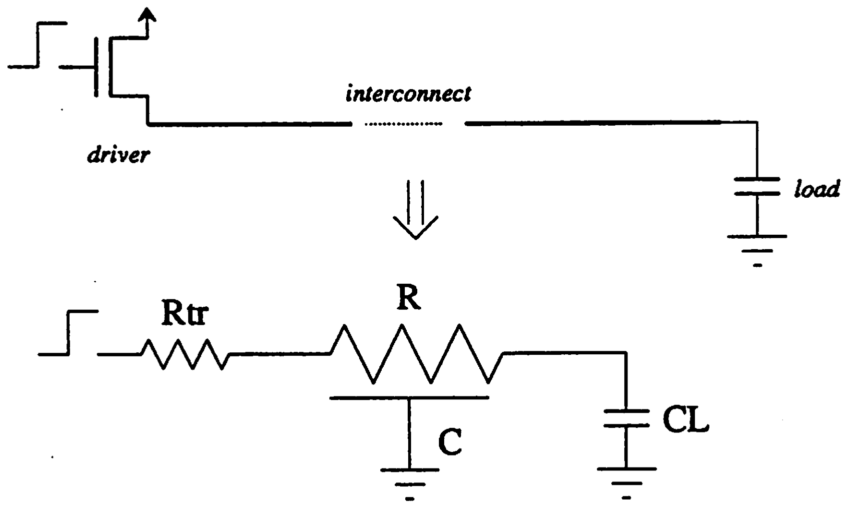


Figure 2.10: Model for interconnect delay time.

be expanded in multi-exponential form as follows by using Heaviside's expansion theorem:

$$\frac{v_2(t')}{V_{dd}} = 1 + \sum_{k=1}^{\infty} C_k e^{-\sigma_k t'} \tag{2.4}$$

where

$$C_k = (-1)^k \cdot \frac{2\sqrt{(1+R_T^2\sigma_k^2)(1+C_T^2\sigma_k^2)}}{\sqrt{\sigma_k}\{(1+R_T^2\sigma_k^2)(1+C_T^2\sigma_k^2)+(R_T+C_T)(1+R_TC_T\sigma_k)\}} \tag{2.5}$$

from the transfer function $T_D(s')$ of a distributed RC line. The $\sigma_k's$ are the solutions to the following equation:

$$\tan\sqrt{\sigma_k} = \frac{1-R_TC_T\sigma_k}{(R_T+C_T)\sqrt{\sigma_k}} \tag{2.6}$$

Equation 2.6 can be solved exactly only when $C_T = R_T = 0$. In other cases, the solution must be numerical. It can be shown that an excellent approximation for Equation 2.4 is

$$\frac{v_2(t')}{V_{dd}} = 1 + C_1 \cdot e^{-\sigma_1 t'} \tag{2.7}$$

Numerically calculated values for C_1 and σ_1 can be found in [31]. Given that the accuracy of Equation 2.7 is high[2], the 90% time delay between the input step function and the voltage at the load capacitance C_L is easily found from

[2]The magnitude of the next term at the time when $v_2(t')$ is at the 90% level is less than 10^{-9}.

$$t'_{0.9} = \frac{1}{\sigma_1} \ln \mid 10C_1 \mid \tag{2.8}$$

When Equation 2.8 is plotted, the delay time is seen to be proportional to R_T or C_T when they are large, the drive resistance or load capacitance dominating. When R_T and C_T are small, the delay is almost constant, limited by the wire itself. In that case, there is an almost linear dependence of $t'_{0.9}$ on both R_T and C_T. Sakurai wrote the equation for $t'_{0.9}$ with only linear terms as $a + bC_T + bR_T + cR_TC_T$ and calculated the constants a, b, and c to minimize the error in the range $R_T, C_T < 1$. The result was

$$t_{0.9}/CR = 1.02 + 2.21(R_TC_T + C_T + R_T) \tag{2.9}$$

$$\Rightarrow t_{0.9} = 1.02\, RC + 2.21(R_{tr}C_L + RC_L + R_{tr}C) \tag{2.10}$$

The relative error of the formula above was found to be less than 4 percent for any value of R_T and C_T. For evaluating quantities such as gate delays and clock skew, the 50% delay time is a more useful measure[3]. Equation 2.7 is a simple exponential and therefore the $t_{0.5}$ time should be Equation 2.10 multiplied by $\ln(2)/\ln(10) = 0.30$. The value 0.32 was actually used after some verification using the SPICE circuit simulator. So, the formula

$$t_{0.5} = 0.33\, RC + 0.71(R_{tr}C_L + RC_L + R_{tr}C) \tag{2.11}$$

was plotted in the following graphs as an approximation for the 50% delay time associated with an interconnect wire driving a capacitive load. The following assumptions also apply to the graphs:

1. Wire parasitics: $R_{sh} = 0.07\Omega/\square$, $C = 0.04fF/\mu^2$.
2. $R_{tr} = 1/g_m$ of driver transistor in ohms.
3. Delay of driver transistor itself is *not* included.

Therefore, Equation 2.11 is the delay of the wire *only* since it does not include any time for the driver transistor channel to form or for any of the buffer stages that precede the driver transistor to switch.

[3]The 50% delay time is the time difference between the input and output waveforms reaching the gate threshold value, typically $V_{dd}/2$ volts in a CMOS circuit

Using the approximation described above, some common clock distribution schemes can be studied to make a determination of the potential clock skews that would be encountered. As mentioned before, the simplest scheme would be to have a series of buffers increasing in size so that the last buffer would be large enough to drive the capacitance associated with all of the clock inputs and wires in the circuit. In this case, the value for C_L is large. For chips that are 5-10mm on a side, the longest clock wire can easily reach 1-2cm. Figures 2.11 and 2.12 show the delay of a 1cm and 2cm wire when driving a 10pF capacitive load. The delay is plotted for different size driver transistors (R_{tr}) and versus the width of the wire itself. There is an optimum wire width to use for a specific load

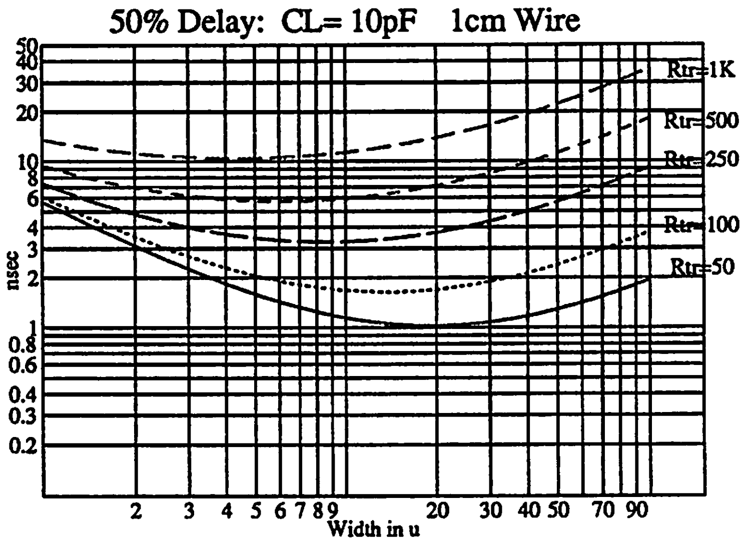


Figure 2.11: Delay time associated with a 1cm wire driving a 10pF capacitive load.

and wire length. The smaller wire widths increase the overall delay time due to the term RC_L dominating in Equation 2.11. If the wire is made too wide, the delay increases when the term $R_{tr}C$ begins to dominate. The optimum width for this load is between 15-25μ. Even for an optimum width, the delay times are roughly $1nsec$ and $2nsec$ for a 1cm and 2cm wires respectively.

In Table II, the entry τ_{10pF} shows the minimum number of gate delays in which the clock buffer driving a 10pF load could switch. It assumes the optimum sizing of each stage of the buffer where the device lengths are made $e^{1.0}$ larger than the previous stage. The entry $T_{10pF-min}$ just multiplies by τ_d to get the minimum delay time of a buffer

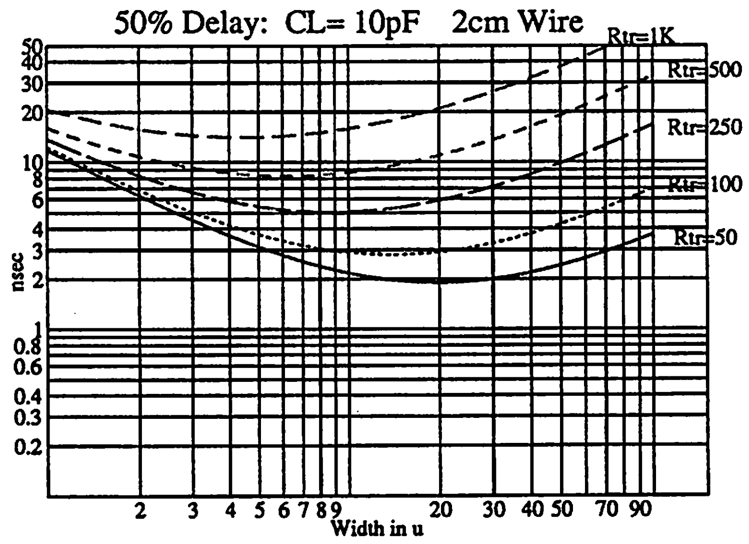


Figure 2.12: Delay time associated with a 2cm wire driving a 10pF capacitive load versus wire width.

driving 10pF.

A more common method of distributing the clock is shown in Figure 2.13. Rather than driving the full clock load directly, a distributed buffer system is employed. By having less of a load on the longest clock wire that originates from the clock generator, there is less charging current and the wire can be smaller. In the hypothetical situation suggested in Figure 2.13, the 10pF load has been split into ten 1pF loads, each having its own buffer. The input capacitance to each of the distributed buffers is 0.1pF. therefore, the long interconnect wire distributing the clock has a 1pF load. Figures 2.14 and 2.15 show the delay associated with a wire driving a 1pF load. The minimum delay occurs for a less wide wire since the C_L term is reduced. However, the delay still is significant. Additionally, in spite of the reduced interconnect delay, the buffers driving each 1pF load have a delay that would have to be taken into consideration if data were supplied off-chip from one of the stages. This is illustrated in the figure. The buffers driving a 1pF load have a minimum delay of $5.5\tau_d$ in a 0.3μ process which is $0.46nsec$. Therefore, the overall delay which has to be compensated for with non-overlap time is still in the $1 - 2nsec$ range. In fact, for comparison, it is interesting to look at the delay of the interconnect wire when driving a relatively small 0.1pF capacitive load. Figures 2.16 and 2.17 show

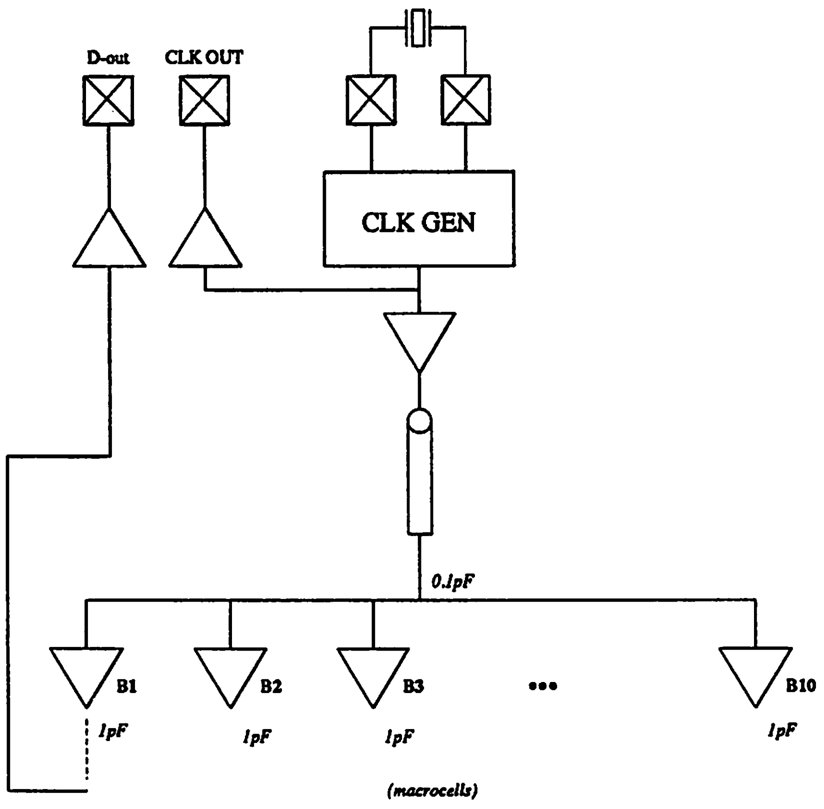


Figure 2.13: Distributed buffer clock distribution scheme.

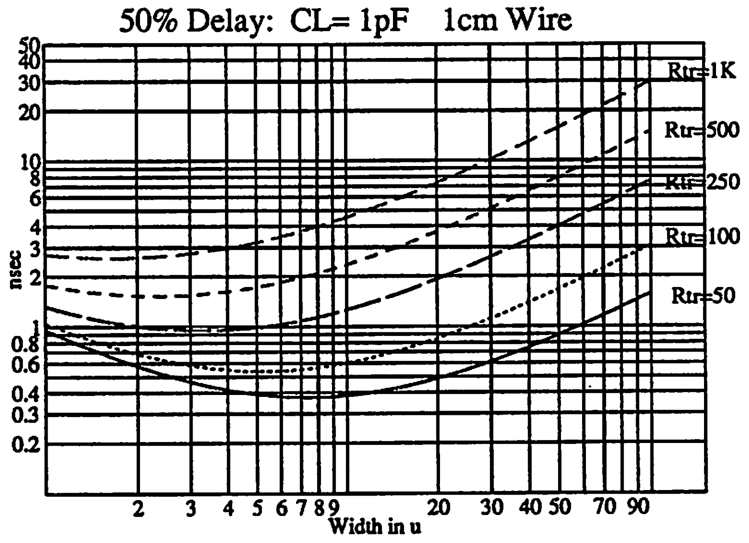


Figure 2.14: Delay time associated with a 1cm wire driving a 1pF capacitive load versus wire width.

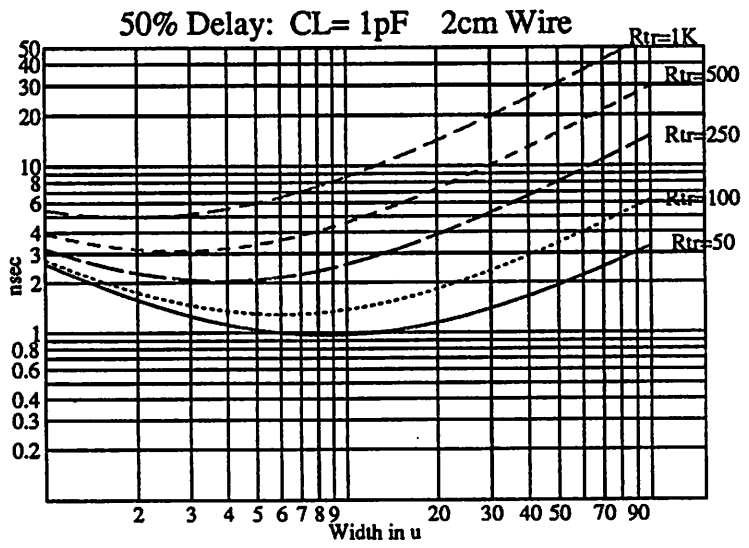


Figure 2.15: Delay time associated with a 2cm wire driving a 1pF capacitive load versus wire width.

this. While the minimum delay for a 1cm wire shown in Figure 2.16 is about $0.25nsec$,

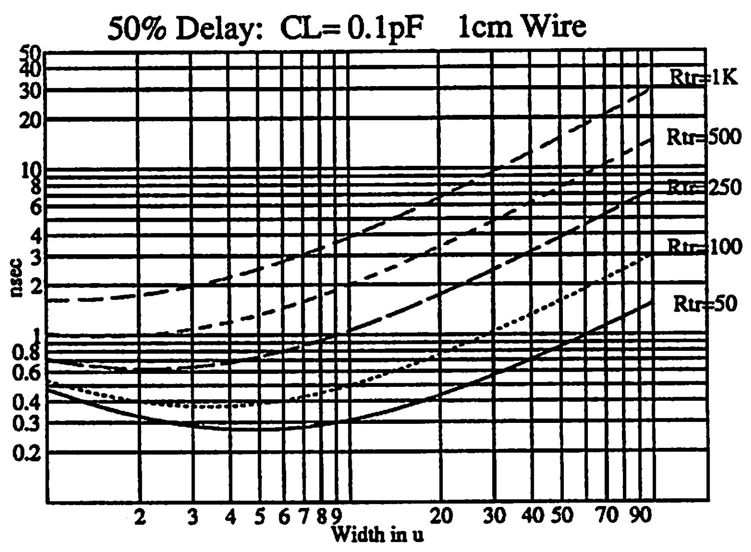


Figure 2.16: Delay time associated with a 1cm wire driving a 0.1pF capacitive load versus wire width.

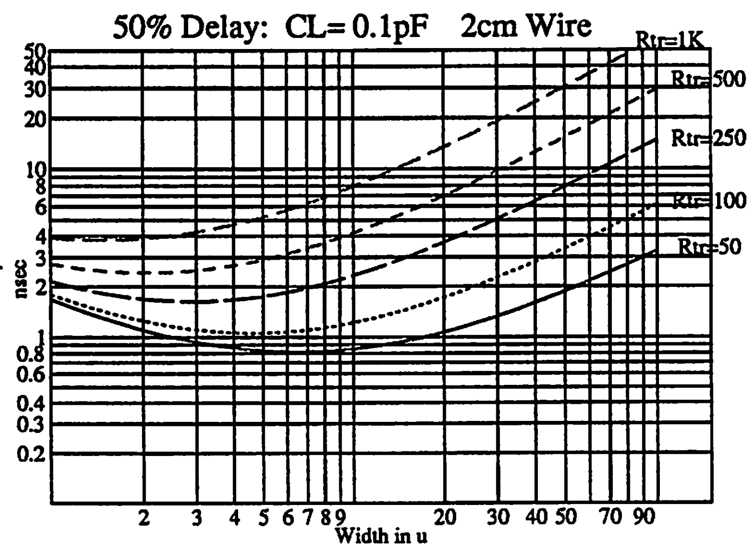


Figure 2.17: Delay time associated with a 2cm wire driving a 0.1pF capacitive load versus wire width.

note that the value of R_{tr} to obtain such a small delay is 50Ω. A transistor W/L of roughly

100 would be required to obtain such a small drain resistance! The time required to buffer the input to such a large device would be considerable. The graphs therefore demonstrate that regardless of the way the load is distributed, the time required to transfer a clock signal across the chip does not fall rapidly below a certain value. As Table II shows, as the clock frequency is increased for future processes (T_{CK} falls), the percentage of that period required for non-overlap time to prevent clock skew problems increases dramatically, leaving little time for actual computation.

3. Summary

The motivation for investigating self-timed circuitry for DSP designs comes from two main areas. First, as the scaling of the digital IC process continues, the speed (and layout) of circuit designs tends to become interconnect limited. The layers that connect devices together do not scale as well as the devices themselves such that the RC time constant delays of these layers dominate. Additionally, the trend towards an increased number of interconnect layers makes the characterization of the parasitic loading on a wire extremely difficult. Clock distribution in a cell based design therefore has associated with it certain finite delays which can cause a skew in the clock waveform between different parts of the circuit. Another reason for investigating self-timed circuitry is that CAD based chip development systems are suited for hierarchical designs. As the digital process scales, the problems of distributing the clock become more dependent on global conditions of the chip. Clock skew problems may not even be characterizable until the layout of a chip is completed. Self-timing allows the hierarchical design style that is currently used for synchronous designs to be used in spite of process changes. By eliminating the global clock distribution problem, highly developed CAD tools can be used and correct circuit operation after layout is ensured.

Chapter 3

Realization of Self-Timed Circuits

The minimum requirement of a self-timed circuit is that it generate some kind of *completion signal* indicating that its outputs are valid. This is usually required each time a new set of inputs is applied to the circuit. There are several approaches to designing a circuit that supplies completion information and in this chapter, they will be examined. One logic family, called DCVSL, provides a simple means for generating a completion signal via a simple extension to the basic circuitry. This logic family is explained in detail since it was chosen for the DSP design presented in the following chapters. Some alternative structures that can supply completion information are also surveyed.

1. Completion Signals

A conceptually simple method for generating a completion signal for a block of circuitry is to exploit the matching characteristics of an integrated circuit and duplicate the circuit block, using the duplicate copy solely for the purpose of generating the completion signal. This is usually done by supplying the duplicate block with a known input signal or vector. The output signal is then also known in advance and when it appears, it should indicate that the circuit doing the actual computation has valid outputs. This approach can work under certain conditions although it should be noted that it is not strictly a self-timed circuit because it relies on the matching characteristics. The success of using matching or "dummy" circuits to determine completion times depends on the actual circuitry involved and the environment in which it is placed. For example, if the circuitry has a delay that depends on the input data, the matching technique suffers since the circuit producing the completion signal always receives the same pre-determined input data. While this might be chosen to yield the worst-case delay of the circuit, then the system efficiency is hurt in the same way as in a synchronous design, where the clock period must reflect the worst-case delay of the elements. Another risk is in a design where the loading of the logic cells is not known in advance. In parameterizable logic cells, the loading on logic cell outputs can vary over a wide range. Unless the dummy circuit exactly duplicates the original, requiring a 100% area overhead, it may not accurately reflect the delays. Another example is when the block is driving bus wires to other blocks. Again the loading may not be determined until the chip layout is complete. Therefore, the use of matching is best restricted to local, well defined pieces of logic where the matching is indeed determined by the IC process rather than circuit loading and parameters.

Some asynchronous schemes have proposed the use of "spacers"[2] or multi-valued circuits[3] as a way of encoding data lines so that data-detectors can detect valid outputs. A spacer is a word that cannot normally appear in the data stream. It is placed at the boundaries of valid data words so those boundaries can be detected. In a similar way, multi-valued logic circuits use a value outside of the normal binary signal levels to indicate the boundaries. The overhead of encoding the data with spacers severely limits the practicality of the method. Similarly, the difficulty in implementing multi-valued logic with good noise margins has limited its usage.

A more hardware-oriented method for encoding data lines is to use *dual-rail coding*[19]. This is really a bit-wise extension to implement the multi-valued coding idea. For a single bit of data, two wires are used to indicate its value. The HIGH value is indicated by 10, the LOW value is indicated by 01, and the value 00 is reserved for non-valid or boundary states. Dual-rail coding can be implicit in the operation of certain differential logic families. One logic family introduced a few years ago named Differential Cascode Voltage Switch Logic or *DCVSL* [34, 35, 36, 37, 41, 42] follows a behavior that makes the generation of a completion signal a very simple and efficient extension to the basic circuitry.

2. DCVSL Description

Differential Cascode Voltage Switch Logic is a pre-charged (also called *dynamic*) logic family that is very similar to a differential form of *domino* logic [49]. Domino logic and NORA logic [50] both gained popularity in recent years because they are typically less area consuming than static CMOS logic due to the fact that they use mostly only one type of device (NMOS or PMOS) to implement the logic function. This is contrasted to static CMOS which has dual NMOS and PMOS device trees to implement a logic function. The

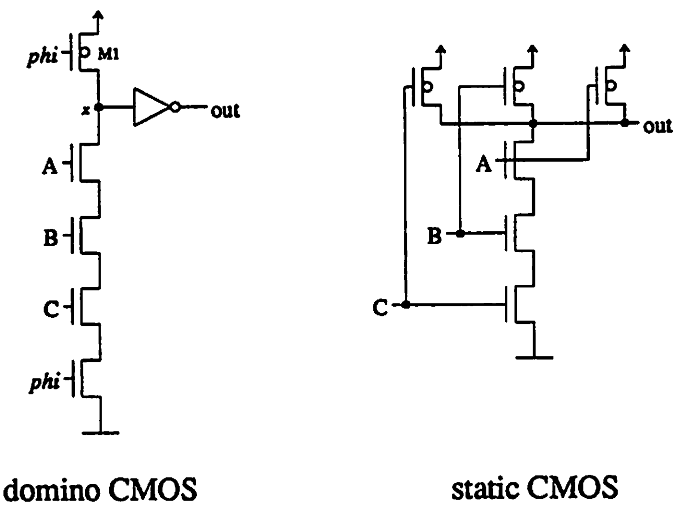


Figure 3.1: A simple domino logic gate and its static CMOS counterpart.

speed of a domino gate is enhanced because the input capacitance of the gate is reduced over a static CMOS gate since only NMOS devices are driven. Figure 3.1 shows a simple domino AND gate along side a static CMOS version of the same gate. In the precharge stage of operation, the clock signal *phi* is low and the internal node x is charged high by PMOS device M1. When *phi* goes high, the evaluation stage is entered and node x is discharged to ground only if A, B, and C inputs are high. The node labeled x must be buffered in order to drive another domino gate and therefore domino logic gates are non-inverting.

A generalized DCVSL gate is illustrated in Figure 3.2. The gate also has an NMOS tree which implements the required logical function but there are two outputs. Similar to domino logic, there is pre-charge phase (I high) where both nodes f and $fbar$ at the top of the NMOS tree are charged high by PMOS devices M1 and M2 respectively. During evaluation however, only node f *or* $fbar$ will be discharged causing one of the two outputs to go high. The NMOS tree is designed in such a way that for valid input signals (complementary), only a single output will become high. The pre-charge/evaluate

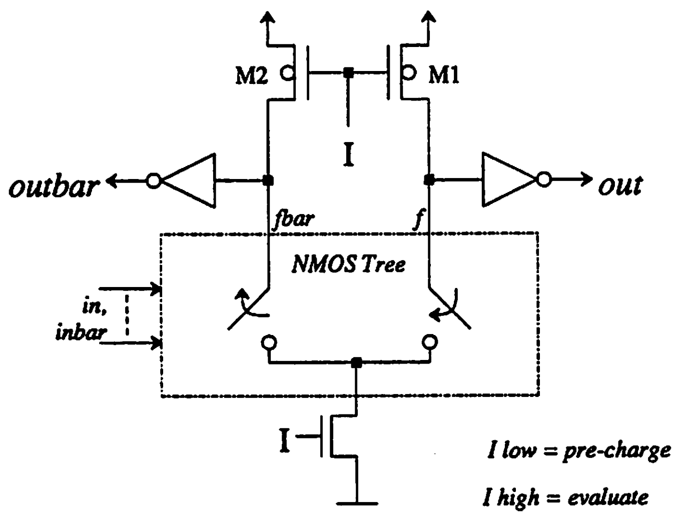


Figure 3.2: A generalized DCVSL gate.

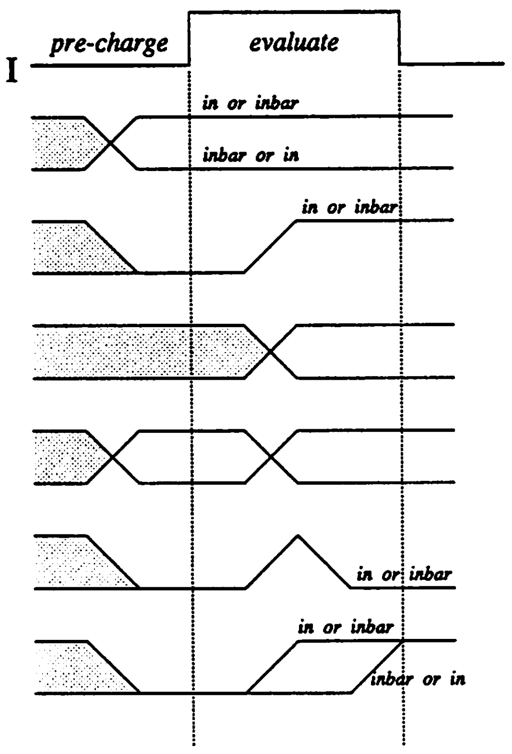
signal is labeled *I* in this figure for historical reasons; it is short for I*nitialize.* As with domino logic, the complementary outputs of one DCVSL gate can directly drive another DCVSL gate [1]. Unlike domino logic, a DCVSL can be inverting or non-inverting since inversion is simply achieved by permuting the outputs. The NMOS tree in a DCVSL gate does *not* always contain twice the number of devices as that of a single ended gate. In fact, the area of the NMOS tree grows at a slower rate as the logical function becomes more complicated. This will be shown in a following section but a simple explanation for this is that by allowing connections between the two sides of the NMOS tree, transistors can be shared so that the implementation of the logical function and its complement becomes more efficient than just doubling the number of devices.

As with domino logic, there are constraints on the timing of the inputs to a DCVSL gate. Improperly generated input signals can result in either an incorrect output value or a disallowed output value i.e., both outputs being high. The constraints on the inputs to a DCVSL gate are shown in Figure 3.3.

2.1 Completion Signal Generation in DCVSL

The key to exploiting DCVSL logic for self-timed circuits lies in the dual rail coded nature of the output signals. During each pre-charge/evaluate cycle, the outputs

[1]This is the origin of the term "domino" logic. When a cascade of domino gates is allowed to evaluate, the outputs become valid in sequence, imitating the behavior of a row of falling dominos.



Waveform	Allowed?	Comments
	Yes	Stable before evaluate phase
	Yes	One Transition, Low to High Both Inputs Low before evaluate
	No	Transition on both inputs during evaluate phase
	No	Two Transitions
	No	Glitch, two transitions
	No	Transition on both inputs Both Inputs Low before evaluate

Figure 3.3: Constraints on the inputs to a DCVSL gate.

of a DCVSL gate return to zero and then become complementary. By placing a single OR gate across the two outputs of the gate as shown in Figure 3.4 a completion signal is successfully generated. In this document, the label DV is used for completion signals and it stands for "D*ata* V*alid.*" The output of the OR gate in Figure 3.4 has dual importance since besides indicating when data is valid at the outputs, when low, it indicates that the pre-charge is complete. Hence, it is labeled DV/PC^* in the figure.

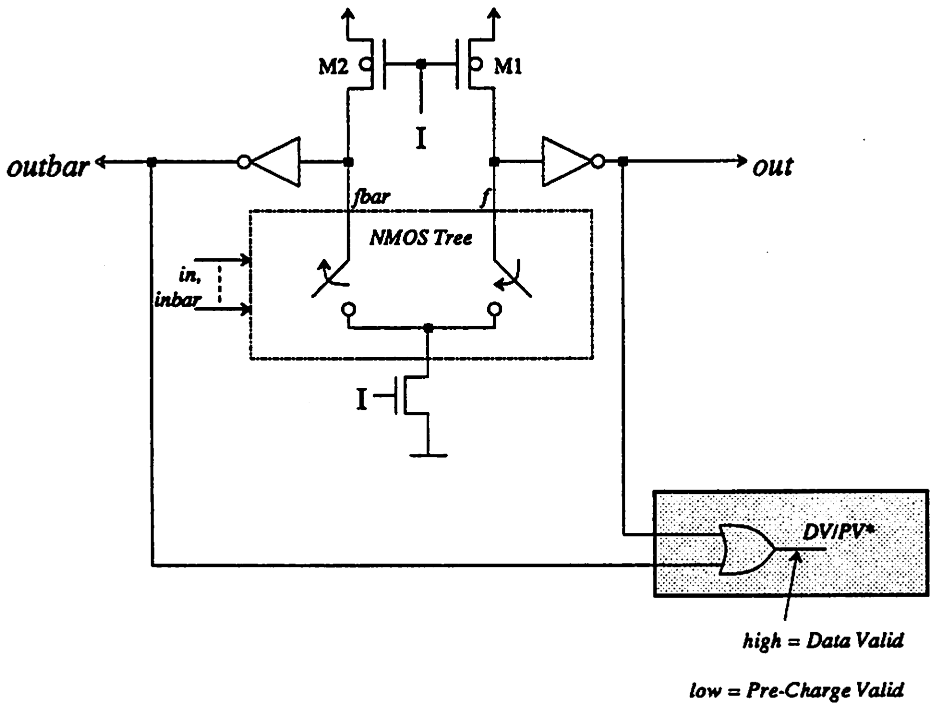


Figure 3.4: The generation of a completion signal on a DCVSL gate. DV denotes "data valid" and PV denotes "pre-charge valid".

In a combinatorial logic macrocell made up of a cascade of DCVSL gates, the generation of a completion signal consists of OR ing the outputs of only the last gate in the chain. This is an important point about processor design using self-timed methods. Typically, partitioning is done at the pipeline stage boundaries and it is only at these boundaries that a completion signal is required. Therefore, the overhead in circuitry for generating DV is small.

For a multi-bit cell such as an ALU - often used in DSP datapaths, there is a completion signal available for each bit of the data word. Strictly speaking, the correct completion signal for the entire cell, assuming an n-bit wide data word, is obtained by feeding the n OR gate outputs from the last DCVSL gate into an n-input Muller C-element[2]. As will be seen in later chapters however, this is not usually necessary for several reasons. In the case where the logic delay through each bit slice of the cell is nominally the same and the completion time is data independent, it is often sufficient to use the completion signal for a single bit. This of course depends on the matching characteristics of the integrated circuit. Matching at the local level has been exploited both in analog and digital circuits since the inception of the IC. In a self-timed circuit it can reduce the overhead in area required to generate the completion signal. This seems like a viable approach but since it is process dependent, it must be evaluated by the designer before a decision is made.

In the case where the completion time of a cell is data dependent such as for a ripple carry adder, all bits must be examined for DV generation if the data dependency is to be monitored. Since self-timed circuits can take advantage of this dependency and achieve cycle times that are closer to the *average* delay of the elements, all bits are typically used for such data dependent elements. Again however, it is rarely necessary to use a n-bit C-element to generate DV because of the way DCVSL functions. The data dependent nature of the completion time is typically the result of one gate that cannot evaluate until another gate produces an output such as the ripple carry adder. During pre-charge though, there is no such dependency and all gates generating a DV signal will pre-charge in nominally the same amount of time. Therefore, rather than using an n-input C-element to generate a completion signal, an n-input AND gate is sufficient. This also depends on the matching characteristics of the IC but it is usually a low risk approach. The data dependent macrocells described in this report use a tree of 4-input NAND /NOR gates to generate a completion signal.

2.2 Charge-Sharing

Like domino logic, DCVSL gates can exhibit charge-sharing problems. Figure 3.3 showed the allowed input sequencing of a DCVSL gate for valid outputs. It is a "legal" condition to have all of the inputs to the DCVSL gate low during pre-charge. The only constraint on the inputs then is to have one wire of each input - either in_x or $inbar_x$ -

[2]A Muller C-element is defined as a logical element whose output traverses high only after *all* of its inputs are high and stays high until *all* of its inputs go low in which case it traverses low.

eventually make a single transition from low to high to yield the correct output. Actually, any DCVSL gate whose inputs are fed directly from the output of other DCVSL gates will have those inputs (both true and complement) low during pre-charge simply because during the pre-charge phase, all DCVSL outputs are low. When all of the inputs are low during pre-charge, only the top two nodes of the NMOS tree of the gate are guaranteed to be pre-charge high. After pre-charge, the charge on those two nodes can be shared with nodes internal to the NMOS tree as transistors in the tree are subsequently turned "on". While one side of the tree will eventually be discharged to ground, the other side of the tree must remain at a logical "1" to generate valid complementary outputs. Therefore, charge sharing will cause incorrect operation if the parasitic capacitance associated with the devices being turned "on" after pre-charge (usually dominated by the source and drain diffusion capacitance of the NMOS transistors of the tree) roughly equals the capacitance of the pre-charge node at the top of the NMOS tree. One circuit technique that is used to improve the charge sharing situation of DCVSL gates is shown in Figure 3.5. Weak p-channel devices are fed back around the output inverters to act as current sources when the outputs of the gate are low. [35, 41] After the precharge phase, the current sources continue to operate and if charge sharing causes a drop in the voltage at the top of the tree, the voltage will be restored eventually to V_{dd}. The p-channel devices directly affect the speed performance of the gate since the NMOS transistors that are supposed to discharge one side of the tree must compete with the current source. Therefore, the p-channel devices cannot be very large and they are limited in the speed which they can supply charge to a node in the tree that is depleted by charge sharing. They tend to enhance the *static* behavior of the gate since they ensure that the voltage on the high side of the NMOS tree returns to V_{dd} instead of dropping below the threshold value of the output inverter due to a combination of charge sharing and leakage on internal nodes. However, if there is enough parasitic capacitance on the internal nodes and the inputs are brought high in rapid succession (i.e., faster than the p-channels can restore the charge from each one), then a problem will still exist.

If all of the inputs are set up before the I signal goes high for evaluation no charge sharing will occur and therefore the weak p-channel devices should be eliminated in the interests of speed and area consumption of the gate. However, in strictly static applications, the p-channel devices must remain. In the case where the weak feedback transistors are insufficient to prevent charge sharing from being a problem, then it is

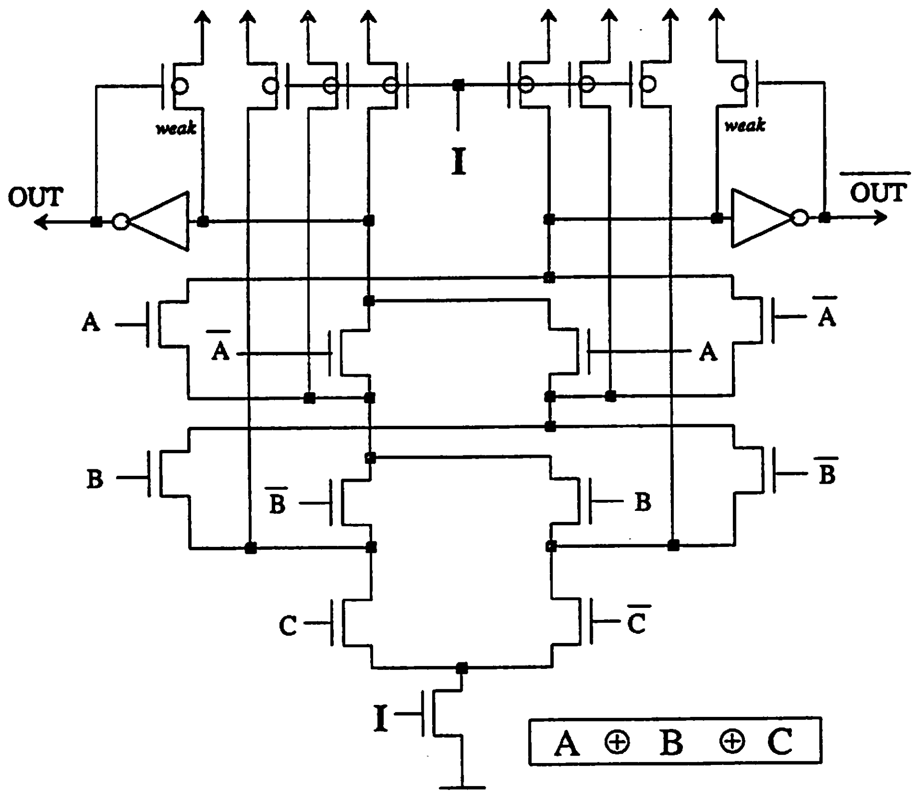


Figure 3.5: Alleviating charge sharing in a DCVSL gate.

necessary to pre-charge some or all of the internal nodes of the NMOS tree. The most straightforward way of pre-charging more of the NMOS tree internal nodes is by adding extra p-channel pre-charge devices connected between V_{dd} and those internal nodes. This technique, while ensuring that internal nodes are fully pre-charged, has accompanying disadvantages of 1) requiring more p-channel devices, 2) adding capacitive load to the I signal line and 3) slowing the gate switching speed by adding capacitance to the NMOS tree nodes and also requiring all internal nodes to discharge fully from V_{dd} to Ground. If the order of the input switching is know *a priori* then only nodes causing charge sharing problems should be pre-charged. Another method for eliminating charge sharing problems involves delaying the fall time of the DCVSL outputs at the start of pre-charge. By doing so, the input devices that they in feed will in turn stay "on" longer allowing more charge to enter the tree. [42] The more the internal nodes are charged before evaluation, the less the output nodes will be lowered in voltage during evaluation. Another method for alleviating charge sharing problems is to use NMOS devices in the tree to charge internal nodes to $V_{dd} - V_t$. [43, 45] This method reduces the routing required for the pre-charge devices and since it does not charge the internal nodes fully to V_{dd}, the gate switching time is faster.

2.3 Design and Layout Issues

The NMOS device tree

Readers familiar with the design of static CMOS logic may have noticed from the description of DCVSL that the design of a typical gate is not as straightforward. The first task involves mapping a desired logic function into the actual schematic for the NMOS device tree. A systematic means for this is necessary and it must include the simplifications possible by sharing the transistors between the two sides of the tree in order to yield the smallest area for the gate. There are several methods for systematically generating the NMOS tree for any arbitrary DCVSL gate. The chip designs described in later chapters were done using a new 'c'-language program written expressly for this purpose. Some other simpler hand-design methods will also be described that can be used for gates with a small number of inputs.

Ntree is a DCVSL design program based upon representing the Boolean function in terms of a directed acyclic graph which is then manipulated to reduce it to a canonical form for the original function. The branches of the graph directly represent the branches of the NMOS tree (and therefore the n-channel transistors) for the DCVSL gate being designed. Routines for manipulating the graph were developed by R. E. Bryant and full details of them can be found in [51]. Historically, many digital system designs have been expressed as a sequence of operations on Boolean functions. Algorithms to efficiently manipulate these Boolean functions symbolically are very useful, however some of the common requirements of the algorithms, such as testing for *satisfiability*[3] or *equivalence*[4] have NP-complete solutions which need computer time that grows exponentially with the size of the problem. The computation problem occurs when Boolean functions are represented symbolically using some of the more classical techniques such as Karnaugh maps or canonical sum-of-products forms. These representations are of size 2^n for every function of n arguments. More important is that none of the classical representations are *canonical forms*. In other words, a given function may have many different representations[51].

Bryant represents Boolean functions as directed acyclic graphs that resemble binary decision diagrams [52]. By placing further restrictions on the ordering of decision variables in the vertices, algorithms for manipulating the representations in a more efficient manner could be developed. All symmetric functions can be represented by graphs where the number of vertices grows at most as the square of the number of arguments. Also, after the graphs are *reduced*, the representation is a canonical form. The disadvantage of using this type of graphical representation is that the ordering of the inputs to the Boolean function can dramatically affect the size of the graphical representation. In fact, the problem of computing an ordering that minimizes the size of the graph is itself an NP-complete problem. Bryant suggests that a small set of heuristics can be used to solve this problem with satisfactory results.

To describe the graphical representation used for Boolean functions requires some definitions. First, for the graph itself:

Definition 3.1 *A function graph is a rooted, directed graph with vertex set V containing two types of vertices. A nonterminal vertex v has as attributes an argument index $index(v) \in \{1, \ldots, n\}$ and two children $low(v)$, $high(v) \in V$. A terminal vertex v has as attribute a value $value(v) \in \{0, 1\}$.*

[3]A Boolean function is satisfiable if there exists an assignment of input variables which causes the function to evaluate to 1.

[4]Two Boolean expressions are equivalent if they denote the same function.

Furthermore, for any non-terminal vertex v, if $low(v)$ is also nonterminal, then it must be that $index(v) < index(low(v))$. Similarly, if $high(v)$ is nonterminal, then it must be that $index(v) < index(high(v))$. Function graphs form a proper subset of conventional binary decision diagrams. The ordering restriction also implies that a function graph is acyclic because the nonterminal vertices along any path have strictly increasing index values.

Definition 3.2 *A function graph G having root vertex v denotes a function f_v defined recursively as*

1. *If v is a terminal vertex:*
   (a) *If $value(v) = 1$, then $f_v = 1$.*
   (b) *If $value(v) = 0$, then $f_v = 0$.*
2. *If v is a nonterminal vertex with $index(v) = i$, then f_v is the function*

$$\begin{aligned} f_v(x_1, \ldots, x_n) &= \bar{x}_i \cdot f_{low(v)}(x_1, \ldots, x_n) \\ &\quad + x_1 \cdot f_{high(v)}(x_1, \ldots, x_n) \end{aligned} \tag{3.1}$$

Another way of saying this is that a set of argument values $x_i, \ldots, x_n$ describes a path in the graph starting from the root where, if some vertex v along the path has $index(v) = i$, then the path continues to the low child if $x_i = 0$ and to the high child if $x_i = 1$. The value of the function of these arguments equals the value of the terminal index at the end of the path. In the physical NMOS tree that is the analog of the graph, the root is **GROUND** and the terminal vertices **0** and **1** correspond to the nodes *fbar* and *f* in Figure 3.2.

Definition 3.3 *Function graphs G and G' are isomorphic if there exists a one-to-one function ϕ from the vertices of G onto the vertices of G' such that for any vertex v, if $\phi(v) = v'$, then either both v and v' are terminal vertices with $value(v) = value(v')$, or both v and v' are nonterminal vertices with $index(v) = index(v')$, $\phi(low(v)) = low(v')$, and $\phi(high(v)) = high(v')$.*

Definition 3.4 *For any vertex v in a function graph G, the subgraph rooted by v is defined as the graph consisting of v and all of its descendants.*

A function graph can be reduced in size without changing the Boolean function it represents by eliminating redundant vertices and duplicate subgraphs. The reduced graph is the goal of the DCVSL NMOS tree design program.

Definition 3.5 *A function graph G is reduced if it contains no vertex v with $low(v) = high(v)$, nor does it contain distinct vertices v and v' such that the subgraphs rooted by v and v' are isomorphic.*

The following theorem is important in the use of this method for designing DCVSL trees:

Theorem 3.1 *For any Boolean function f, there is a unique (up to isomorphism) reduced graph denoting f and any other function graph denoting f contains more vertices.*

The proof of this theorem can be found in [51]. So, by defining the graphical representation of an arbitrary Boolean function f as above, one can make use of efficient computer routines for building and reducing the graph. The reduced graph is the smallest graph that represents the function f, hence the corresponding DCVSL circuit implementing f will contain the minimum number of devices.

Program Operation

The main program flow of *ntree* is shown below in pseudo-code:

```
main() {
        Read input function from file;
        Construct input parse tree;
        for (each input ordering){
            Build reduced graph;
            if(graph smaller)
                Save
            else
                Discard
        }
        Write output spice file;
}
```

The input format is LISP-like using parentheses to delineate primitive gate functions. The program supports all of the primitives used to describe logic functions in the set $F \in$ {AND,OR,NOT,NAND,NOR,XOR,XNOR}. As an example, the input file below is shown:

```
# Example DCVSL gate description:
#    4-inputs
(example gate 1 (nand (or 1 2) 3 4))
#
#
```

The input parse tree for the example gate is shown in Figure 3.6. Since it is a binary tree, the gate function for multiple input gates must be changed to avoid multiple inversions (using associativity of the logic function). That is why the lower right node of the graph is an AND gate rather than a NAND gate.

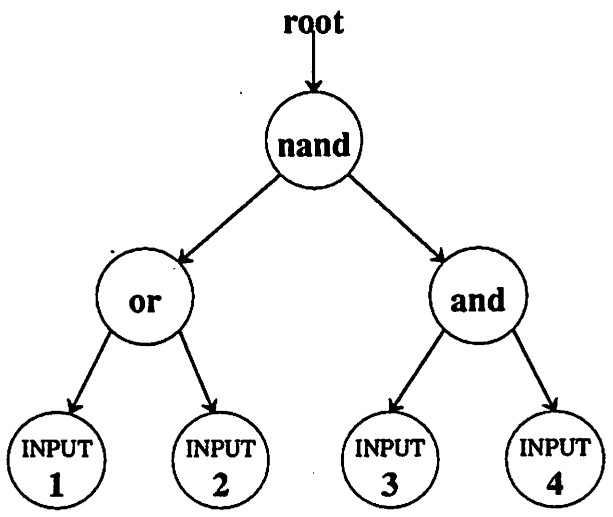


Figure 3.6: Input parse tree for example gate.

To build up the function graph a data structure is used for each vertex in the graph. It has the following form:

```
typedef struct vertex {
        int index;                    /* index from 1 to n+1 */
        int value;                    /* -1,0,1 (-1 for non-terminal)*/
        int id;                       /* identification number */
        unsigned int mark;            /* Boolean marker */
        struct vertex *low;           /* pointer to low child */
        struct vertex *high;          /* pointer to high child */
} VERTEX;
```

The structure contains all of the necessary information about each vertex including a unique "id" number, an index entry, the value - showing 0 or 1 for terminal vertices and -1 for non-terminal vertices, and a Boolean marker which is useful for indicating whether a vertex has been visited when traversing a function graph. The pointers contain the address of the low and high children vertices in the graph. A general function for traversing the graph by making use of the marker entry is shown below. Each time the graph is traversed, the markers are set to all 0 or all 1. A version of this traversing function is used in several parts of the algorithms.

```
Traverse(v) {                    /* vertex v = root of graph */
    v.mark = not(v.mark);
    ... do something to v...
    if(v.index ≤ n) {            /* v is non-terminal */
        if(v.mark ≠ v.low.mark) Traverse(v.low);
        if(v.mark ≠ v.high.mark) Traverse(v.high);
    }
}
```

To build up a function graph from the Boolean expression, the function *Apply* is used. It takes graphs representing functions f_1 and f_2, a Boolean operator and produces a reduced graph representing the function $f_1 < op > f_2$ defined as

$$[f_1 < op > f_2](x_1, \ldots, x_n) = f_1(x_1, \ldots, x_n) < op > f_2(x_1, \ldots, x_n) \tag{3.2}$$

The operation of the algorithm is based on the following recursion:

$$[f_1 < op > f_2] = \bar{x}_i \cdot (f_1 |_{x_i=0} < op > f_2 |_{x_i=0}) + x_i \cdot (f_1 |_{x_i=1} < op > f_2 |_{x_i=1}) \tag{3.3}$$

To apply the operator to functions represented by graphs with roots v_1 and v_2, there are several cases to consider. If both v_1 and v_2 are terminal vertices, then the result graph has a terminal vertex having a value equal to $value(v_1) < op > value(v_2)$. Otherwise, at least one of the two vertices are nonterminal. If their indices are both equal to i, then a new vertex u is created having index i. The algorithm is applied recursively on $low(v_1)$ and $low(v_2)$ to generate the subgraph whose root becomes $low(u)$, and on $high(v_1)$ and $high(v_2)$ to generate the subgraph whose root becomes $high(u)$. If, on the other hand, $index(v_1) = i$ but either v_2 is a terminal vertex or $index(v_2) > i$, then the function represented by the graph with root v_2 is independent of x_i or $f_2 |_{x_i=0} = f_2 |_{x_i=1} = f_2$. So, a vertex u with index i is created and the algorithm is applied recursively on $low(v_1)$ and v_2 to generate the subgraph whose root becomes $low(u)$, and on $high(v_1)$ and v_2 to generate the subgraph whose root becomes $high(u)$. The graph produced by the algorithm is not in general reduced, so the function *Reduce* is applied to it before it is returned.

```
Apply(v1,v2,op) {              /* v1,v2 vertices */
      Initialize Table to NULL;
      u = Apply_step(v1,v2);
      return((Reduce(u));
}

/* Recursive function to implement Apply */
Apply_step(v1,v2) {
      u = Table[v1.id,v2.id];
      if(u == NULL) return(u);
      u = new vertex; u.mark = false;
      Table[v1.id,v2.id] = u;
      u.value = v1.value op v2.value;
      if(u.value != dontcare) /* create terminal vertex */
            u.index = depth+1; u.low = NULL u.high = NULL;
      else {                  /* create nonterminal and eval further down */
            u.index = min(v1.index,v2.index);
            if(v1.index == u.index)
                  vlow1 = v1.low; vhigh1 = v1.high;
            else
                  vlow1 = v1; vhigh1 = v1;
            if(v2.index == u.index)
                  vlow2 = v2.low; vhigh2 = v2.high;
            else
                  vlow2 = v2; vhigh2 = v2;
            u.low = Apply_step(vlow1,vlow2);
            u.high = Apply_step(vhigh1,vhigh2);
      }
      return(u);
};
```

The actual function implementation for *Apply* above contains some enhancements to reduce the computation time of its application. A table is maintained containing entries of the form (v_1, v_2, u) indicating that the result of applying the algorithm to subgraphs with roots v_1 and v_2 was a subgraph with root u. Before applying the algorithm to a pair of vertices, the table is checked to see if it contains an entry for the pair. If so, they need not be evaluated again and the result u is simply returned. If the function is called with one of the vertices being a terminal vertex and it is a "controlling" value for the operator, such as a 1 which controls an OR function - always returning a 1, then the appropriate terminal vertex is just returned.

The algorithm for *Reduce* works as follows: Proceeding from the terminal vertices up to the root, a unique identifier number is assigned to each unique subgraph root. In other words, each vertex v is assigned a label $id(v)$ such that for two vertices v and u, $id(v) = id(u)$ if and only if $f_v = f_u$ in the terminology of Definition 3.2. After the labeling is completed, the algorithm then constructs a graph with only one vertex for each unique label. By following the rules listed below, the correct labeling is ensured. Remember that for any index i, vertices with an index greater than i have been labeled since the algorithm starts at the terminal vertices (index = $n+1$).

1. Any two terminal vertices are assigned the same label as long as they have the same value ($\in \{0,1\}$).
2. If $id(low(v)) = id(high(v))$, then vertex v is redundant, and $id(v)$ is set to $id(low(v))$.
3. If there is some labeled vertex u with $index(u) = i$ having $id(low(u)) = id(low(v))$, and $id(high(u)) = id(high(v))$, then the reduced subgraphs rooted by these two vertices will be isomorphic and $id(v)$ is set to $id(u)$).

The pseudo-code for *Reduce* is shown next:

```
Reduce(v) {                                   /* v is a vertex */
     VERTEX subgraph[G];
     LIST Q, vlist[n+1];
     Put each vertex v on vlist[v.index];
     nextid = 0;
     for(i = n+1 down to 1) {
          Q = empty set;
          for( each u in vlist[i]) {
               if(u.index = n+1)
                    Add key to Q where key = u.value; /* terminal */
               else if(u.low.id == u.high.id)
                    u.id = u.low.id;   /* redundant vertex */
               else
                    Add key to Q where key = (u.low.id,u.high.id);
          };
          Sort Elements of Q by keys;
          oldkey = (-1,-1)              /* unmatchable key */
          for each key in Q removed in order {
               if(key = oldkey)
                    u.id = nextid;      /* matches existing vertex */
               else {                   /* unique vertex */
                    nextid = nextid+1; u.id = nextid; subgraph[nextid]=u;
                    u.low = subgraph[u.low.id]; u.high = subgraph[u.high.id];
                    oldkey = key;
               };
          };
     };
     return(subgraph[v.id]);
};
```

The vertices are first collected in lists according to their indexes. The function *Traverse* can be used for to do the collection. The lists are processed starting from the terminal vertices and proceeding up to the root. For each vertex processed, a key is created that is either the *value* for a terminal vertex or the pair $< id(low(v)), id(high(v)) >$ for nonterminal vertices. If the vertex has $id(low(v)) = id(high(v))$, then $id(v)$ is immediately set to $id(low(v))$. The remaining vertices are sorted according to their keys and to perform the reduction, a given label is assigned to all vertices having the same key. For each unique label, a single vertex is selected and a pointer to it is stored so that the reduced version can be built.

As shown in the pseudo-code for the main program operation, each input ordering is tried when building function graphs. This is not very efficient in terms of computation time since for an n-input gate design, $n!$ graphs must be built. This places a practical limitation on n to be $n \leq 8$ for most workstations, however for the gate designs described in later chapters, the limitation was not severe. In fact, limiting the number of gate inputs also limits the worst case number of series transistors in the NMOS tree design, which is commonly done in the interests of speed performance. The program operation could easily be enhanced to allow a greater number of inputs by the addition of some heuristics for determining the input orderings to try. In *ntree*, each function graph is built and then saved only if it is smaller than the previous graph (where smaller means having less vertices). There is one tradeoff that is made in determining the final graph which is selected for the tree design. A record is kept of both G, the number of vertices in a graph and L, the number of series connected devices which can be determined by looking for the longest path between Ground and a terminal vertex. G_{min} is the number of vertices of the smallest graph and L_{min} is the number of series devices in the graph with the minimum number of series devices. For the graph which has size G_{min}, the number of series devices $L_{G_{min}}$ is also saved. Similarly, for the graph with L_{min}, its size is saved in $G_{L_{min}}$. A cost is computed for each of the two graphs as follows:

$$COST_{G_{min}} = G_{min} + L_{G_{min}} \tag{3.4}$$

$$COST_{L_{min}} = L_{min} + G_{L_{min}} \tag{3.5}$$

In the interests of speed, a larger graph will be chosen if it has less series connected devices. In other words,

$$If(COST_{L_{min}} < COST_{G_{min}}),\ choose\, Graph_{L_{min}} \tag{3.6}$$

The size differences between a minimum size graph and one that contains the smallest number of series devices is typically only one or two vertices. Therefore, it was felt that this tradeoff was worthwhile.

Besides heuristics that could be used to try less input orderings, there is another area that could be addressed in future versions of the *ntree*. Given that there are multiple graphs which have the same number of vertices, heuristics could be added to somehow determine the best graph to choose for layout considerations. The location in the tree for example of a device receiving a certain input signal can be important in the overall layout of the circuit. More information is needed about the rest of the system however, before choices of this nature can be made. Some CVSL design software that addresses the issue of wirablilty has been described in the literature[54, 55, 56].

The program textual output for this example is shown below. The binary input parse tree is printed along with the final input ordering and function graph. Indentation is used to try to clarify the levels in the trees in the printout. Because most of the routines are recursive in nature, there was some problems initially in memory management. Memory statistics are printed out as a tool to check for problems of this sort.

```
CVSL Logic Minimization Program by Gordon Jacobs
Rev 1.1

Date:  Sun Nov 13 15:00:03 1988


********* INPUT LISTING ************  Input File:  exmpl1 ****

(example gate 1 (nand (or 1 2) 3 4))
#
#

Gate Name: example gate 1
Number of unique inputs (depth) = 4
**********************************************************


---------- INPUT PARSE TREE : ----------------

nand  0
 with left side --
|   or  0
|    with left side --
|    |   INPUT  1
|    with right side --
|    |   INPUT  2
 with right side --
|   and  0
|    with left side --
|    |   INPUT  3
|    with right side --
|    |   INPUT  4

----------------------------------------------------
       INPUTS: 1 2 3 4
For indices:   1 2 3 4   G = 6   L = 4

G => number of vertices.   L => number of series devices.
Gmin = 6/L = 4.  Lmin = 4/G = 6
```

```
--------------- FUNCTION GRAPH: ---------------------

-> INPUT ORDERING: 1 2 3 4

Vertex: INPUT 1  index = 1  id = 1
`with low(0) side --
|   Vertex: INPUT 2  index = 2  id = 2
|   `with low(0) side --
|   |   ***** ONE ******
|   `with high(1) side --
|   |   Vertex: INPUT 3  index = 3  id = 4
|   |   `with low(0) side --
|   |   |   ***** ONE ******
|   |   `with high(1) side --
|   |   |   Vertex: INPUT 4  index = 4  id = 5
|   |   |   `with low(0) side --
|   |   |   |   ***** ONE ******
|   |   |   `with high(1) side --
|   |   |   |   ***** ZERO *****
`with high(1) side --
|   Vertex: INPUT 3  index = 3  id = 4
|   `with low(0) side --
|   |   ***** ONE ******
|   `with high(1) side --
|   |   Vertex: INPUT 4  index = 4  id = 5
|   |   `with low(0) side --
|   |   |   ***** ONE ******
|   |   `with high(1) side --
|   |   |   ***** ZERO *****

Memory used:
 Vertices used = 14
 Lists used = 8
 Trees used = 9

*** end ***
```

A graphical representation of the function graph is shown in Figure 3.7.

For lack of a better circuit description, the corresponding spice file written by *ntree* is:

```
         NMOS Tree for (example gate 1 )
*
* Logic Expression: (nand (or 1 2) 3 4)
*
* This file generated by ntree on Sun Nov 13 15:00:03 1988
*
*  NODE ASSIGNMENTS:
*  GND  = 0      Vdd  =  100
*  Pbulk = 102   Nbulk = 101
*  (Complement of )Input Number 1  is node (11) 1
*  (Complement of )Input Number 2  is node (12) 2
*  (Complement of )Input Number 3  is node (13) 3
*  (Complement of )Input Number 4  is node (14) 4
*  F_OUT = 21   F_BAR_OUT = 20
*
*     D   G   S   B
m1   24  11   0  101  NMOS
m2   21  12  24  101  NMOS
m3   26   2  24  101  NMOS
m4   21  13  26  101  NMOS
m5   27   3  26  101  NMOS
m6   21  14  27  101  NMOS
m7   20   4  27  101  NMOS
m8   26   1   0  101  NMOS
***end***
```

The schematic drawing for the NMOS tree in the spice file is shown in Figure 3.8.

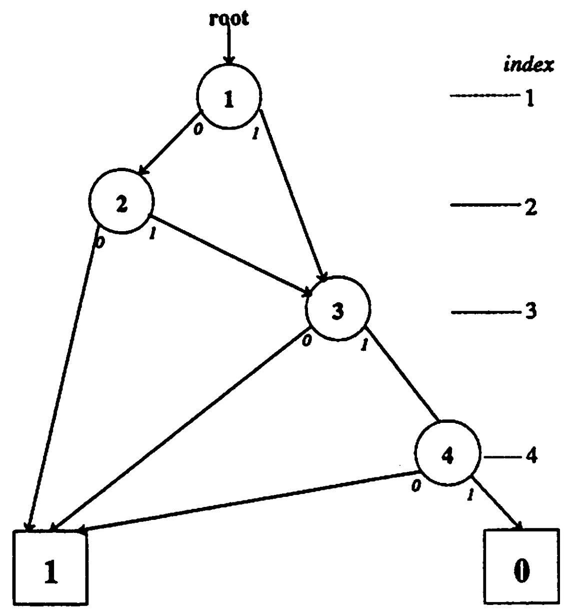


Figure 3.7: Function graph for the example gate.

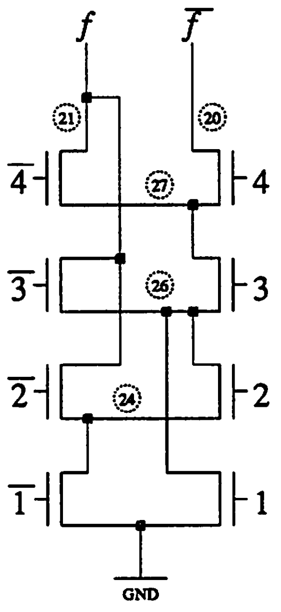


Figure 3.8: NMOS tree for the example gate.

More information about *ntree* can be found in Appendix A.

Layout style

Once the schematic for the gate is determined, the layout must be performed in a manner which helps to minimize the chip area and maximizes speed by keeping parasitic capacitances low. The custom layouts employed in most of the gates described in later chapters followed the basic style as illustrated in Figure 3.8. A stack of NMOS differential pairs is placed first. Diffusion is usually required to make connections between device pairs in a reasonable area, however it should be minimized for speed considerations. Vertical running $metal_2$ was used for data inputs and connections between device pairs. Horizontal $metal_1$ was used for bussing control inputs and power. This style met with limited success in being competitive with static gates in terms of overall area, however the author claims no great talents with regards to layout ability.

More common among published circuits is a "sea-of-gates" style layout that is comparable with an automatic tool for wiring the DCVSL gates[53, 54, 55, 56]. These techniques have shown favorable comparisons with standard gate designs in terms of area. Additionally the adaptability to automatic tools for layout make this approach advantageous.

3. Alternatives to DCVSL

Since the introduction of DCVSL, there have been several similar logic structures described in the literature. Most of these address the issues of speed or area efficiency of DCVSL and offer improvements while maintaining basically the same functional operation. As stated above, any logic family which can provide completion information is suitable for self-timed circuit design [5]. A brief survey of the recent developments in this sort of logic design is presented here. It should also be mentioned that a large effort in the development of *single-ended* Cascode Voltage Switch Logic, or just plain CVSL, has taken place. [43, 38] While this can compare more favorably to static CMOS logic in terms of area it does not provide means for generating an adequate completion signal.

The circuits described in this document all use the basic DCVSL architecture. While performance was of concern in the DSP chips designed, DCVSL presented a slightly

[5]This almost certainly will be dependent on having complementary outputs available.

more conservative approach in terms of design which was chosen in order to expediently demonstrate self-timed circuits.

3.1 Sample-Set Differential Logic (SSDL)

SSDL logic addresses the speed issues of DCVSL gates as they become large. While an arbitrary logic function can be implemented in an NMOS tree, the size of the tree and the number of series connected devices will grow when the number of inputs or complexity of the logic function increases. DCVSL literature notes that the delay is relatively constant despite the logic function of the gate, a big improvement over normal gate design. However, more series connected devices in the NMOS tree will cause a slowdown of the discharge action of the tree and hence a slower switching time.

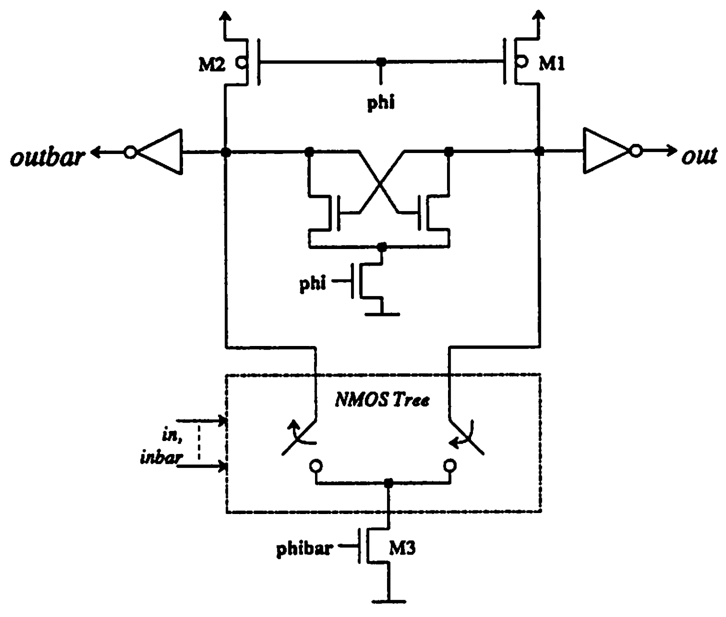


Figure 3.9: A generalized SSDL logic gate.

SSDL adds a sense amplifier to the basic DCVSL structure as shown in Figure 3.9. The key to using this sense amplifier is changing the timing of the operation of the gate. In what would normally be the pre-charge time of the gate, the inputs are assumed to be valid and devices $M1 - M3$ are all "on". Since there is a path from one of the output nodes of the tree to ground, that node will be at a voltage less than V_{dd}. This is called the *sample* phase of the operation. When *phi* switches, the *set* phase of operation begins and the sense amplifier is activated. It detects which side of the NMOS tree was being discharges and switches rapidly. The advantage to this approach is that the switching time is independent of the complexity of the NMOS tree.

Since both outputs are still low during *sample* and they become complementary during *set*, the generation of a completion signal is identical to DCVSL. The interface between consecutive SSDL gates however requires different control circuitry from DCVSL to work properly. Note that the inputs are required to be valid during the time when the outputs of the gate are both low (sample phase). This precludes feeding the outputs to another SSDL gate with the same clock signal. One SSDL gate *can* drive another directly if the clock to the next gate is inverted with respect to the first gate. This still poses a problem in a self-timed circuit as will be seen in the next chapter. One could envisage generating a completion signal at the output of each gate in a cascade of SSDL gates and using this to generate the required clock for the next stage. The added delay of doing this might unfortunately cancel the benefits of using SSDL instead of a cascade of DCVSL gates.

On the other hand, since SSDL lends itself to very complicated logic functions, it would be entirely appropriate to any single stage self-timed macrocell that implements a complex logic function. If the circuit can be partitioned this way, then the benefits of using SSDL over DCVSL could be obtained in the form of higher performance. One might replace a cascade of DCVSL gates with a single SSDL gate taking advantage of the fast switching time that is independent of the NMOS tree size. Therefore, in some cases SSDL may represent a real improvement.

The speed gains of SSDL come at the expense of adding the sense amplifier. The disadvantage of this is that the sense amplifier design will tend to be more process dependent. Also, the power consumption of a SSDL gate is higher than that of a DCVSL gate because during the sample phase, when the inputs become valid, there is a path between V_{dd} and Ground through the NMOS tree.

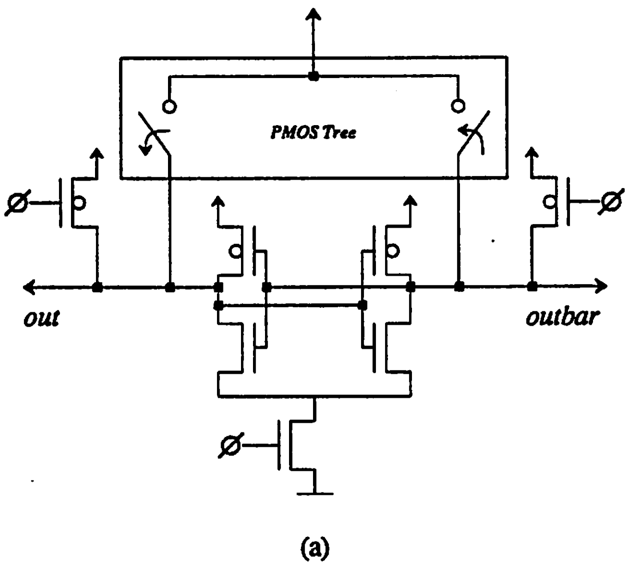
3.2 Enabled/disabled CMOS Differential Logic (EDCL)

A variation on SSDL logic as described above was published recently and it attempts to eliminate the shortcomings of the SSDL style logic. EDCL works on a similar principle to SSDL by using a bi-stable sense amplifier circuit for rapid switching times [47].

Figure 3.10 shows the basic EDCL gate in both n-type and p-type configurations. The circuit operation will be described in terms of the NMOS version in Figure 3.10a. Unlike SSDL, the NMOS tree is not conducting during pre-charge so the power consumption is reduced to that of a DCVSL style logic gate. Also, the output inverters were removed in order to save area and increase speed. Therefore, when the clock *phi* is high, which is the so-called pre-charge period, both outputs are shorted to ground by M1-M2 and the NMOS tree and sense amplifier (M3 off) are disabled. On the falling edge of *phi*, the sense amp is enabled and if the inputs are valid, then one side of the sense amp/bi-stable output will be held lower than the other causing the appropriate switching of the outputs to the correct complementary state.

The EDCL gate has the same advantage of taking the same switching time for any complexity NMOS tree as does an SSDL gate. It also is fully static since once the bi-stable element switches, it remains in the same state as long as it is enabled. The gate design is however more dependent on its circuit connection with other gates since the output nodes are not buffered. The size of the sense amp devices can be increased for increased output drive however, this also adds more capacitance to the output nodes and can affect the speed of the gate. The generation of a completion signal is identical to DCVSL gates.

As with SSDL, the connection of several EDCL gates to form a more complicated logic function or sequential function is not as straightforward as with DCVSL gates. The switching of an EDCL gate occurs at the clock edge and the inputs must be valid before this edge. The inventor of EDCL [47] suggests a method for connecting a cascade of the gates by generating a "done" or completion signal for each gate as suggested in the last section. This forces the operation of the cascaded gates to be sequential as they would be in DCVSL or Domino logic. The proposed method involves connecting an inverter to the drain of M3 in Figure 3.10a which detects when the sense amplifier is active. This is not a fully reliable method because it assumes that the switching time of the sense amplifier will match that of the added inverter under all conditions. A lower risk method would involve using the actual gate complementary outputs as described for DCVSL. EDCL does provide a method for implementing self-timed circuits without the power consumption of an SSDL gate. Therefore, it is feasible for certain applications, especially where a complex NMOS tree is required to generate a certain logic function.



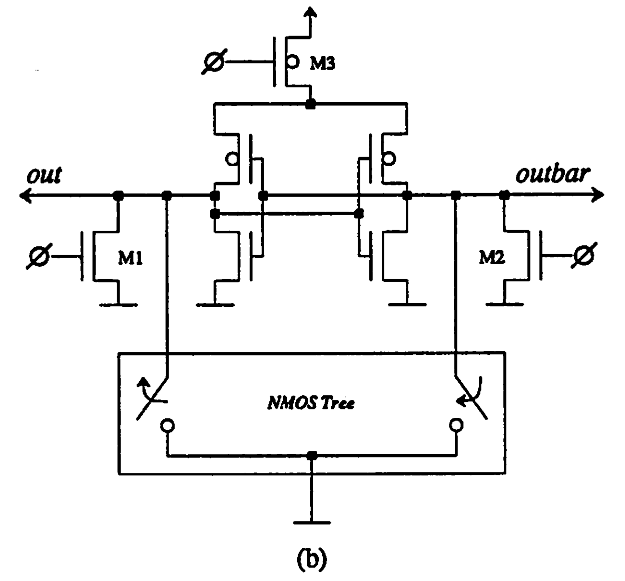


Figure 3.10: A generalized EDCL logic gate.

3.3 Latched Domino CMOS Logic (Ldomino)

Another interesting variation on DCVSL logic is a kind of hybrid between standard domino logic and DCVSL logic. It is called Ldomino logic and while generating a complementary output, it only requires a single ended NMOS tree to implement the logic function [46]. While Ldomino logic was proposed to circumvent the problem of a lack of inversion in a standard domino gate, the existence of a complementary output makes it a candidate for making self-timed circuits. Additionally, Ldomino logic gates can possibly serve as an interface between single ended and differential logic families which could significantly reduce the area required for a complex piece of logic requiring a completion signal output.

The circuit diagram for a generic Ldomino gate is shown in Figure 3.11. Consisting of a standard domino gate and an unbalanced sense amplifier/bi-stable element, it generates complementary outputs for the function implemented by the domino portion. The sense amplifier is formed by devices M1,M2,M4, and M5. The two sides of the sense amplifier are unbalanced by the larger capacitance present on the output node connected to the domino NMOS tree. Device M4 can also be made wider to add to the imbalance. After pre-charging, the clock is raised which enables the sense amplifier and NMOS tree of the standard domino gate. If the NMOS tree provides a path to ground, then the drain of M5 will be pulled down and the sense amp will switch such that that *out* will go high and *outbar* will stay low. If however, there is no path from the drain of M5 to ground, then the sense amp will switch into the other state where *out* is low and *outbar* is high. The imbalance in the sense amplifier must cause the gate to reliably switch when the NMOS tree does not provide a path to ground at the drain of M5. With a few more transistors than standard domino logic, complementary outputs are available.

The speed of Ldomino gates can be made higher than DCVSL and the area is less than that required by DCVSL. Designing a Ldomino gate does however require more care since there is a direct tradeoff between speed and noise margin when sizing the sense amplifier transistors. An important limitation not mentioned in the literature [46], is the loss of the "domino" action of a gate. In the absence of any valid inputs (i.e. all devices in the NMOS tree disabled), the Ldomino gate will fire due to the imbalance of the sense amplifier. This action precludes cascading several Ldomino gates to form a complicated section of combinatorial logic. The inputs must be valid before the clocking signal rises

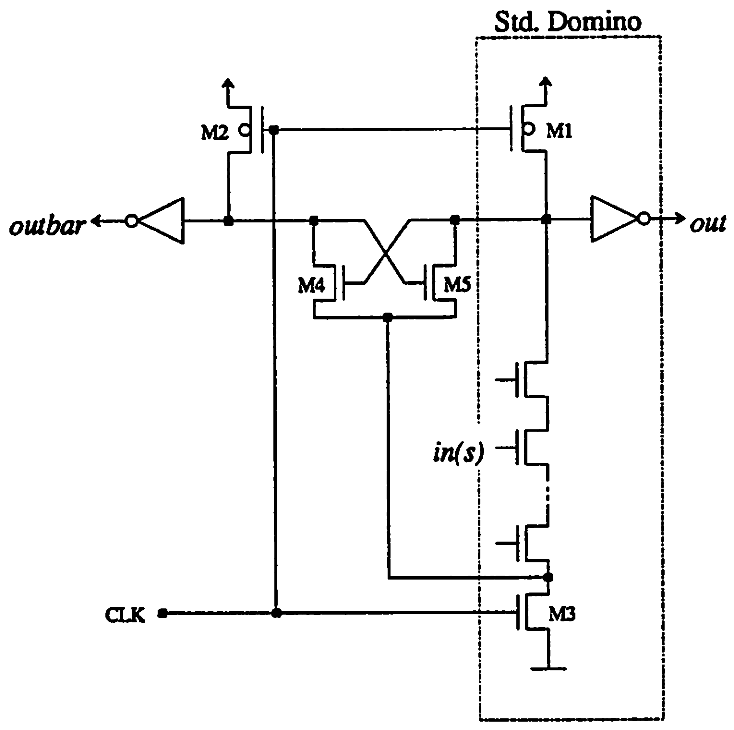


Figure 3.11: A generalized Latched Domino logic gate.

so that the NMOS tree can discharge its side of the sense amplifier before the imbalance discharges the other side of the amplifier. Therefore, in a series of Ldomino gates, all gates would switch at approximately the same instant after the rising edge of the clock instead of waiting for the results of the previous gate as in standard domino logic. One would like to be able to use single ended logic for area and speed benefits at all stages preceding the one where completion information is required (the output or last stage). The only way this could be done would be to somehow delay the clock of the Ldomino stage until the inputs were valid. Ldomino would be appropriate in a self-timed block in which only a single gate is necessary.

4. Summary

The first basic requirement of a self-timed circuit is that it generate completion information when its outputs are valid. This requirement can be met by using a logic family called DCVSL which generates both an output signal and its complement for any logic function. By simply ORing together these complementary outputs, a reliable completion signal is generated. During the pre-charge phase of operation of a DCVSL gate, both outputs become low which ensures that the completion signal fully cycles for each distinct operation.

The NMOS tree of a DCVSL gate can be designed in such a way that the two sides of the tree "share" transistors which makes the gate design efficiency increase as the logic function becomes more complicated. Automation of the NMOS tree design for arbitrary logic functions has been demonstrated. Since differential logic signal must be routed between DCVSL gates, the layout is more challenging than a single ended logic family. A regular layout style allows automation of the gate layout although hand packing was employed in the designs described in later chapters.

There are several variations on the DCVSL principle that have been introduced to offer increased speed, smaller layout area, and maintain complementary outputs to simplify logic over standard domino gate designs. These variations include SSDL, EDCL, and Ldomino logic. The timing of these alternative logic families cause added complexity for self-timed applications in some cases but almost all of the alternatives will work for self-timed stages where a single complex logic gate will suffice. Thus, enhanced performance is possible over using DCVSL at the expense of more sensitivity to design parameters and the loss of generality in where the gates can be placed in a circuit.

Chapter 4

Handshaking Circuit Synthesis

In the previous chapter, we examined the physical realization of a self-timed logic family suited for integrated circuits. The completion information provided by a self-timed logic block is one of two essential ingredients for composing a self-timed *system*, the other ingredient being the logic which makes use of the information to manage the transfer of data between stages. This chapter studies the synthesis and design of reliable *handshake* circuits which handle the interstage communication. The term "handshake" describes the local nature of the communication. In a synchronous system, all operations ideally happen at precisely the same moments in time, synchronized by the system clock, just the same as a school bell signals the class periods to all students at once. In an asynchronous system, adjacent stages negotiate the transfers of data between them independent of what is happening in other parts of the system. This is more like going from booth to booth at an exhibition, where the spent at each booth depends only on you and the people in the booth and not what is happening at other locations. Each transfer of data in a asynchronous system follows a handshaking sequence which ensures that no loss of data occurs. This consists of conversation between stages in which handshake signals are raised and lowered to do the signalling.

Handshake signals are typically labeled *Request* and *Acknowledge* signals in the literature. As is probably obvious, the request line usually signals that one stage is ready to initiate a transfer while the acknowledge line signals that the transfer is complete. The exact sequence which defines a transfer can vary between systems and it is called the handshaking *protocol.* In the circuits described in later chapters, a *4-cycle protocol* is employed. The sequence of handshake signals for this protocol is given below.

The reliable synthesis of handshaking circuits has been one of the major challenges of designing an asynchronous system. Given a sequence of operations, one must synthesize a circuit which will follow the sequence but also take a minimum amount of overhead time to do so as well as avoiding becoming deadlocked or causing some other error under any conditions. Additionally, the synthesis of such handshaking circuits should be relatively simple, perhaps using a higher level language description, so that the design process is not impeded. The design methodology described in this chapter attempts to meet these goals and the results obtained have shown great promise.

1. Partitioning

Since handshake circuits oversee the transfer of data between stages, the partitioning of the system into stages is the first task required in specifying a self-timed system. No automatic way for doing this is being presented here. Rather, a few guidelines that are related to physical constraints are discussed. The time required to complete a handshaking "conversation" for each data transfer represents an overhead associated with the self-timed approach which is undesirable. Therefore, while the methods described in the Chapter 3 allow for self-timing all the way down to the individual gate level, this would most likely be impractical for any large system. Another reason to avoid partitioning at the gate level is the hardware overhead to generate a completion signal. While a data valid signal can be generated by a single OR gate, the OR gate would be a prohibitive excess of hardware if it were necessary on *every* gate in the system (100% overhead). Therefore, it makes sense to partition the system into self-timed blocks as illustrated in Figure 4.1. The DCVSL logic family is ideal for larger self-timed blocks because a collection of DCVSL gates can be cascaded directly to form a larger combinatorial block. The completion signal generation is only necessary on the last stage of the cascade.

Another way to look at the partitioning is that the handshake signals of a stage in effect make up the local "clocks" of that stage. Clock distribution is typically not troublesome on a local level. Therefore, one should envisage making the size of each self-timed block large enough to minimize the completion circuit overhead and small enough to avoid timing signal distribution difficulties. A logical choice for partitioning a DSP cell based chip is at the macrocell level. A handshake stage often (although not strictly) can be considered a pipeline stage so the partitioning might be done at the pipeline boundaries

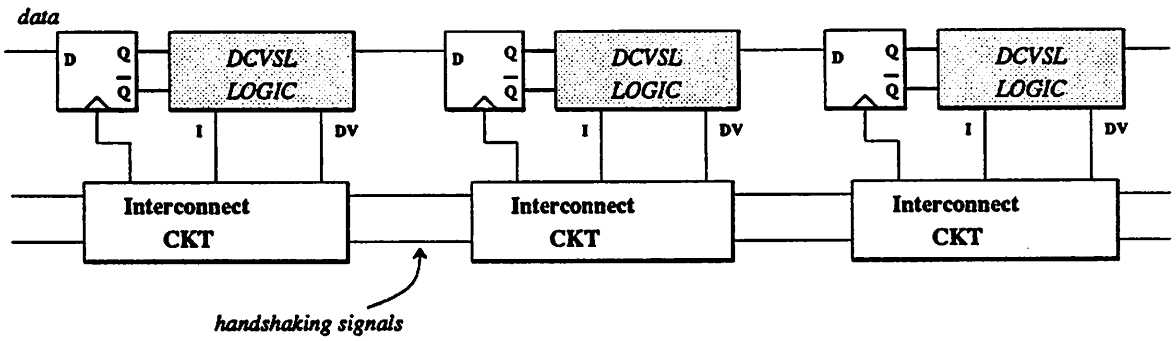


Figure 4.1: Block diagram of a self-timed system.

of a datapath for example.

Since interconnection or handshake circuits are separate from computation blocks, the overall system timing is simply that of the set of handshake circuits used. The delays of computation blocks just add a latency to handshake signals but the sequence of operations is maintained.

2. STG's for Describing Sequential Behavior

Once the system has been partitioned into self-timed blocks, an organized way of describing the transfers between blocks is required to synthesize handshaking circuitry. Both data and control signals must be described for correct timing of a block. Usually, a certain sequence of events must be imposed on the block in order for proper operation. For example, if a block has a single input and a single output, the correct sequence of events for proper operation will be that the current output must be transferred to the next stage before the next input is applied to the block. For a DCVSL logic stage, precharging is required between computations. A series of timing diagrams of the input and output signals of a block are sufficient to describe the required sequence but timing diagrams can often be difficult to interpret or manipulate. Another way of representing the information contained in timing diagrams is with *Signal Transition Graphs (STGs)* [59, 60, 61, 62]. STGs give a concise representation of a desired sequence and they can be manipulated to both check for timing problems and also synthesize speed-independent logic for implementing a handshake circuit.

In the past, Petri Nets have been utilized to model speed-independent asynchronous circuits. One problem in employing Petri Nets however, is that while the modeling may be accurate, using them as a synthesis tool often results in circuitry that is overly complex[57]. Signal Transition Graphs are a form of Petri Nets restricted by a set of axioms where transitions in nets are interpreted as signal transitions in a handshake circuit. The restrictions make STGs more amenable to analysis and manipulation for circuit synthesis due to reduced complexity while maintaining enough expressiveness to describe the behavior of almost all necessary handshake circuits for datapath applications.

An example of a simple STG is shown in Figure 4.2 for a circuit with the set of signals $J = \{Req, Ack\}$. The set of ***signal transitions***, denoted as T, is given by $J \times \{+, -\}$. The vertices of an STG represent events where one signal in the circuit makes a transition. The "+" denotes a rising edge while the "−" denotes a falling edge. Arcs in the graph between transitions represent instances of the causal relation, denoted by R, between transitions. The notation $t_1 R t_2$ means "t_1 causes t_2" and it represents a constraint between the transitions such that the firing of t_1 brings the system into a state in which t_2 is enabled to fire. Where two arcs come together (at their heads), an AND construct is implied, meaning that both of the events originating the two arcs must occur before the event to which they point is enabled. Formally a STG is defined as:

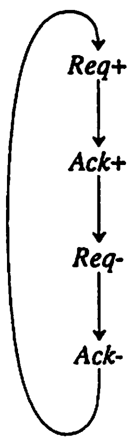


Figure 4.2: Simple signal transition graph.

Definition 4.1 *A STG defined on a finite set of signals J is represented by* $\Sigma_J = [T, R, M_0]$, *where $T = J \times \{+,-\}, R \subseteq T \times TandM_0 \subseteq R$ is the set of transitions which are enabled in an initial state of the circuit.*

Figure 4.2 shows a STG that represents a simple 4-cycle or reset-signalling handshake protocol at the input or output of one stage. The sequence that occurs for each data transfer is $Req^+ \rightarrow Ack^+ \rightarrow Req^- \rightarrow Acki^-$. This STG only shows the basic protocol at the input or output of a stage. Since it does not include both input and output sequences, it would not be useful in constructing any circuitry.

By representing the state of a handshake circuit as a binary number where each bit represents one signal in the the STG, the underlying state graph can be constructed from the signal transition graph. A circuit realization can then be determined from the state graph using traditional state diagram techniques [70, 71] as is commonly done for finite state machines. An important step however, is to manipulate the STG beforehand to ensure that the operation of the synthesized circuit will be correct and not become deadlocked. The simple rules to apply to a STG to ensure correct operation are described next.

2.1 Synthesis using STGs

In order to explain the synthesis procedure for circuits described by STGs, some definitions from speed-independent circuit theory are necessary [65, 66, 4]. Any handshake circuit must be defined in terms of a finite number of *states* in the set S where

Definition 4.2 *Each state* a *of S is represented by an* m*-tuple* a$= (x_1, x_2, \ldots, x_m)$ *where $x_1, x_2, \ldots, x_m$ are signals in the circuit.*

It is assumed here that binary signals are used to represent states and that implies that the maximum number of states $N = 2^m$. A set of sequences of states describes the behavior of the circuit where each sequence in the set is called an *allowed sequence.*

Definition 4.3 *A circuit is called* **speed independent** *if, for all allowed sequences starting in one state, each sequence ends up in the same state.*

If allowed sequences contain transitions of signals either sensed by or generated by the outside world, then the definition above is not always strong enough to ensure speed independence since different allowed sequences, while ending up in the same state, may not follow the desired sequence for correct interfacing to external signals. The definition below provides a stronger condition for speed independency in (practical) circuits containing inputs and outputs:

Definition 4.4 *A circuit is* **semi-modular** *if once a signal transition in the circuit is enabled, only firing of that signal transition can deactivate it.*

This property is sometimes referred to as *persistence* in the literature[59, 60, 64]. The underlying circuit represented by a STG of a handshake operation must absolutely meet the requirement of semi-modularity to function correctly under all conditions. An example of this is shown in Figure 4.3. Here, the signal $Reqi^+$ enables both $Acki^+$ and $Reqo^+$ in the graph. The graph fails to meet the semi-modularity requirement since if the loop on the left was implemented by faster circuitry than the loop on the right, the circuit might make the state transitions $Acki^+ \rightarrow Reqi^-$ *before* $Reqo^+$ occurs. Another way of stating the requirement is that when one signal enables another, the latter must fire before the first signal changes again. Fortunately, an STG can be checked (and corrected) for semi-modularity in an organized way.

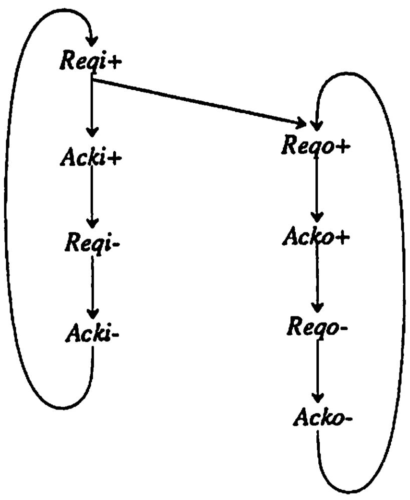


Figure 4.3: STG not possessing the property of semi-modularity.

Handshake circuits must also possess the property of *liveness* to function properly. Simply stated, a circuit is live if it does not become deadlocked. A deadlocked circuit will stay in one state and ignore requests for communication rendering it useless in a system. A test for liveness in a circuit represented by a STG is that for every signal in the graph, there exists one simple loop which contains both the high and low transitions of that signal. A simple form of an STG which represents a circuit which could become deadlocked contains a branch which does not lead to any other transitions. Since no other transitions are enabled, if the circuit enters that branch during operation, it ceases to respond to any further stimulus.

2.2 4-cycle protocol

The 4-cycle handshake protocol mentioned above is used in all of the circuits described. In the context of a typical computation stage, the protocol is defined here and some comments about its importance when using DCVSL logic are also discussed.

A typical computation block has at least a single input port and a single output port. The 4-cycle handshake sequence is used on each of these ports to control when computed data is fed to the next stage and when new data is accepted. Thus, there are commonly four handshake signals associated with a simple stage as shown in Figure 4.4. They are $Reqi, Acki, Reqo, Acko$. A single "cycle" of operation proceeds as follows: Assume that all four signals are initially low. When the preceding stage has valid data ready, it will raise *Reqi* to request a data transfer. When this stage is ready for a new data sample, it will latch the data and raise *Acki*, acknowledging the transfer. The $Acki^+$ transition allows the preceding stage to reset *Reqi* ($Reqi^-$) which in turn causes *Acki* to return low. When the computation of the stage is completed, *Reqo* will go high to signal to the next stage that data is valid. When the next stage latches in the new data, it raises *Acko*. This in turn allows *Reqo* to be lowered which in turn should be followed by the lowering of *Acko*. The term 4-cycle handshake is used to describe this protocol since each of the four handshake signal completes a full cycle or high/low transition during each transfer.

The choice of this protocol is based on constraints placed by the use of DCVSL as computation blocks. In a 2-cycle protocol a single *edge* of a handshake signal (either rising or falling) determines a data transfer. For example, if the *Req* and *Ack* signals between two stages are both low, the a request for a transfer is signalled by Req^+. The transfer is acknowledged by $Acki^+$. The two handshake signals stay at the high level until another transfer takes place which is signalled by the sequence $Reqi^-$, $Acki^-$. While this protocol can lead to faster handshake circuits since less transitions are required for each transfer, it is not compatible with DCVSL logic. The I signal of the DCVSL controls whether the logic is pre-charging or computing. The *level* of I is important, i.e. the logic family is level sensitive to its control signal. Since handshake signals control the DCVSL operation, they must also be level generating. A 2-cycle handshake protocol would require a prohibitive amount of extra circuitry to convert edges signalling data transfers to the levels required by DCVSL logic.

When following the 4-cycle protocol, the input port handshake signals always follow the sequence $Reqi^+ \rightarrow Acki^+ \rightarrow Reqi^- \rightarrow Acki^-$ during a single cycle of operation. Similarly, the output port handshake signals always follow the sequence $Reqo^+ \rightarrow Acko^+ \rightarrow Reqo^- \rightarrow Acko^-$. These two loops therefore become part of any STG in order to satisfy the 4-cycle protocol. The constraints added between the two loops to achieve a simple data transfer are very important in determining both the efficiency of operation and the correctness of operation of the stage. In the next section, the 4-cycle handshake circuit for a single pipelining stage will be derived.

3. 4-cycle Handshake Circuit

Envisage constructing an n-stage pipeline. In a clocked system, the stages could be all clocked by a single timing signal which would shift data down the pipe. Each stage would contain a register which acts as a shift register and some computational logic which must complete its task during the time taken by a single clock period. For a self-timed system, the datapath looks the same but rather than having a single global timing signal, each stage is controlled by a handshake circuit. This provides the signal to clock data into the register and it communicates with adjacent stages to negotiate transfers. In this section, the handshake circuit to perform this function will be synthesized. The 4-cycle protocol will be followed and the synthesis procedure will be explained in great detail to act as an example of the process. While an automated method for doing this is discussed later, this example shows the underlying tasks that take place. The 4-cycle basic handshake circuit or "HS4" is really the basic building block of many other handshake circuits so it is an appropriate example to detail. A block diagram of the 4-cycle handshake circuit is given in Figure 4.4. The computational block, when added to this circuit simply represents an added unknown latency in the *Reqo* signal.

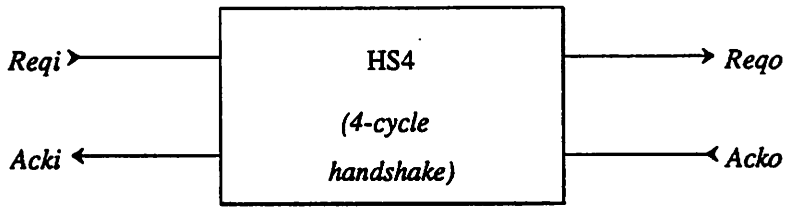


Figure 4.4: Block diagram for 4-cycle handshake circuit.

Figure 4.5a shows the two loops of a signal transition graph where basic 4-cycle handshake protocol is followed on the input and output ports. A condition linking the operation of the two ports has been added to define the handshake circuit for the entire stage. Conditions such as this are the essence of the timing of the stage. Too weak a condition might cause samples to be lost while excessively strong conditions might result in low hardware utilization due to long delays involved in waiting for the handshake signals to reach a certain state.

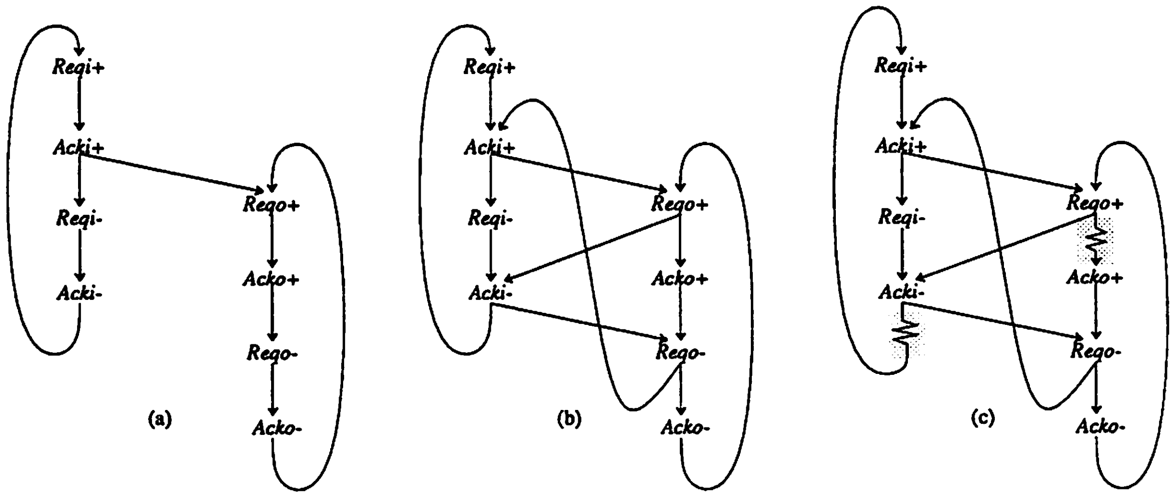


Figure 4.5: STG's for 4-cycle handshaking pipeline stage.

Accept for now that the condition shown in Figure 4.5a is the best choice. The

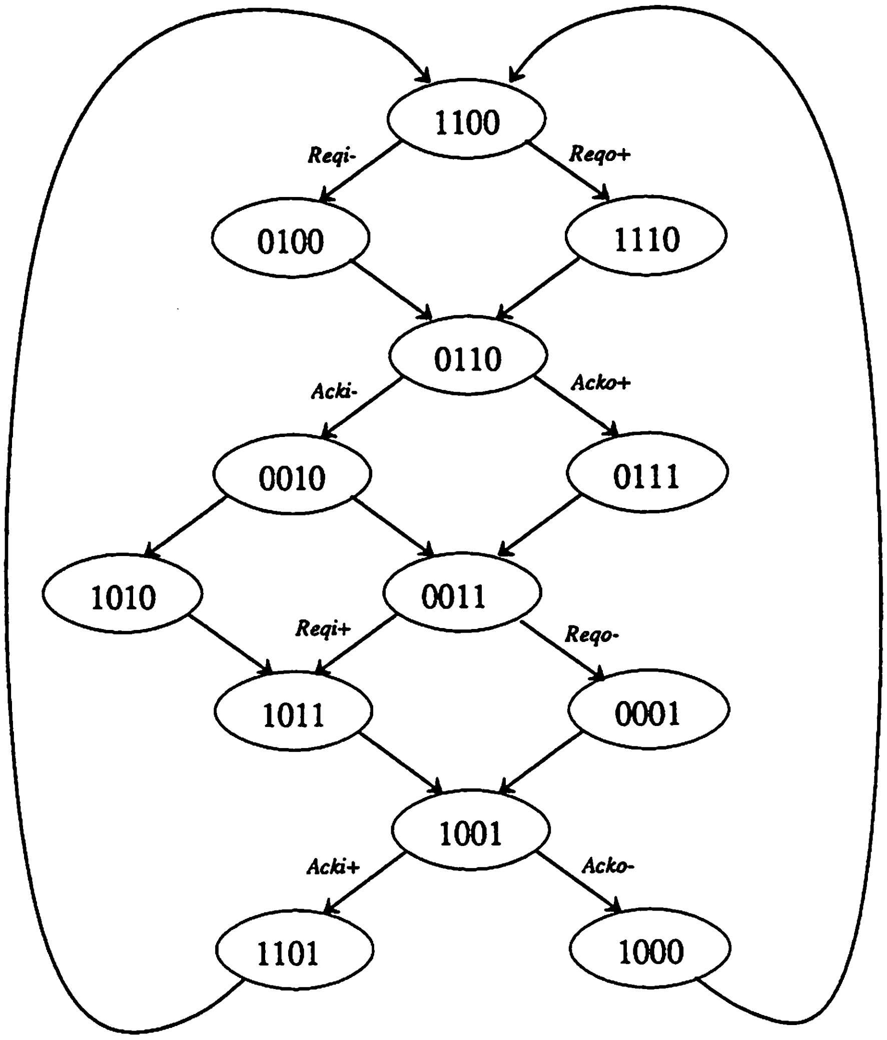
STG shown however must be checked for liveness and persistency before a valid circuit can be synthesized. Clearly, more arcs (constraints) must be added to satisfy the property of semi-modularity. This is done recursively since each arc added can create a condition in which persistence will be violated. $Acki^+$ enables $Reqo^+$ and therefore $Acki^-$ must be be disallowed until $Reqo^+$ actually occurs. The $Reqo^+ \rightarrow Acki^-$ arc is added to fix this however, examining the STG again, a new condition violates persistency. $Reqo^+$ enables $Acki^-$ and therefore $Reqo^-$ must be disallowed until $Acki^-$ actually occurs. The $Acki^- \rightarrow Reqo^-$ arc is added. This procedure is followed until persistency is satisfied and the resulting completed STG is given in Figure 4.5b. The STG is live since there are no "dangling" branches so it in now ready for circuit synthesis.

The state graph is now constructed from the completed signal transition graph by performing *state assignment* on the graph. The state graph is formally defined as [59]:

Definition 4.5 *A state graph of a STG Σ_J is represented by $\Phi_J = [S, T, \delta, s_0]$ where S is a set of states, s_0 is the initial state corresponding to the initial marking of the STG. Each $s \in S$ is a binary vector $[s(a), s(b), \ldots]$, where $J = a, b, \ldots$ is the set of signals in the graph and $s(j)$ denotes the value of signal j in state s. $\delta : SXT \rightarrow S$ is a partial function called the transition function; if the firing of transition t in state s leads to state s' then $\delta(s,t) = s'$.*

Thus, states are binary vectors representing the values of signals in the circuit, while transitions are transitions of these signals. Only a single signal is allowed to change between states. Using the state representation $s = [Reqi\ Acki\ Reqo\ Acko]$ for the example, the state graph is constructed and shown in Figure 4.6. The graph represents every allowed state of the signals in the STG. Where the STG splits into two arcs, the next state can be one of two different states since either of the transitions pointed to by the two arcs in the STG can occur first. No duplicate states occur in the stage graph of a STG which satisfies semi-modularity.

The goal here is to synthesize the logic required to generate the two outputs of the HS4 circuit: *Acki* and *Reqo*. From the state graph, a Karnaugh map of the circuit can be derived, however it is easier to use *reduced* state graphs for each of the outputs of interest.



s = [Reqi Acki Reqo Acko]

Figure 4.6: State Graph for the 4-cycle handshake circuit STG.

Acki

If the STG is redrawn only with the signals that have arcs connected to the signal *Acki*, then the reduced state graph can be derived for that signal. This is shown in Figure 4.7. Transitions for signals not affecting *Acki* (namely *Acko*) are just "shorted"

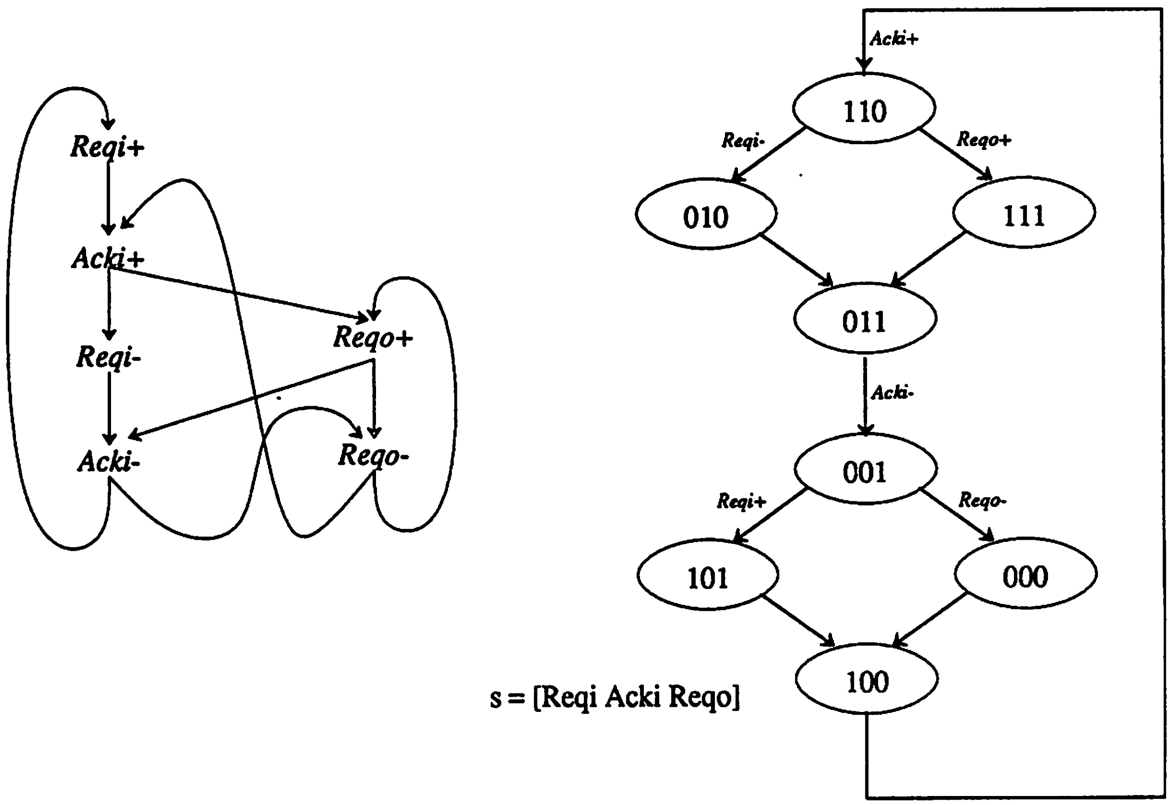


Figure 4.7: Reduced STG and state graph for signal *Acki*.

since they are unimportant for the timing of *Acki*. The reduced state graph for this new STG is also given in the figure. The elimination of one signal makes the logic synthesis more efficient. Figure 4.8 shows the Karnaugh map construction from the reduced state graph for *Acki*. Starting at some initial state, the state graph is traversed. For each state, the corresponding position in the Karnaugh map is filled with the value of the signal of interest in the *next* state(s). The x denotes the initial state used in the example: 1 1 0 . A position not traversed in the Karnaugh map *or* a position which points to next state values of *both* 0 and 1 should be filled with an X or "don't care". Using the grouping shown with dotted lines in the figure, the logic for *Acki* is found:

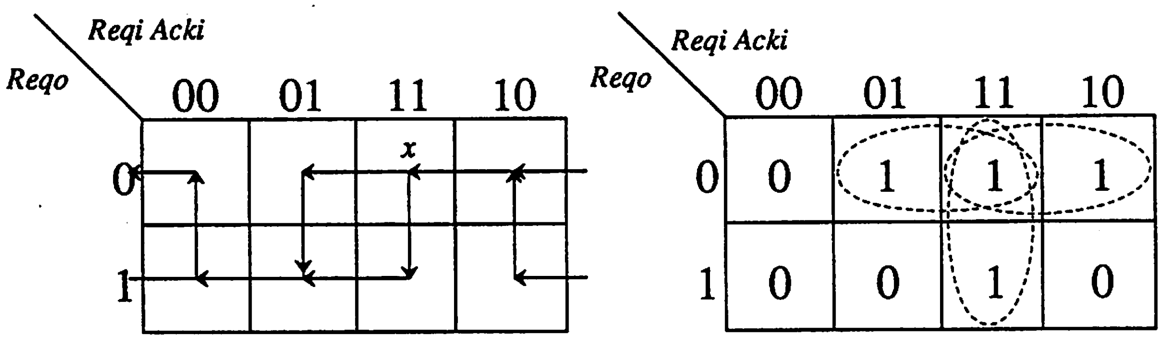


Figure 4.8: Karnaugh map construction for *Acki*.

$$\begin{aligned} Acki &= ReqiAcki + Reqi\overline{Reqo} + Acki\overline{Reqo} \\ &= Reqi\overline{Reqo} + Acki(Reqi + \overline{Reqo}) \qquad (4.1) \end{aligned}$$

Remembering that the logical equation for a SR-latch is

$$Q = S + Q\overline{R}$$

the logic can be expressed in the form of a single latch with several accompanying gates.

$$\begin{aligned} S &= Reqi\overline{Reqo} \qquad (4.2) \\ R &= \overline{(Reqi + \overline{Reqo})} = \overline{Reqi}Reqo \qquad (4.3) \end{aligned}$$

Reqo

The same procedure is followed for *Reqo* and the reduced graph and Karnaugh map are shown in Figures 4.9 and 4.10 respectively. From the Karnaugh map grouping shown, the logic for *Reqo* is found:

$$\begin{aligned} Reqo &= ReqoAcki + Reqo\overline{Acko} + Acki\overline{Acko} \\ &= Acki\overline{Acko} + Reqo(Acki + \overline{Acko}) \qquad (4.4) \end{aligned}$$

$$\begin{aligned} S &= Acki\overline{Acko} \qquad (4.5) \\ R &= \overline{(Acki + \overline{Acko})} = \overline{Acki}Acko \qquad (4.6) \end{aligned}$$

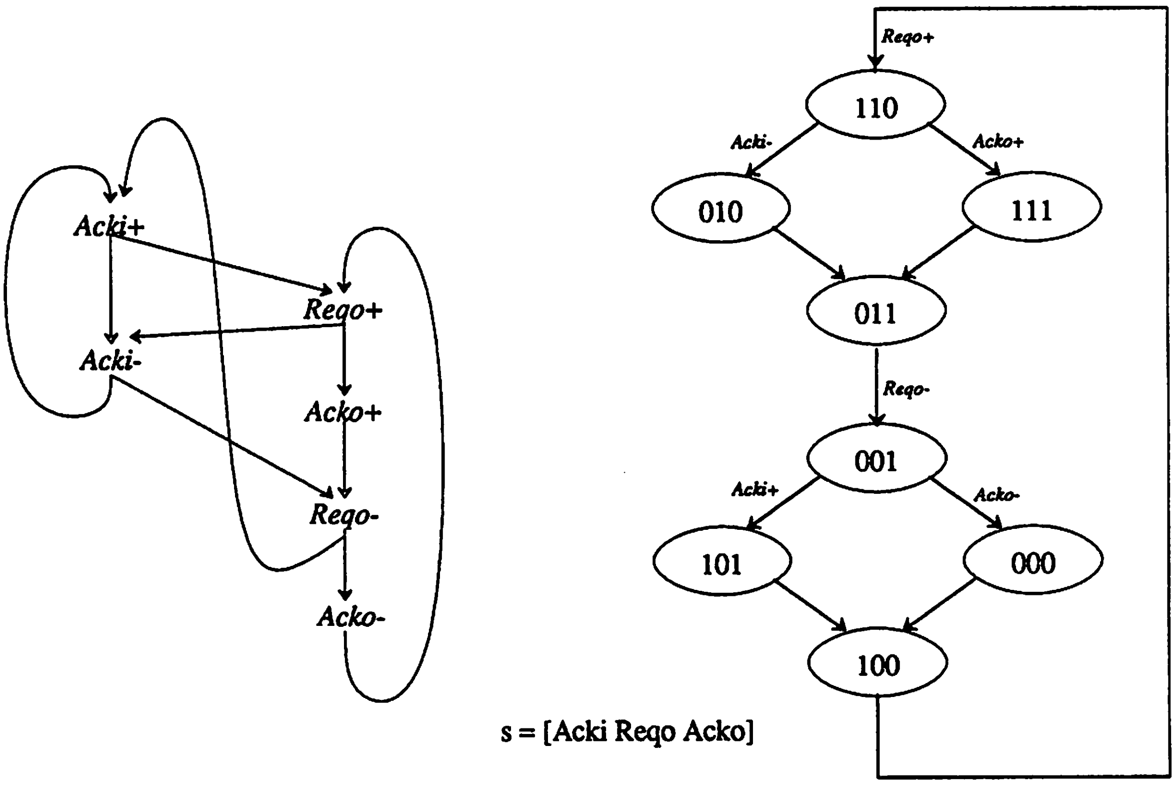


Figure 4.9: Reduced STG and state graph for signal *Reqo*.

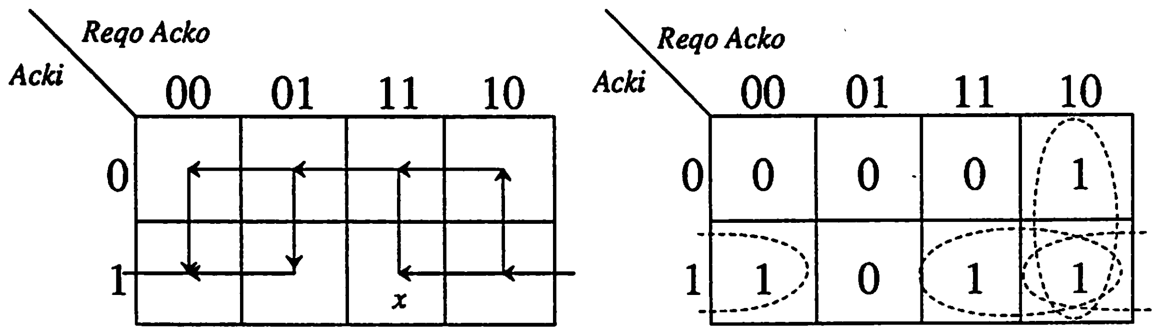


Figure 4.10: Karnaugh map construction for *Reqo*.

The completed 4-cycle handshake circuit is shown in Figure 4.11. As labeled in the figure, the SR-latches are simple transparent latches and *not* full registers. Careful

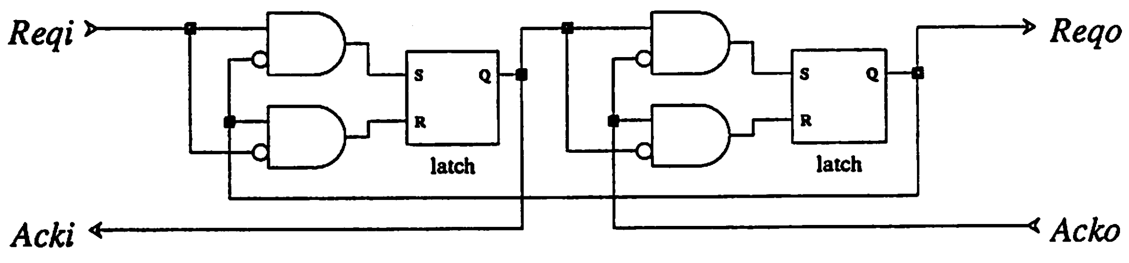


Figure 4.11: Drawing of 4-cycle handshake circuit.

inspection of the logic reveals that each section is really an implementation of a Muller c-element where one input is inverted. Therefore, a simplified drawing of the HS4 circuit can be made as illustrated in Figure 4.12. C-elements are a basic building block of handshake circuits and they will be used often in schematics presented. The actual design of the c-element is further discussed in Chapter 5.

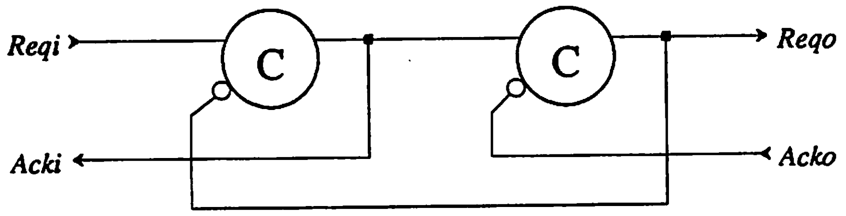


Figure 4.12: Drawing of 4-cycle handshake circuit using Muller c-elements.

The connections between the HS4 circuit and a self-timed logic block are necessary to complete the design of a pipeline stage. The signal *Acki* is acknowledging receipt of data at the input to the stage so it is used to clock the actual data register of the stage. The signal *Reqo* tells the next stage that valid data is ready. By connecting the self-timed logic "compute" signal to *Reqo* and then using the completion signal form the logic as the request to the next stage, the communication waits the proper amount of time for the logic to do its task. A typical self-timed DCVSL pipeline stage containing the HS4 circuit is illustrated in Figure 4.13. The timing sequence of the circuit remains the same as the

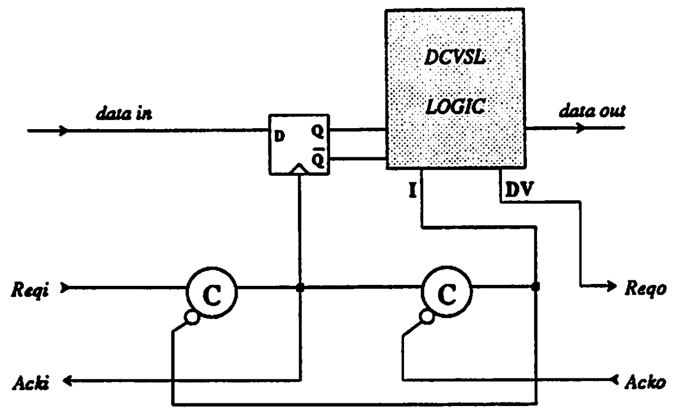


Figure 4.13: Connection of a DCVSL logic block to the 4-cycle handshake circuit.

STG specification. There is a temporal difference in the signals however because of added latency of the computational circuitry. This is ideal in that the time taken for each stage is exactly that of the computation. In a cascade of these circuits, the throughput will be limited by the slowest stage just as in a synchronous pipeline. There is also the overhead of the handshaking circuit itself. The overhead for the whole pipeline for handshaking is that of a single HS4 circuit. By adding symbolically the computation delays to the STG of a stage, more evaluation can be done in terms of circuit efficiency. The computation delays are assumed to be greater that the handshake circuit latch delays. In Figure 4.5c the STG has been modified to show the computation delays as shaded zig-zag lines. This notation will be used in the next example to make a comparison of circuit efficiencies.

3.1 Other HS4 circuits

In the last subsection, you were asked to accept as optimum the starting specification on the 4-cycle handshake stage STG, $[Acki^{+} \rightarrow Reqo^{+}]$. It was suggested however that the choice of this specification is very important in determining the operation/design of the circuit. By looking at some alternative specifications this can be revealed. For example, the novice designer might construct the specification $Reqi^{+} \rightarrow Reqo^{+}$ as a starting point. This sounds correct, at least at first. A request at the inputs enables a request at the output (with computation in between). Using this specification and applying the tests for persistency yields the STG shown in Figure 4.14a. The specification is shown

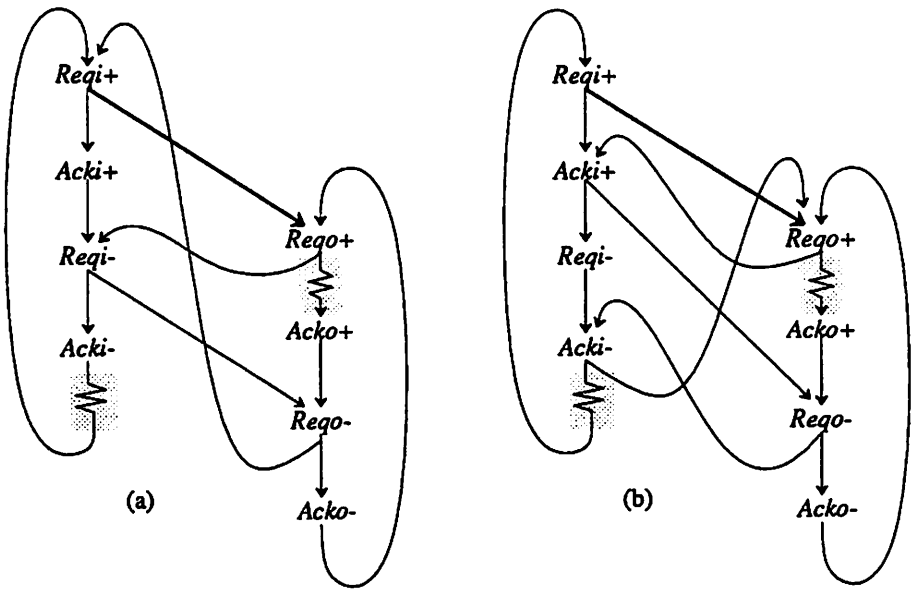


Figure 4.14: STGs for the specification $Reqi^+ \rightarrow Reqo^+$.

with a heavier line for the arc. Blindly applying the persistency tests has led to an initial problem in that arcs have been added that enable *Reqi*, which is an input to the circuit. A distinction must be made between the inputs to the circuit and the outputs that the circuit generates since there is no control over the sequencing of inputs. A modified STG that also meets the persistency test and does not control circuit inputs is given in Figure 4.14b. The corresponding circuit diagram for the alternate HS4 circuit is shown in Figure 4.15. For this specification, the data register of the stage would have to be clocked by *Reqi*. Examining the STG for the circuit, one can find a simple[1] loop through the graph which contains *both* computation delays. This means that the delay of the stage could be as long as the computation times of two stages, i.e. a 50% hardware utilization. This represents an obvious disadvantage to this specification which does not exist in the STG shown in Figure 4.5. It makes sense to expect this behavior because the specification states that a *request* at the input port initiates computation. The request signal for the next sample may arrive well before the computation on the current sample is complete. This implies every other stage must act to store the data while waiting for the next stage computation to finish. The specification of the last section uses the *Acki* Signal to initiate

[1]A simple loop in a signal transition graph does not pass through any transition more than a single time.

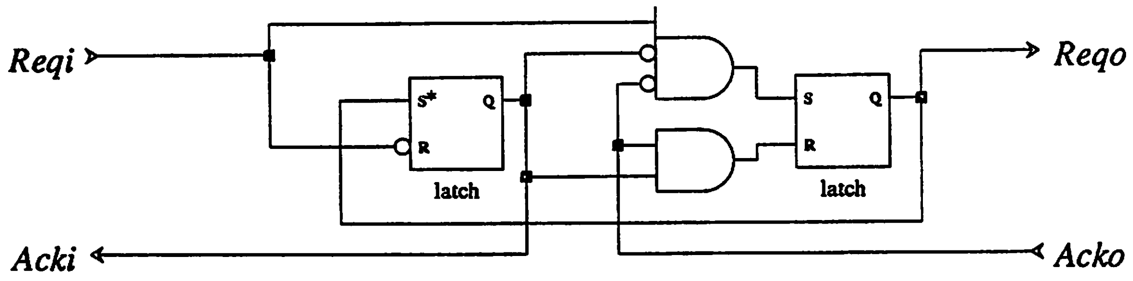


Figure 4.15: HS4 Circuit for the STG shown in Figure 4.14. The "*" means that the first latch is set-dominant, i.e. if S and R are both HIGH , the output is set.

computation, taking full advantage of the data registers of each stage to allow every stage to participate in concurrent computation. Similarly, the reader is left to try the specification [$Reqi^+ \cap Acko^- \rightarrow Reqo^+$], where $\cap$ denotes the AND construct. This also leads to a circuit which is less efficient [64].

3.2 Assumptions on Delay Matching

The circuit of Figure 4.13 makes several assumptions on element delay times. The required specifications for the data input D-register are shown in TableIII. Strictly speaking, a self-timed circuit does not require its elements to meet any specific delay constraints. However, in reality, making such a circuit would be difficult if not impossible to design, expensive in terms of complexity and die area, and more important, unnecessary. For example, making a D-register with a completion signal output and increasing the complexity of the 4-cycle handshake circuit to monitor when data is valid at the output of the register would remove the delay specification on the register itself. But this is overkill as it is usually not difficult to ensure that the register meets the specifications shown in the table. In fact, the same exercise must be performed when designing synchronous clocked circuits. Simplifications such as this, in the interests of silicon efficiency and higher performance, yield working circuits and require design criteria much the same as those employed in synchronous designs.

Table III	D-register Specifications
τ_{setup}	$< \tau_{Reqi+ \rightarrow Acki+}$
τ_{hold}	$< \tau_{Acko+ \rightarrow Reqo-}$
τ_{delay}	$< \tau_{Acki+ \rightarrow I+}$

4. Higher level description for synthesis

In the examples of the last section, a signal transition graph had to be constructed for each handshake circuit to be synthesized. Also, the choice of the starting point or weakest conditions in the graph had a great effect on the circuit realization and its efficiency during operation. It would be very desirable to automate the synthesis procedure and dispense with the need to construct an entire STG for each design. Meng[64, 66] studied this problem and found that a subset of Dijkstra's[63] guarded commands formed a good basis for describing handshake STG specifications. A guarded command is a statement list prefixed by a Boolean expression. When the Boolean expression becomes true, the statement list is enabled for execution. Only the subset of the guarded commands that apply to deterministic conditions are presented since others involve metastable circuits which are not used for DSP applications. The list below describes the deterministic guarded commands:

Basic Construct: $[C \rightarrow S]$
where C is a *pre-condition* and S is a
list of statements that are to be executed if C is true.

AND Construct: $[C_1 \cap C_2 \cap \cdots \cap C_n \rightarrow S]$
where C_i is a pre-condition and S is to be executed if
all C_i are true.

OR Construct: $[C_1 \cup C_2 \cup \cdots \cup C_n \rightarrow S]$
where C_i is a pre-condition and S is to be executed if
any C_i is true. For the purposes of determinism,
only one of the pre-conditions can be true at one time.

Sequential Construct: $[C_1 \rightarrow S_1; C_2 \rightarrow S_2]$
where C_2 can be tested only after S_1 has been executed.

Parallel Construct: $[C_1 \rightarrow S_1 \parallel C_2 \rightarrow S_2]$
where two clauses $C_1 \rightarrow S_1$ and $C_2 \rightarrow S_2$ can
be processed concurrently.

Alternative Construct: $[C_1 \rightarrow S_1 | C_2 \rightarrow S_2]$
where S_i is executed only after C_i is true, but only one of
the pre-conditions can be true.

Repetitive Construct: $^*[C \rightarrow S]$

where the clause $[C \rightarrow S]$ is to be repeatedly executed.

The first design of an HS4 circuit above used the guarded command specification $^*[Acki^+ \rightarrow Reqo^+]$. Similarly, the second design used the guarded command specification $^*[Reqi^+ \rightarrow Reqo^+]$ which is shown as the heavier line in the STGs of Figure 4.14. By using this language description of a handshake specification, the designer can concentrate on the most important part of the graph and try different conditions rapidly. Meng wrote a program in the LISP programming language that reads a guarded command specification and generates automatically the handshake circuit Boolean equations. This is done in several steps. First, the 4-cycle loops in the STG are added and the STG is recursively checked for semi-modularity until it is correct. Where conditions are added, they are made not to affect input signals. After a correct STG is constructed, the logic synthesis is performed using one of several standard techniques.

The use of the program to generate the circuit in Figure 4.15 is shown below: The input specification of the guarded command must be given in a LISP format.

```
;;; Guarded command specification for *[Reqi+ -> Reqo+]
(presyn '((source (Reqi Acki)) (destination (Reqo Acko))
(condition nil) ((Reqi+) nil (Reqo+))))
```

The source and destination port signals are identified and then the guarded command specification is given. A basic construct is used, so the conditions are nil. The session of running the program is shown next.

```
ASYNC LOGIC SYNTHESIS: VERSION 0.0 (under Franz Lisp 43.1)
-> (load 'f2.1)
[load f2.1]
signal_tranistion_graph
(t ((Reqi- Acki-) (Acki+ Reqi-) (Acki- Reqi+) (Reqo+ Acko+) (Reqo- Acko-) (Acko+
 Reqo-) (Acko- Reqo+) (Reqi+ Acki+ Reqo+)))
semi-graph
(t ((Reqi- Acki-) (Acko+ Reqo-) (Acko- Reqo+) (Reqi+ Acki+ Reqo+) (Reqo+ Acko+ A
cki+) (Acki+ Reqi- Reqo-) (Reqo- Acko- Acki-) (Acki- Reqi+ Reqo+)))
t
-> (synthesis semi-graph 'Acki)
Acki*Reqi+Reqo
-> (quick_syn 'Reqo)
~Acki*~Acko*Reqi+~Acki*Reqo+Reqo*~Acko
-> (quick_syn 'Reqi)
~Acki
-> (quick_syn 'Acko)
Reqo
-> (exit)
```

The list signal_transition_graph is the STG before being checked for semi-modularity. Each list consists of a transition followed by all transitions enabled by it, i.e. the arcs of the graph originating at that transition. The list semi-graph gives the semi-modular STG. Then the logic to generate each signal is generated. As a check, the logic for input signals *Reqi* and *Acko* was generated. *Reqi* is simply $\overline{Acki}$ as the 4-cycle handshake defines. Similarly, *Acko* is just *Reqo*. (These signals originate from adjacent stages)

The automation procedure substantially reduces the design time for reliable handshake circuits. It also allows more experimentation to be performed when choosing weakest conditions. This is important since in spite of the streamlining of the design procedure, there can still be some uncertainty in defining the handshake conditions for a datapath or system.

5. Other Common Handshake Circuits

Using the automated synthesis procedure, a library of common handshake circuits can be built up for future use. A library of self-timed circuits has the advantage that it is based on a behavior sequence alone. Therefore, the library can follow scaled technologies without re-design. In this section, some of the more common handshake circuits are shown along with their guarded command specifications. The HS4 circuit described above of course is the most common circuit and it is used between any two pipeline stages.

5.1 Sequential HS circuit

The circuit of Figure 4.12 implements a sample delay pipeline stage when connected to a computation block. In some cases, it is necessary to perform two functions sequentially with a single pipeline delay. A sequential handshake circuit provides this function. Using the guarded command $*[Reqi^+ \rightarrow Reqo^+; Acko^+ \rightarrow Acki^+]$ ensures that the next stage receives a request and performs its computation before an acknowledgment signal is sent to the previous stage. Thus, two blocks compute in sequence with a single pipeline delay. If we think of using DCVSL logic as the computation blocks, a trivial connection to also provide this sequencing would be to just connect the *Reqo* of the first block ($\mathbf{DV_1}$) to the initialization signal I of the next block. While this provides the correct sequencing during computation, it also imposes the same sequence during the pre-charge state of the logic. The sequential handshake circuit provides means to concurrently[2] pre-charge the two blocks while maintaining the sequenced operation during computation. The completed signal transition graph from the guarded command specification and the synthesized circuit for the sequential handshake are shown in Figure 4.16.

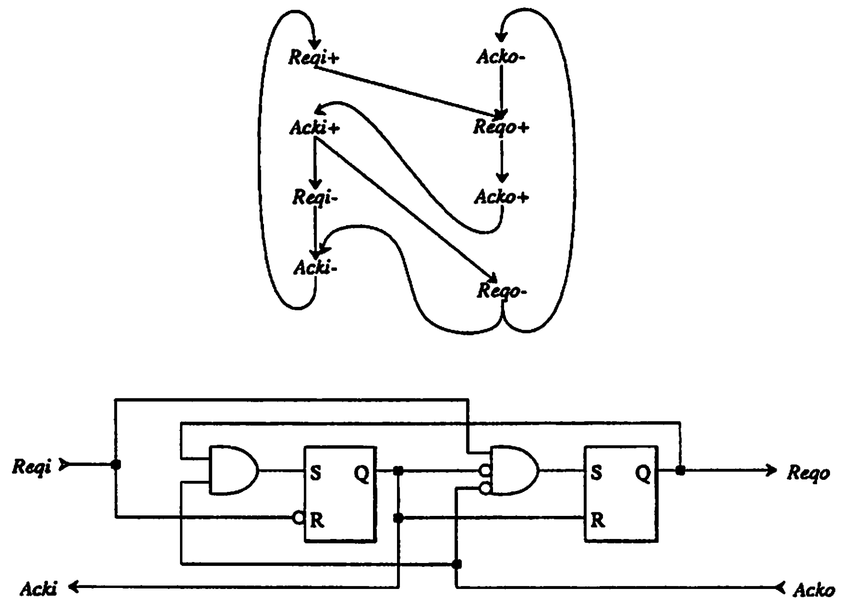


Figure 4.16: Sequential handshake STG and circuit from the specification $*[Reqi^+ \rightarrow Reqo^+; Acko^+ \rightarrow Acki^+]$.

5.2 2-Source, 1-destination HS circuit

When multiple inputs to a computational block originate from several sources, handshaking must be completed with each source to make sure all the inputs are valid. The guarded command specification for a block with two input sources is $*[Acki_1^+ \cap Acki_2^+ \rightarrow Reqo^+]$. The synthesis procedure yields the circuit of Figure 4.17 for this function.

5.3 1-Source, 2-destinations HS circuit

Similarly, when a single computational block must feed outputs to several destination blocks the guarded command specification used is $*[Acki^+ \rightarrow Reqo_1^+, Reqo_2^+]$. The circuit for this handshaking operation is shown in Figure 4.18.

[2]Actually, the delay of the handshake circuit itself separates the pre-charge start times of the two blocks

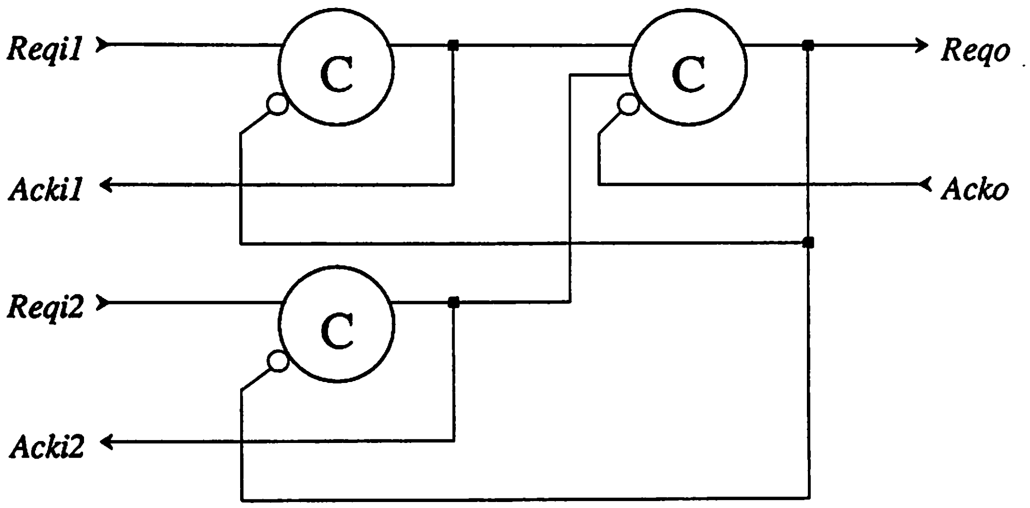


Figure 4.17: Handshake circuit for block with two input sources. The guarded command is $*[Acki_1^+ \cap Acki_2^+ \rightarrow Reqo^+]$.

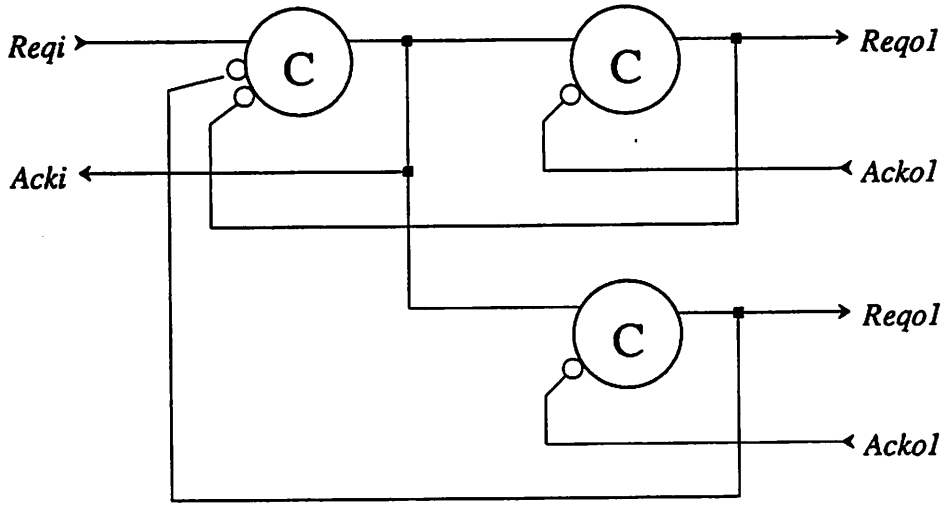


Figure 4.18: Handshake circuit for block with two output destinations. The guarded command is $*[Acki^+ \rightarrow Reqo_1^+, Reqo_2^+]$.

5.4 2-in Multiplexer HS circuit

Multiplexers often re-configure a datapath architecture for different types of processor instructions. When a multiplexer is employed, its control signal determines both the datapath architecture and the required handshaking architecture. When data from an input of the MUX passes to the output, then handshaking must be completed with the source feeding that input. A simple guarded command describing this (continuing to use the 4-cycle protocol) is $*[(Acki_1^+ \cap T) \cup (Acki_2^+ \cap \bar{T}) \rightarrow Reqo^+]$. The LISP format input to the synthesis for the guarded command is

```
    ;;; Guarded Command for 2 input MUX
;;; with NO controller handshaking.
;;;
(presyn '((source (Reqi1 Acki1) (Reqi2 Acki2)) (destination (Reqo Acko))
(condition T) ((Acki1+) (T+) (Reqo+)) ((Acki2+) (T-) (Reqo+)) ))
```

The OR construct can be synthesized with semi-modular models with the constraint that only one side of the OR is true at a time. This will always be true in this case since the control signal signal T and its complement $\bar{T}$ occupy the two sides of the OR . Since the control signal determines the architecture of the handshake circuit itself, there is a STG for each state of T.

The output of the synthesis program shows the two STGs for the MUX handshake circuit, denoted by the T^+ and the T^- sections of the list semi-graph.

```
    ASYNC LOGIC SYNTHESIS: VERSION 0.0 (under Franz Lisp 43.1)
-> (load 'mux_gc2)
[load mux_gc2.l]
signal_tranistion_graph
((T+)
     ((Reqi1+ Acki1+) (Reqi1- Acki1-) (Acki1- Reqi1+) (Reqi2+ Acki2+)
      (Reqi2- Acki2-) (Acki2+ Reqi2-) (Acki2- Reqi2+) (Reqo+ Acko+)
      (Reqo- Acko-) (Acko+ Reqo-) (Acko- Reqo+) (Acki1+ Reqi1- Reqo+))
 (T-)
     ((Reqi1+ Acki1+) (Reqi1- Acki1-) (Acki1+ Reqi1-) (Acki1- Reqi1+)
      (Reqi2+ Acki2+) (Reqi2- Acki2-) (Acki2- Reqi2+) (Reqo+ Acko+)
      (Reqo- Acko-) (Acko+ Reqo-) (Acko- Reqo+) (Acki2+ Reqi2- Reqo+))
)
semi-graph
((T+)
     ((Reqi1+ Acki1+) (Reqi1- Acki1-) (Reqi2+ Acki2+) (Reqi2- Acki2-)
      (Acki2+ Reqi2-) (Acki2- Reqi2+) (Acko+ Reqo-) (Acko- Reqo+)
      (Acki1+ Reqi1- Reqo+) (Reqo+ Acko+ Acki1-) (Acki1- Reqi1+ Reqo-)
      (Reqo- Acko- Acki1+))
 (T-)
     ((Reqi1+ Acki1+) (Reqi1- Acki1-) (Acki1+ Reqi1-) (Acki1- Reqi1+)
      (Reqi2+ Acki2+) (Reqi2- Acki2-) (Acko+ Reqo-) (Acko- Reqo+)
      (Acki2+ Reqi2- Reqo+) (Reqo+ Acko+ Acki2-) (Acki2- Reqi2+ Reqo-)
      (Reqo- Acko- Acki2+))
)
t
-> (synthesis semi-graph 'Reqo)
Acki1*~Acko*T+Reqo*Acki1*T+~Acko*Acki2*~T+Reqo*Acki2*~T+Reqo*~Acko
-> (quick_syn 'Acki1)
~Reqo*Acki1*T+~Reqo*Reqi1+Acki1*Reqi1+Reqi1*~T
```

```
-> (quick_syn 'Acki2)
~Reqo*Acki2*~T+Reqi2*T+~Reqo*Reqi2+Acki2*Reqi2
-> (quick_syn 'Acko)
Reqo
-> (quick_syn 'Reqi1)
~Acki1
-> (quick_syn 'Reqi2)
~Acki2
-> (exit)
```

The circuit diagram of the MUX handshake is given in Figure 4.19 along with a simplified c-element equivalent. The circuit and Boolean equations may seem a bit confusing at first, but much of the logic implements MUXes in the handshake circuit itself since the control signal T re-configures the circuit between two forms. This is shown more clearly in the equivalent circuit. The rightmost MUX chooses which input port to communicate with. The two MUXes on the left, connected to $Reqi_1$ and $Reqi_2$ essentially just make their respective c-elements transparent when their input is not in use. Note that a c-element is a latching device. To make it transparent to an input signal, then that input signal must be connected to *all* inputs of the c-element. It is equivalent to connecting all inputs of an AND gate to one input signal in order to make that gate transparent. In the STG for the case when T is HIGH , input port 1 is activated and the graph for input port 2 is a simple loop $Reqi_2^+ \rightarrow Acki_2^+ \rightarrow Reqi_2^- \rightarrow Acki_2^-$. This is why the synthesized circuit will pass $Reqi_2$ directly to $Acki_2$. It could be undesirable in a real circuit to have the deselected input be transparent to requests because the handshake circuit acknowledges data that is not being used. It is really a result of *not* specifying any conditions on the unused port and ending up with a simple 4-cycle loop. There is not a convenient way for expressing a condition that causes the unused input port to not acknowledge any requests in the guarded command, however it is a trivial circuit change. By moving the left-hand MUX inputs connected to the $Reqi$ signals to Ground (logical 0), the deselected port will not receive an acknowledge. The c-element on the deselected input port stays in the LOW state if one input is held LOW .

Since the control signal T re-configures the circuit, it is important to have in a settled state before a new request. Depending on the system design, this may require handshaking to the controller which generates T. An appropriate guarded command specification is $*[(Acki_1^+ \cap T \cap Acki_C^+) \cup (Acki_2^+ \cap \bar{T} \cap Acki_C^+) \rightarrow Reqo^+]$. The circuit complexity is increased by another c-element and several gates[64].

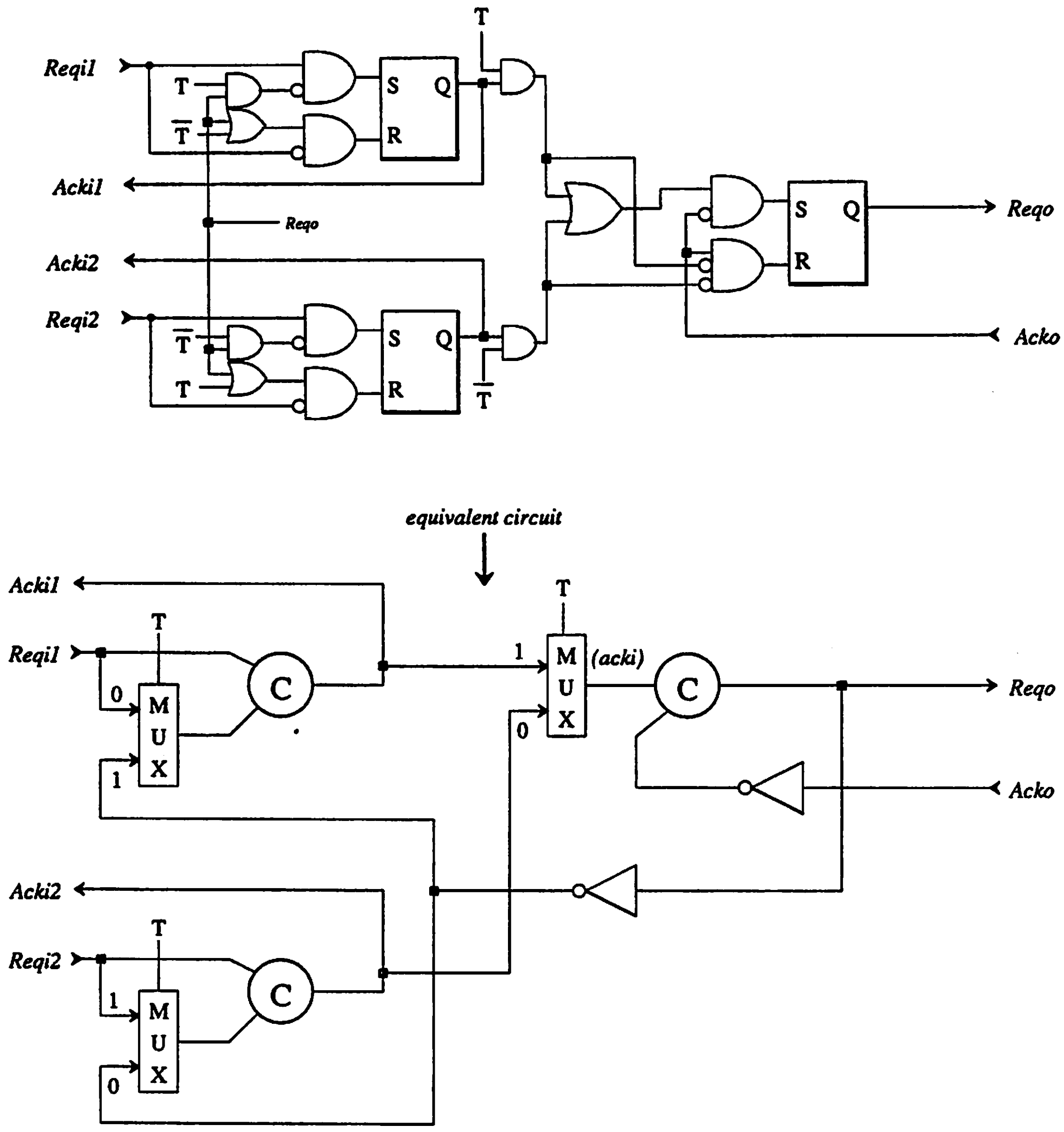


Figure 4.19: Handshake circuit for a 2-input MUX stage.

5.5 2-out Demultiplexer HS circuit

A 2-output demultiplexer can be synthesized with the specification $^*[Acki^+ \cap T \rightarrow Reqo_1^+ \parallel Acki^+ \cap \bar{T} \rightarrow Reqo_2^+]$. The handshake circuit is shown in Figure 4.20. Similar to the MUX circuit, the deselected port (now an output port) becomes transparent to handshake signals. Again, by grounding the input to each right-hand MUX that is shown connected to $\overline{Acko_n}$, the deselected output port will not pass handshake signals.

6. Summary

The synthesis of reliable handshaking circuits is a vital part of any self-timed system design. The interconnection of handshake circuits defines the system timing. When using a level sensitive logic family such as DCVSL to implement computation blocks which provide a completion signal, the appropriate handshaking protocol to use is the 4-cycle protocol. In order to design circuits that meet both the sequencing and 4-cycle specifications of a stage, signal transition graphs can be used to describe the desired behavior. Performing state assignment on a STG allows for the generation of the Boolean equations that define the handshaking logic in traditional ways. While STGs are more convenient than timing diagrams for describing the sequential behavior of a stage, they still can be confusing and error prone. By using a subset of Dijkstra's guarded commands, the most significant portion of the handshaking specification can be expressed in a simple and high level form. Automatic means for generating the handshake circuit directly from a high level description language have been demonstrated. Using the synthesis program, a library of common handshake circuits can be synthesized and used in many designs. Some of the most common handshake circuits required were presented in this chapter.

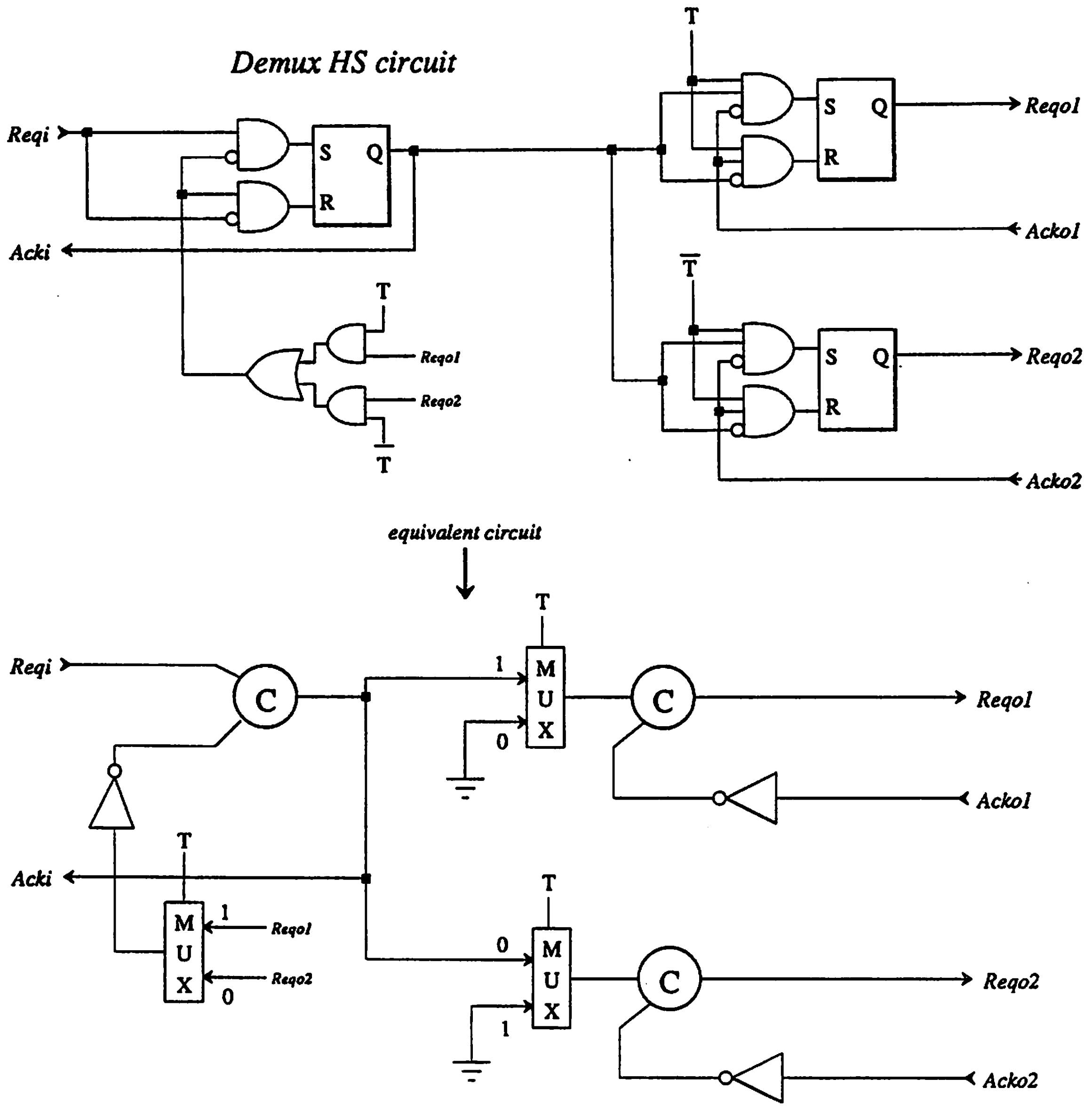


Figure 4.20: Handshake circuit for a 2-output DEMUX stage.

Chapter 5

C-elements for Handshaking Circuit Design

As seen in Chapter 4, a common component found in most handshake circuits is the Muller c-element. Because the performance of a handshake circuit critically depends on the c-element design, a detailed study was done on the design of such elements. A circuit design for the self-timed DSP was chosen based on the study. This chapter presents the design of several types of c-elements and gives a performance comparison between them. In the last section, the placement of buffers in handshaking circuits, often a necessity at the outputs of c-element driving large macrocells, is discussed.

1. C-element design

The Muller c-element is a basic building block of many useful handshake circuits. In Chapter 4, when studying the signal transition graphs for describing a self-timed stage, the assumption was made that the time for arcs representing computation was much longer than the time for arcs representing handshake circuit state changes. This is normally the case, however the time required for the handshake circuits to operate is finite and it reduces the overall efficiency of the system. Therefore, while a self-timed system will compensate for varying delays of computation blocks, the fixed overhead introduced by the handshaking circuitry should be minimized if possible. This means applying effort to the design of the c-element since it is so often required. In this section, the design of a fast c-element is discussed.

Figure 4.12 showed the basic 4-cycle protocol handshake circuit which requires two c-elements. One logic implementation for the circuit was shown in Figure 4.11. The most straightforward implementation of the c-element is to use cross-coupled NOR gates for the latch as illustrated in Figure 5.1. The signal $INIT_0$ allows the latch to be reset for initialization. In real applications, it is useful to be able to reset or set the latch for the purpose of initializing the handshaking network to any particular state. However, there are some ways around this which will be discussed in a later section. A static CMOS

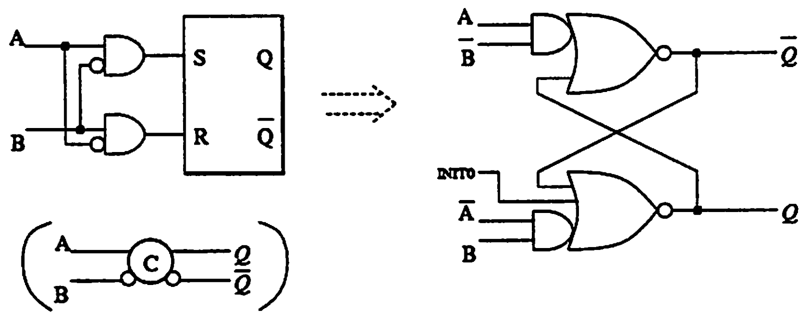


Figure 5.1: Cross-coupled NOR c-element implementation.

design[1] for the c-element latch is shown in Figure 5.2. It combines the input AND gates with the NOR gates making up the bistable element to make a single complex CMOS gate. This design is reliable and easy to design, however it suffers from a lower speed than can be achieved with other circuit techniques. Also, the device count is relatively high. In the c-element of Figure 5.1, the time to *Set* the latch is effectively two gate delays because the $\bar{Q}$ output must change before the Q output can change state. The dual cross-coupled circuit avoids the extra delay for *Set*ting the latch by using NAND gates as shown in Figure 5.3. Adding initialization inputs to force the latch into a certain state adds several more devices. Because of the complexity and ensuing slower speed of the basic CMOS c-element designs in Figures 5.1 and 5.3, other designs were investigated.

A Muller c-element can be made by using a majority function gate, typically used for the *carry* portion of a full adder[19]. The majority function is defined as:

$$\begin{aligned} f(A,B,C) &= AB + BC + AC \\ &= AB + C(A+B) \end{aligned} \tag{5.1}$$

[1]All schematics give device sizes in λ as defined in the MAGIC layout editor. For a 2μ process, $\lambda = 1\mu$.

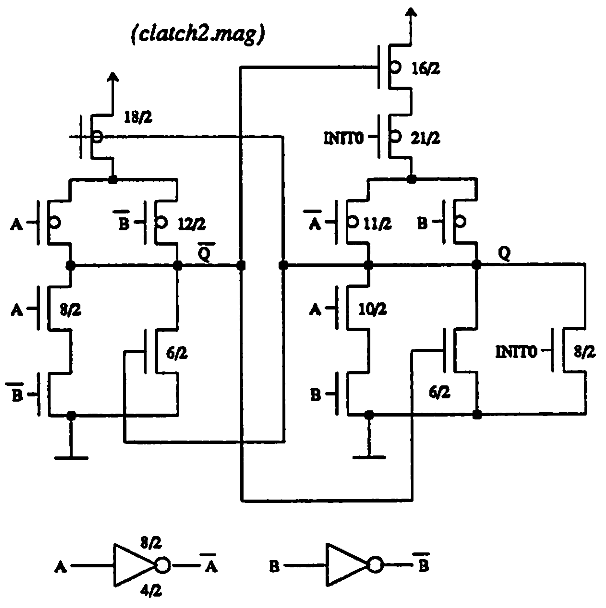


Figure 5.2: CMOS circuit for cross-coupled NOR c-element implementation.

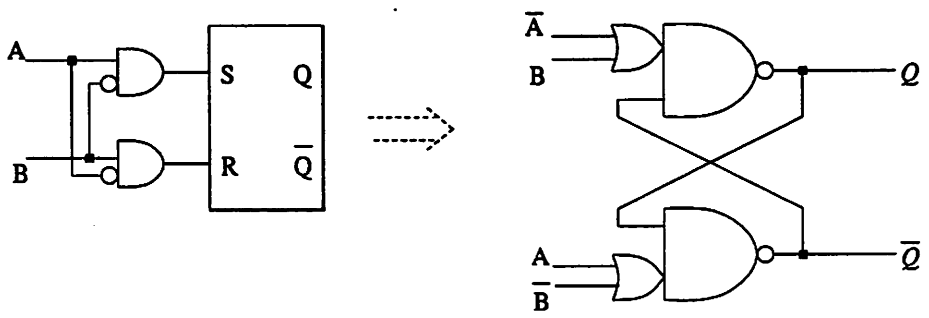


Figure 5.3: Cross-coupled NAND c-element implementation.

If the C input is taken from the output of the gate as shown in Figure 5.4, the equation becomes the familiar equation for a latch:

$$\begin{aligned} Q &= S + Q\bar{R} \\ Q &= AB + Q(A + B) \qquad (5.2) \end{aligned}$$

where the *Set* function is AB and the *Reset* function is $\overline{A+B} = \bar{A}\bar{B}$, precisely the logic equation for a c-element. A single complex CMOS gate version of the majority function

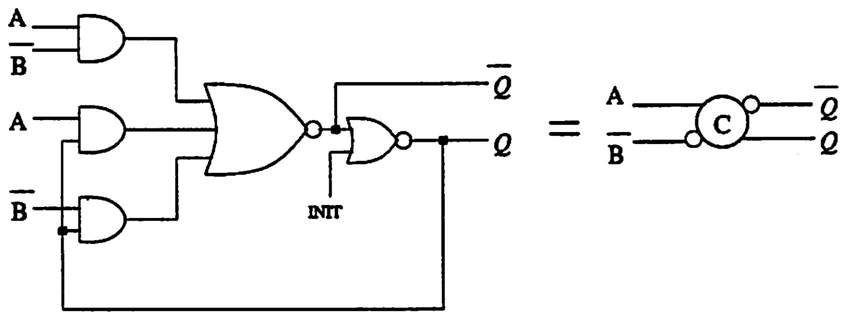


Figure 5.4: Majority function gate implementation of a c-element.

c-element is given in Figure 5.5. It also has the problem of requiring an extra gate to generate the True version of the output which adds to the delay.

A very simple dynamic version[67] of a Muller c-element can be made with the same number of devices as a clocked inverter as shown in Figure 5.6. A version of the dynamic c-element with a clear signal input for initialization is shown in Figure 5.7. It also requires an extra inverter to generate Q from $\bar{Q}$, however the simplicity of the first gate makes the speed high. The downside to using a dynamic c-element would be applications in which the user exploits the self-timed behavior of the system to compensate for delays that may not be defined ahead of time. An example is an I/O function. The system could easily be designed to wait for an sample strobe signal before proceeding with the execution of the signal processing program. If the interrupt was delayed by seconds, the dynamic c-element would not be adequate. However, given the number of applications where dynamic circuits are acceptable, the circuit of Figure 5.6 could prove very useful. A variation of the dynamic gate shown in Figure 5.7 uses feedback to latch the initialization state. Versions that initialize to a Reset Q output and a Set Q output are shown in Figures 5.8 and 5.9 respectively.

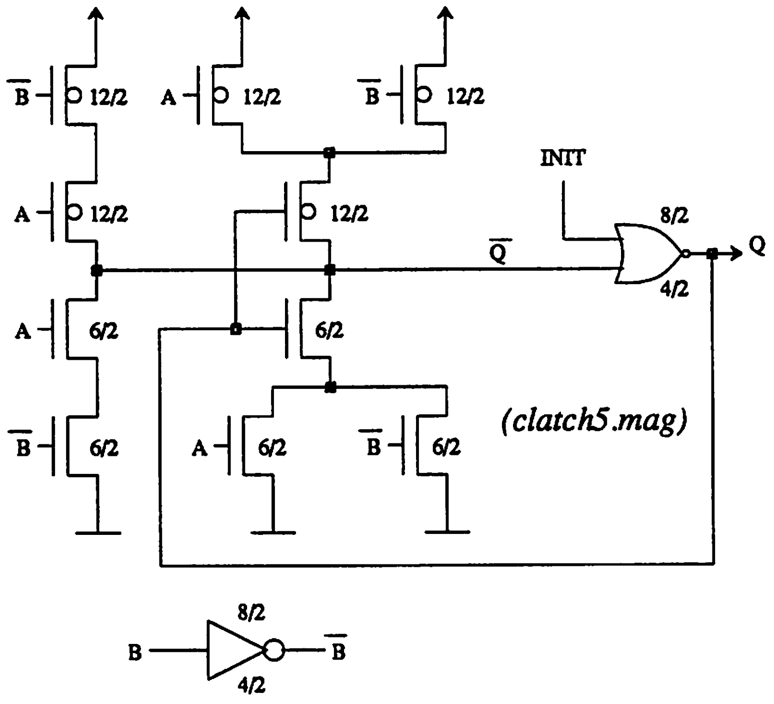


Figure 5.5: CMOS design for majority function c-element.

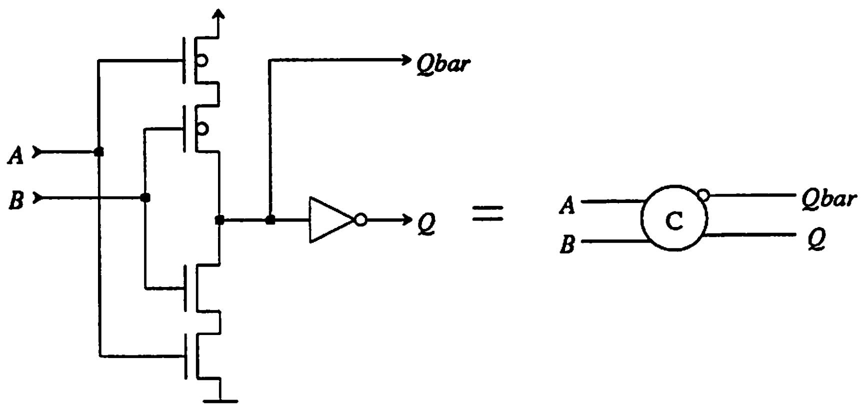


Figure 5.6: CMOS design for dynamic c-element.

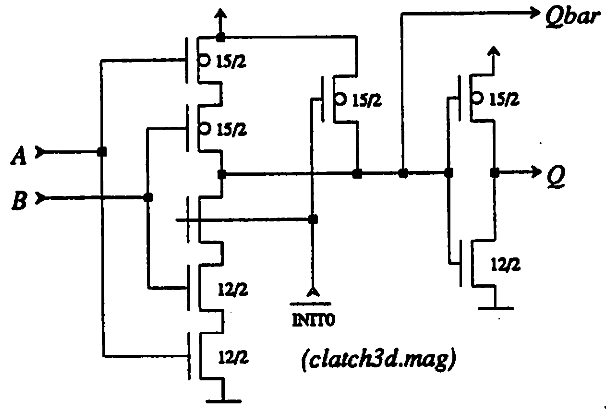


Figure 5.7: CMOS design for dynamic c-element (with clear).

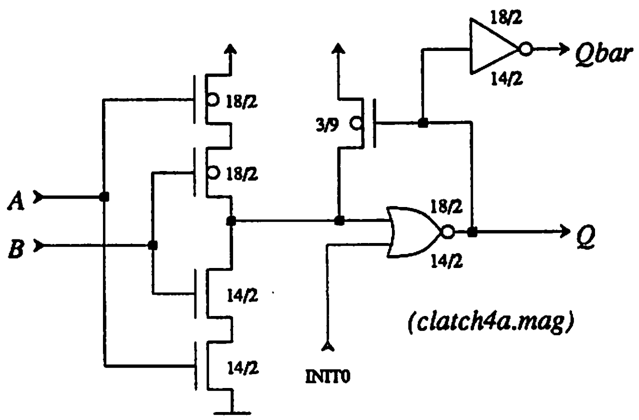


Figure 5.8: CMOS design for dynamic c-element with different initialization scheme. Resets on $INIT0$.

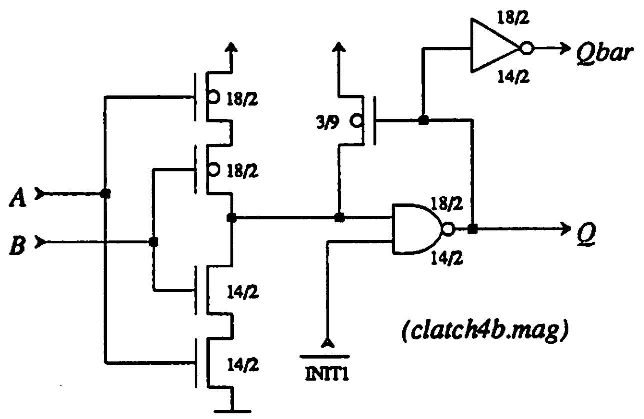


Figure 5.9: CMOS design for dynamic c-element with different initialization scheme. Sets on $\overline{INIT1}$.

A pseudo-NMOS or ratioed design for a bistable c-element[19] is given in Figure 5.10. It also has relatively few devices, so the area is smaller or the device widths can be increased to improve speed without exceeding the area of a more complicated design such as in Figure 5.2. The PMOS devices of the ratioed c-element act as current sources which have to be overcome by the NMOS devices in the circuit for the latch to change state. One disadvantage of the pseudo-NMOS design is that the output logic levels are not as well defined as a true CMOS design. The current sources make the logic LOW level some voltage greater than 0V, degrading the noise margin. It was observed that the current source connection on the Q output device was not really necessary. Figure 5.11 shows a modified ratioed c-element design where the Q output reaches full CMOS levels. The remaining grounded gate PMOS device must remain to provide means for pulling the $\bar{Q}$ node high when A and B go LOW .

Performance Comparison

The choice of a c-element for use in the DSP was somewhat evolutionary, as the naming convention of the clatches in the preceding figures might imply. The study of all of the alternative designs was not done at the beginning of the research because it was not clear at the time what the importance of the speed on overall performance would be. Rather, clatch2 in Figure 5.2 was designed early on and used in several of the test chips for datapath macrocells. When the search for alternative designs was undertaken, a

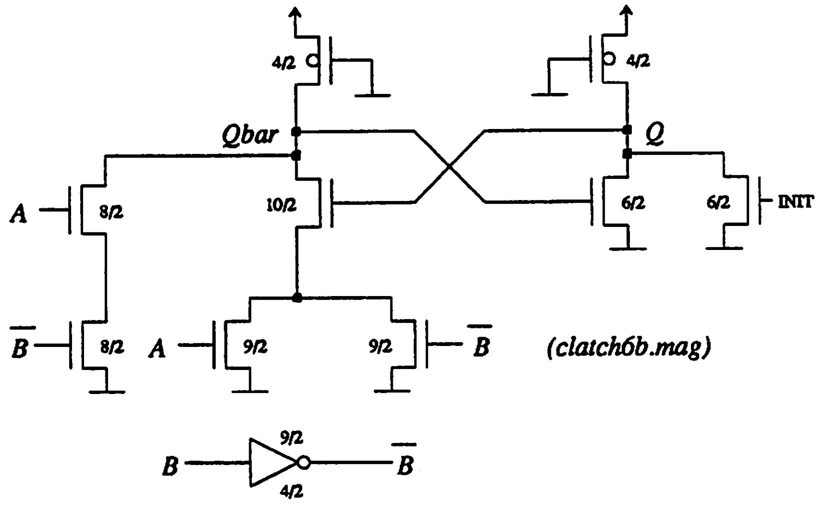


Figure 5.10: CMOS design for ratioed c-element (with clear).

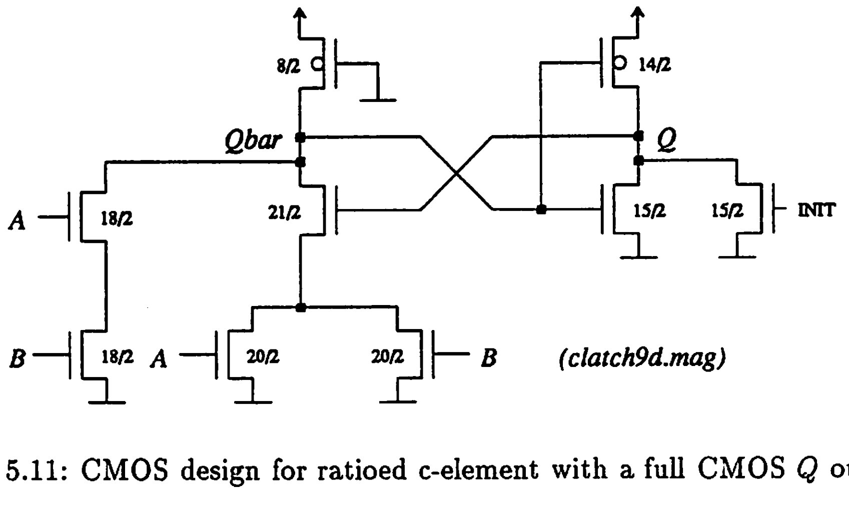


Figure 5.11: CMOS design for ratioed c-element with a full CMOS Q output.

decision was made to fit all new designs into an area that was the same or smaller than that of clatch2. In this way, new designs could be substituted directly for an increase in performance without extensive layout changes in existing circuits. Since some of the alternative designs contain less devices, the devices were grown to enhance speed while using available area.

The objectives of a c-element design as utilized in the handshake circuits presented thus far were also part of the study of new circuit designs. In many latch designs, the *Set* or *Reset* times of the Q and $\bar{Q}$ outputs may be different, usually because one of the two outputs is derived from the other (often via an inverter). This can be exploited to a certain extent. In the basic 4-cycle handshake circuit of Figure 4.12, one input to each c-element must be inverted. The inverted input usually comes from another c-element output. In the design *clatch3d* shown in Figure 5.7, the Q output is derived by inverting the $\bar{Q}$ output. If this is fed back to another c-element inverted input, then another inverter would be necessary. The double inversion reduces the efficiency of the handshake circuit and can be removed by re-drawing the circuit diagram as shown in Figure 5.12. In the

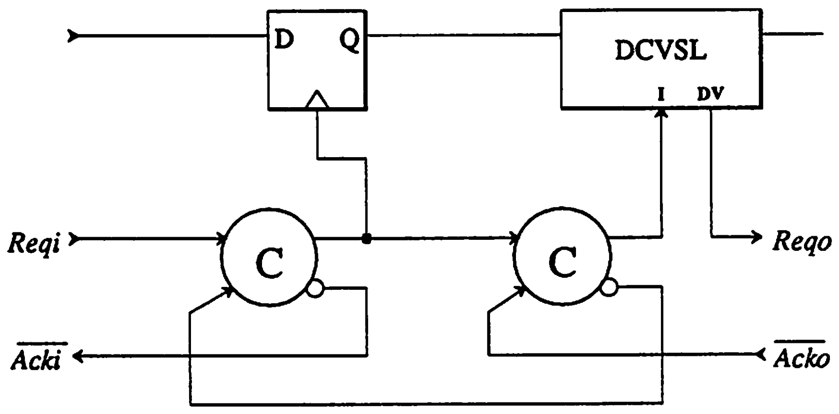


Figure 5.12: Diagram of 4-cycle handshake circuit that exploits both the True and Complement outputs of the c-elements.

latter designs of c-elements presented above, the $\bar{Q}$ output is faster. Because Q and $\bar{Q}$ of the c-elements are not in perfect sync, there is a potential for a timing error in the handshaking circuit when using the two outputs. The potential however is nil in real circuits. The handshake signals $\overline{Acki}$ and $\overline{Acko}$ originating from the $\bar{Q}$ output of the c-elements are fed back to other c-elements. The *difference* in time between Q and $\bar{Q}$ must therefore be less than the *Set* or *Reset* time of the preceding c-element Q output to prevent an error. As shown in the c-element performance table below, that particular circuit constraint is quite easy to meet.

Table IV shows the performance for the different c-element designs discussed thus far. Some notes about this table: All of the c-elements have the ability to be cleared or reset for initialization. The times were taken from simulations using the SPICE simulator The MAGIC extractor (style "ext2spice=1.0") and the program EXT2SPICE were used to get device sizes and parasitic capacitances from the actual layouts. A capacitive load of $0.05pF$ was added to both the Q and $\bar{Q}$ outputs in the spice input file and rise and fall times of $1nsec$ were used in the input pulse generators. Spice model cards were obtained from measured parameters of an actual fabrication run. The MOSIS 2μ, Nwell VTI[2] run parameters (run M8CC) were used.

The overhead for the handshaking circuit can be analyzed by adding the appropriate delays to the STG of Figure 4.5c. Where there is a split in the arcs of the graph, the *maximum* time of any path to reach a point further in the graph must be used since it will dominate. Lets look at the time between rising edges of the I signal in a 4-cycle pipeline stage assuming it drives a DCVSL logic block. As might be expected considering the protocol, the total time is the sum of the times for a *Req* and *Ack* signal pair to cycle high and low. However, since the DCVSL block is in series with the *Reqo* signal, the times associated with it must also be added:

$$T_{cycle} = T_{Reqo^+} + T_{Acko^+} + T_{I^-} + T_{Reqo^-} + T_{Acko^-} + T_{I^+} \tag{5.3}$$

Translating these times to c-element delay times gives the following equation:

$$T_{cycle} = T_{compute} + T_{Q^+} + T_{Q^-} + T_{pre-charge} + T_{Q^-} + T_{Q^+} \tag{5.4}$$

where Q^+ and Q^- are used to denote the *Set* and *Reset* times of a c-element, and $T_{compute}$, $T_{pre-charge}$ represent the computation and pre-charge delays of the DCVSL block respectively. The overhead in time per cycle of operation caused by the c-elements in a 4-cycle handshake circuit is therefore

$$T_{overhead} = 2T_{Q^+} + 2T_{Q^-} \tag{5.5}$$

[2]VTI - VLSI Technology, Inc. is one of the MOSIS prototyping facility's vendors for 2μ fab runs.

Table IV C-element Performance Comparison

Version	Area λ^2	T_{SET} *nsec*	T_{RESET} *nsec*	T_{Total} *nsec*	Notes: (SPICE), 0.05pf load
clatch2	5300	Q^+: 3.0 $\bar{Q}^-$: 1.4	Q^-: 1.5 $\bar{Q}^+$: 2.6	8.5	B-input gets inverted in this design. (Cross-coupled NORs)
clatch5	5200	Q^+: 2.5 $\bar{Q}^-$: 1.2	Q^-: 2.1 $\bar{Q}^+$: 1.1	6.9	Time doesn't include inverter for B-input. (Majority function gate)
clatch3c	3600	Q^+: 2.0 $\bar{Q}^-$: 1.3	Q^-: 1.6 $\bar{Q}^+$: 1.1	6.0	(Dynamic gate)
clatch3d	4300	Q^+: 1.3 $\bar{Q}^-$: 0.8	Q^-: 1.4 $\bar{Q}^+$: 1.0	4.5	(Dynamic gate) Larger devices
clatch4a	5200	Q^+: 1.4 $\bar{Q}^-$: 1.9	Q^-: 1.3 $\bar{Q}^+$: 1.8	6.4	(Feedback Dynamic gate) Larger devices
clatch6b	4300	Q^+: 1.5 $\bar{Q}^-$: 0.8	Q^-: 2.1 $\bar{Q}^+$: 1.2	5.6	(Ratioed logic)
clatch8	4600	Q^+: 2.2 $\bar{Q}^-$: 0.8	Q^-: 2.0 $\bar{Q}^+$: 1.3	6.5	(Ratioed - Full CMOS Q Output)
clatch9d	4600	Q^+: 1.2 $\bar{Q}^-$: 0.6	Q^-: 1.6 $\bar{Q}^+$: 1.2	4.6	(Ratioed - Full CMOS Q Output) Larger devices

When the complementary output of the clatch is utilized as in Figure 5.12, Equation 5.5 must be changed to

$$T_{overhead} = T_{Q+} T_{\bar{Q}-} + T_{Q-} + T_{\bar{Q}+} \tag{5.6}$$

The time T_{Total} given in Table IV above is just the sum of the four delays shown for each clatch, and it equals $T_{overhead}$ as defined in Equation 5.6.

Examining Table IV, it can be seen that as the clatch designs were refined from that of *clatch2*, the delay times decreased significantly while the area was not increased. In the more simple designs, larger devices could be used in the given area to gain a speedup. C-elements *clatch3d* and *clatch9d* were the fastest. Because of the way I/O was implemented on the DSP chips, unknown delay times could be encountered in the program execution depending on the outside world connections. Therefore, the static design *clatch9d* shown in Figure 5.11 was chosen. Two, three, and four input versions of the design were used throughout.

In the early stages of the research, it was assumed that the *Set* time of the c-elements was more important than the *Reset* time in determining overall performance. The logic behind this assumption was as follows: Consider a faster stage preceding a slower stage in a self-timed pipeline. The fast stage finishes and raises the *Reqi* of the slow stage. When the slow stage finally finishes, it allows *Acki* to rise, accepting a new sample. The rising edge of *Acki* "does something", by allowing new data into the input register of the stage, while the falling edge does not affect the state of the register. Therefore, it was assumed that the *Set* time was more important. The design and performance of *clatch9d* reflects this. However, more careful analysis revealed that Equation 5.5 and Equation 5.6 are correct. The overall cycle time of the slow stage dominates the throughput of the pipeline and Equation 5.4 gives that time. Thus, in terms of sheer speed performance, the c-element design *clatch3d* should be chosen if the overall system design can tolerate dynamic latches in the handshake circuitry.

2. Placement of Buffers in HS circuits

The output of each c-element in a handshake circuit is usually another handshake signal. The handshake signals really act as the local "clocks" of a stage and therefore, besides driving adjacent handshake circuits, they must be interfaced to the computational blocks. In Figure 5.12, *Acki* is used to clock the input register and the output of the second c-element (I signal) drives the pre-charge devices of a DCVSL block. For an n-bit wide datapath, the register and DCVSL blocks are also n bits wide and they present a significant capacitive load to the handshake signals from which they are driven.

Buffers large enough to handle the capacitive load in the circuits must be added. The location of the buffers also can greatly affect the overall cycle time and performance of the system. A straightforward choice for the location of the buffers is directly in series with the c-element outputs as shown in Figure 5.13. The delay of the buffers just gets lumped in with the c-element delay and the correct operation in terms of timing is guaranteed because the delay of the c-elements does *not* affect the sequence of operations imposed on the handshake circuit by its specification. The efforts to design a fast c-element presented in the last section are somewhat wasted though, since the delay of the large buffer typically exceeds the delay of the clatch itself. Factoring in the buffer delays

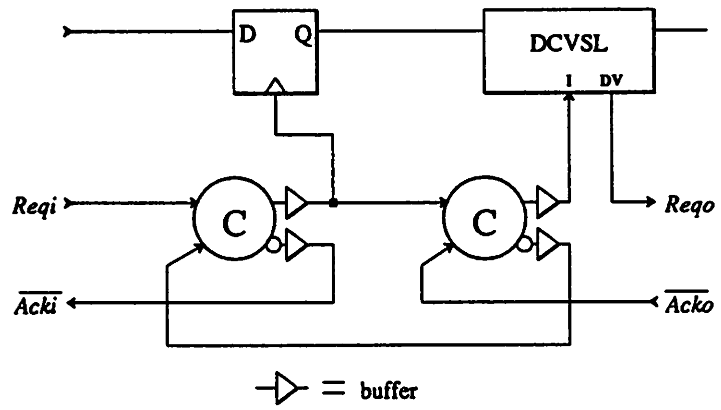


Figure 5.13: 4-cycle handshake circuit with buffers added to c-element outputs.

requires that Equation 5.6 be changed to

$$T_{overhead} = T_{Q^+} T_{\bar{Q}^-} + T_{Q^-} + T_{\bar{Q}^+} + 4T_{buffer} \tag{5.7}$$

assuming that all the buffer delays are equal.

In the case of the rightmost c-element, it makes sense to lump the buffer delay in with the delay of the DCVSL stage and not the c-element since the DCVSL represents an added delay in the handshake signal itself. In the case of the leftmost c-element which generates *Acki*, an assumption must be made to move the buffer to a position which enhances the efficiency. Figure 5.14 shows the new buffering scheme. In this case, the

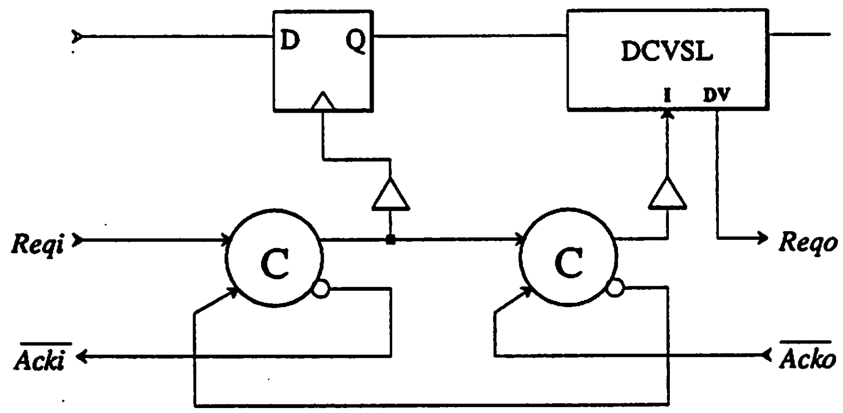


Figure 5.14: 4-cycle handshake circuit with buffers added in a position which enhances the efficiency.

overhead time is

$$T_{overhead} = T_{Q^+} T_{\bar{Q}^-} + T_{Q^-} + T_{\bar{Q}^+} + 2T_{buffer} \tag{5.8}$$

however, the price for the reduced overhead is more sensitivity to the circuit design. By placing the first buffers in a location which in effect can alter the sequence of signal transitions, more care must be taken in controlling delay times. Two possible problems exist. First, when the $Acki^+$ transition occurs, the previous stage is allowed to enter the pre-charge state. The input register however must latch the new data sample before it disappears during pre-charge. Assuming the previous stage contains a buffer on its I signal, the delay of the *Acki* buffer must be $T_{Ackibuf} \leq T_{Q^-} + T_{Ibuf} + T_{pre-charge}$. The pre-charge time allows a fair amount of slack in the mismatch of the *Acki* and I buffer delays. The second constraint is more demanding. The other effect of the *Acki* transition is to enable I to rise and begin the computation for this stage. The outputs of the data register must be settled before this so that the DCVSL logic has valid inputs. With no buffering, this is still an important constraint and it requires that $T_{register} \leq T_{Q^+}$, where T_{Q^+} is the delay of the second c-element. Adding buffers to both the *Acki* and I signals requires the delays of the buffers to match closely so that $T_{Ackibuf} + T_{register} \leq T_{Q^+} + T_{Ibuf}$. Typically, the loading on the I signal is greater than that on *Acki* because of the larger (and greater number of) devices necessary in the DCVSL to do the pre-charging making the constraint manageable. During the design of the DSP chips, no problems were encountered when using the altered buffering scheme. With or without the buffers, the constraint on the settling of the register outputs could be removed simply by employing a register that generates a completion signal as shown in Figure 5.15. While a DCVSL register design is not known, (the pre-charging destroys the necessary memory action required in a register) a completion signal can be generated by using an extra "dummy" register or by comparing the input and output signals of the register when the clock goes high. For the DSP design however, this was not done and the matching was exploited.

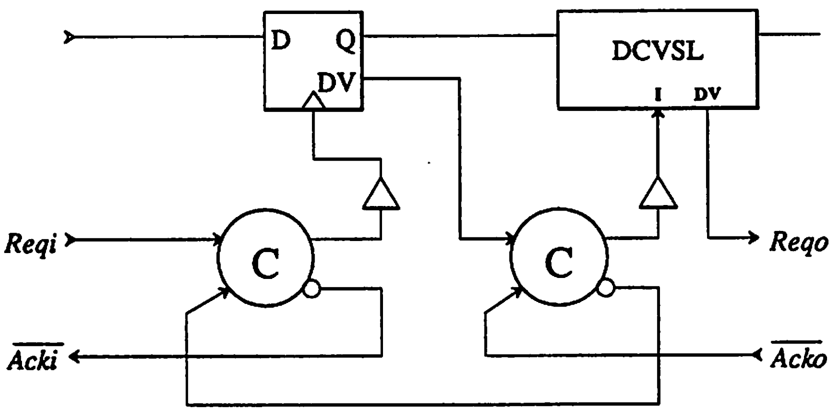


Figure 5.15: 4-cycle handshake circuit interfaced to a DCVSL stage and register which generates a completion signal.

The discussion above points out several difficulties in applying the theory for handshake circuit synthesis to actual circuit implementations. For one, it may be unclear at times how the handshake signals map to the actual circuit signals. The use of *Acki* to clock the input register is not directed by anything in the handshaking circuit specification although it makes sense. Doing so however, adds the constraint that the output of the register settles before the DCVSL begins computation. Second, while the need for buffering signals can be addressed by basically making higher output drive c-elements, the added delay can reduce efficiency. The advantage of the self-timed approach is to make the circuit adapt (slow down) to the added buffers. The inclination of the hardware designer however, is to move the buffers so that they don't add any unnecessary overhead. This requires more careful analysis of circuit delays which can increase the risk of error-free operation.

Chapter 6

Self-Timed Macrocell Design

To construct a self-timed digital signal processor, a collection of self-timed macrocells that make up the datapath must be interconnected via the use of handshaking. In this chapter, designs for the datapath computational elements used in the DSP chip are described. Each cell has a generic interface to surrounding circuitry; they all look like a simple block that accepts a compute signal input and supplies a **DV** completion signal output. This fits the self-timed system paradigm as presented in previous chapters. Two of the datapath cells are a cascade of DCVSL gates. The third more complicated cell is actually a small self-timed sub-system in itself. The cell, an iterative multiplier, demonstrates the hierarchical approach that can be taken even in the design of macrocells. The adaption of standard RAM and ROM designs to allow self-timed operation is discussed last. In the figures below, italicized cell names in the form *name.mag* denote the MAGIC layout editor cell names for the cell in the database of the chips described. Device sizes are given in λ. Further design details and schematics for cells not shown in this chapter can be found in Appendix C.

1. Barrel Shifter

In a DSP datapath, the function of shifting is commonly required. In a control operation, a shifter allows easy masking of certain bits in a word for identification of functions. In a signal processing operation, the shifter provides fast multiplications/divisions by powers of two. Often, when the time for a full multiplication is undesirable, bit-parallel shift and adds are performed instead. Using canonical signed digits representation of coefficients allows efficient and fast multiplications in a shorter time[72]. The barrel shifter allows shifting in any amount between 0 and n places. The design described below is a $0 - 15$ place left shifter. It accepts a 16-bit word input (typically in $q.15$ format) and supplies a 32-bit word output (typically in $q.30$ format).

A block diagram of the shifter is shown in Figure 6.1. It is a logarithmic style shifter which is essentially the cascade of four simple shifters. The first one shifts 0,1 place, the second 0,2 places, the third 0,4 places, and the fourth 0,8 places. Each of the simple shifters is implemented with a 2:1 multiplexer (MUX) and a routing channel. The multiplexer inputs are therefore bit_i and bit_{i-n} of the preceding row, where $n = 1, 2, 4, 8$ depending on which of the four rows it resides. Cascading the four shifters allows shifting by any amount between 0 and $2^4 - 1$ places. A 4-bit binary coded control word selects the shift amount. Due to the logarithmic style of this type of shifter, no decoding is necessary on the control word input. Each bit of the control word just becomes the MUX control signal of one row. A fifth control input allows selectable sign-extension on the output data. When asserted, the most significant bit (MSB) or sign bit of the input word is fed to bits 31 to $31 - (15 - n)$ of the output, where n is the shift amount. The sign-extension (SE) control allows the user to select either logical shifts (no sign-extension) or normal arithmetic shifts. When a word is shifted left, 0 bits fill the least significant bits (LSBs).

The DCVSL circuit for the 2:1 MUX is shown in Figure 6.2. In the implementation of this macrocell, several simplifications were made for efficiency. For one, since each of the four rows in the barrel shifter consist of a DCVSL gate and routing only, they are directly connected as would be done with domino logic gates. Therefore, the data signal ripples through each row, consecutively firing the MUX gates. A 16-bit D-register is incorporated into the macrocell to hold data during pre-charge. As Figure 6.2 shows, the NMOS device at the bottom of the NMOS tree in the 2:1 MUX was eliminated (see Figure 3.2 for the general structure). This could be accomplished by adding two gates to the control input for each row of MUXes. Since the MUX circuit cannot have valid complimentary outputs unless either *c* or *cbar* is HIGH , the extra NMOS device at the bottom of the tree can be eliminated by feeding the MUXes with *c*.I and *cbar*.I. In this way, the tree can only be discharged during the evaluate stage as desired. The modification increased the speed of the shifter by eliminating an extra series device in each gate (128 in all) at the expense of the two AND gates per row. The circuit to buffer each control input is shown in Figure 6.3 and it shows the extra gates used (actually NOR gates) to eliminate the bottom

SELF_TIMED SHIFTER BLOCK DIAGRAM

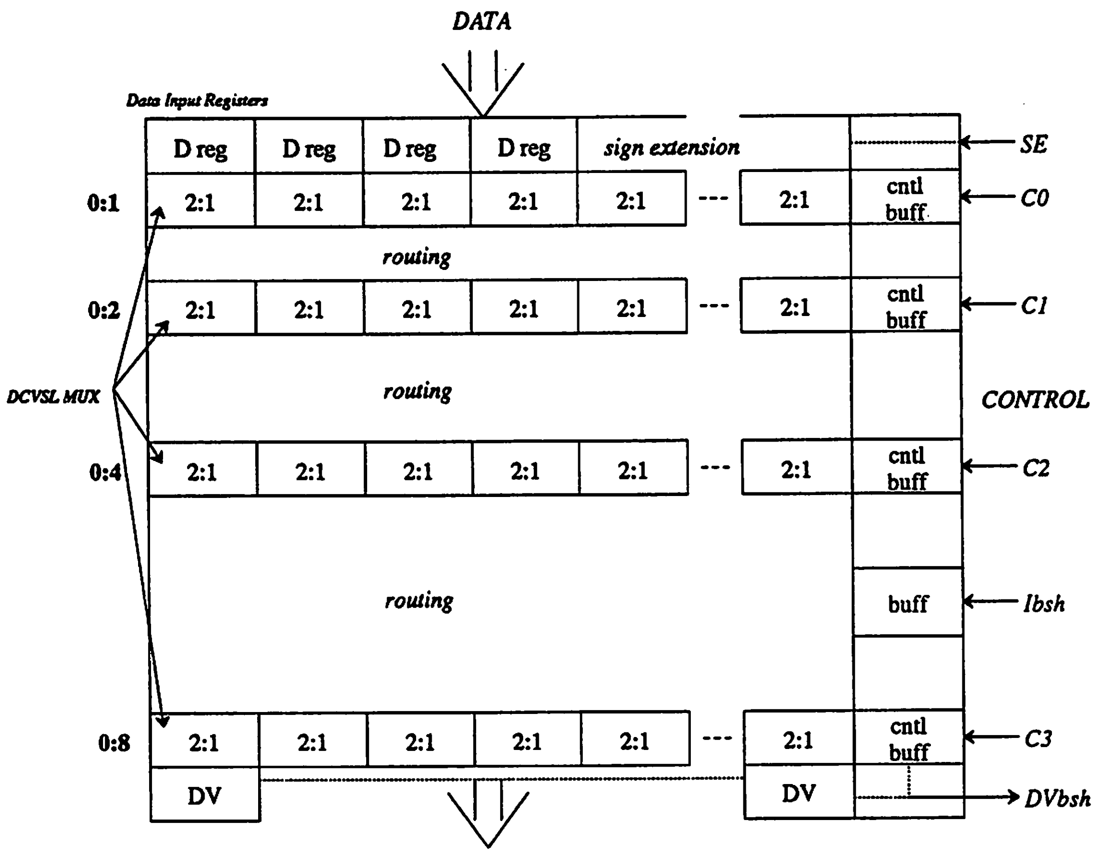


Figure 6.1: Block diagram of the self-timed barrel shifter.

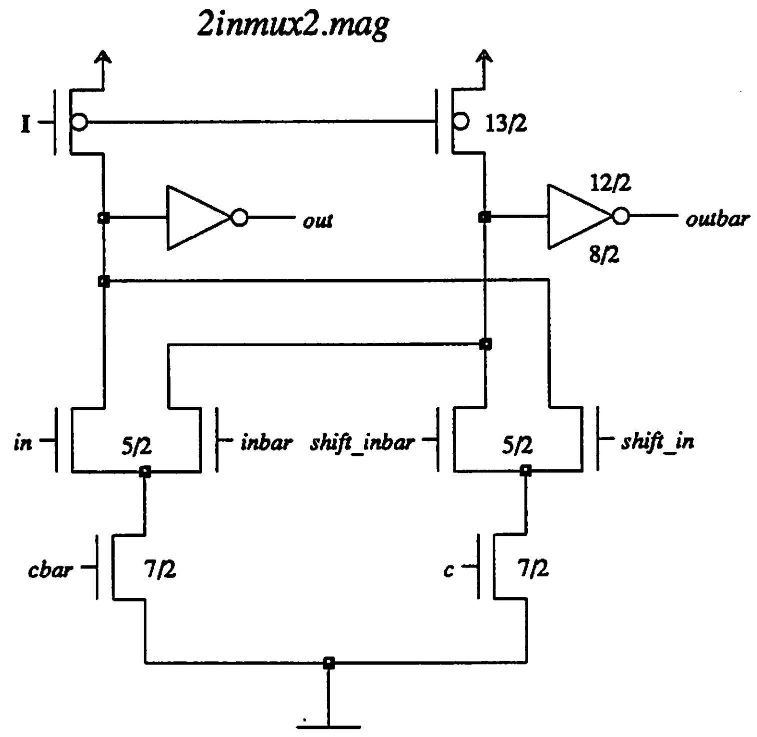


Figure 6.2: DCVSL 2-input MUX used in barrel shifter.

NMOS device of the 2-input MUX. The D flip-flop used in the data input register is shown in Figure 6.4.

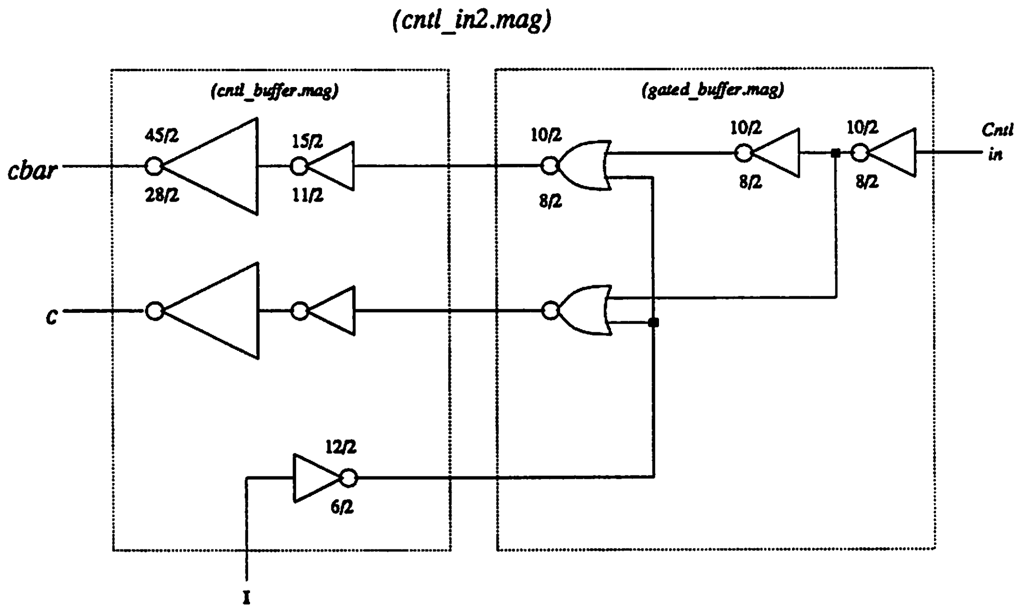


Figure 6.3: Control Input buffer for barrel shifter.

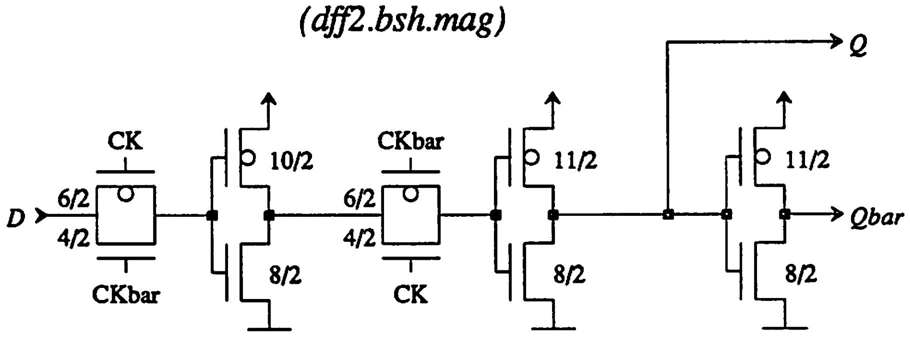


Figure 6.4: D flip-flop used in the input register of the barrel shifter.

Finally, the symmetry of the shifter structure allowed for placing completion signal circuitry (an OR gate tied to *out* and *outbar* of the last shifter) only on several key gates of the last row of DCVSL MUXes. This technique depends on the matching of delays through identical logic gates in the shifter and it deserves some extra comments. In a $n-bit$ wide macrocell that strictly follows the self-timed paradigm, each bit provides a completion signal. To generate the completion signal for the entire macrocell, the n completion signals from the bits should be applied to an n-input c-element. The added hardware required and the associated delay and overhead of this type of completion circuit is significant and surely undesirable. There are several reasons however that make it unnecessary in most applications. Firstly, while data computation in a combinatorial network may take place in a collection of series and parallel connected gates, the pre-charge operation takes place in all of the gates simultaneously. Therefore, the pre-charge time of a large DCVSL block is constant for all bits to within the local delay time of the pre-charge signal. Since the individual bit **DV** signals may rise at different times but will fall during pre-charge at the same time, an n-input AND gate can replace the n-input c-element to generate the final completion signal. Secondly, some macrocells are very regular in their structure and the completion time of each bit is not data-dependent. For such a cell, the completion signal for a single bit is valid for the entire cell to within the delay matching of the circuitry for each bit. As with any integrated circuit, the local matching of circuit parameters is excellent and circuitry depending on the matching is low-risk. In the barrel shifter, the delay time is data-independent although the worst case delay for a particular bit depends on the amount of the shifting. For the shifter described above only several out of the 32 bits were used to generate a **DV** signal, exploiting the matching characteristics. The block diagram in Figure 6.1 illustrates this by having only two "DV" cells in the last row. The signal outputs from those gates are AND 'd together to form the final completion signal in the control slice on the right.

2. ALU

A 32-bit ALU was designed for the datapath so that full precision could be maintained in the accumulator values. The ALU design however, is bit-sliced so that any multiple of four bits can be constructed. The functions supported by the ALU include addition, subtraction, AND , OR , XOR , NOT logic functions as well as the ability to zero either input for clearing/loading the accumulator. Figure 6.5 shows a block diagram of the ALU and a single bit-slice. The accumulator is built into the ALU although it is clocked by a separate input signal. By incorporating the accumulator into the ALU, more efficient routing is achieved over having it reside elsewhere as a separate block. For the same reason, the A-input register is also built into the ALU (the B-input register is the accumulator).

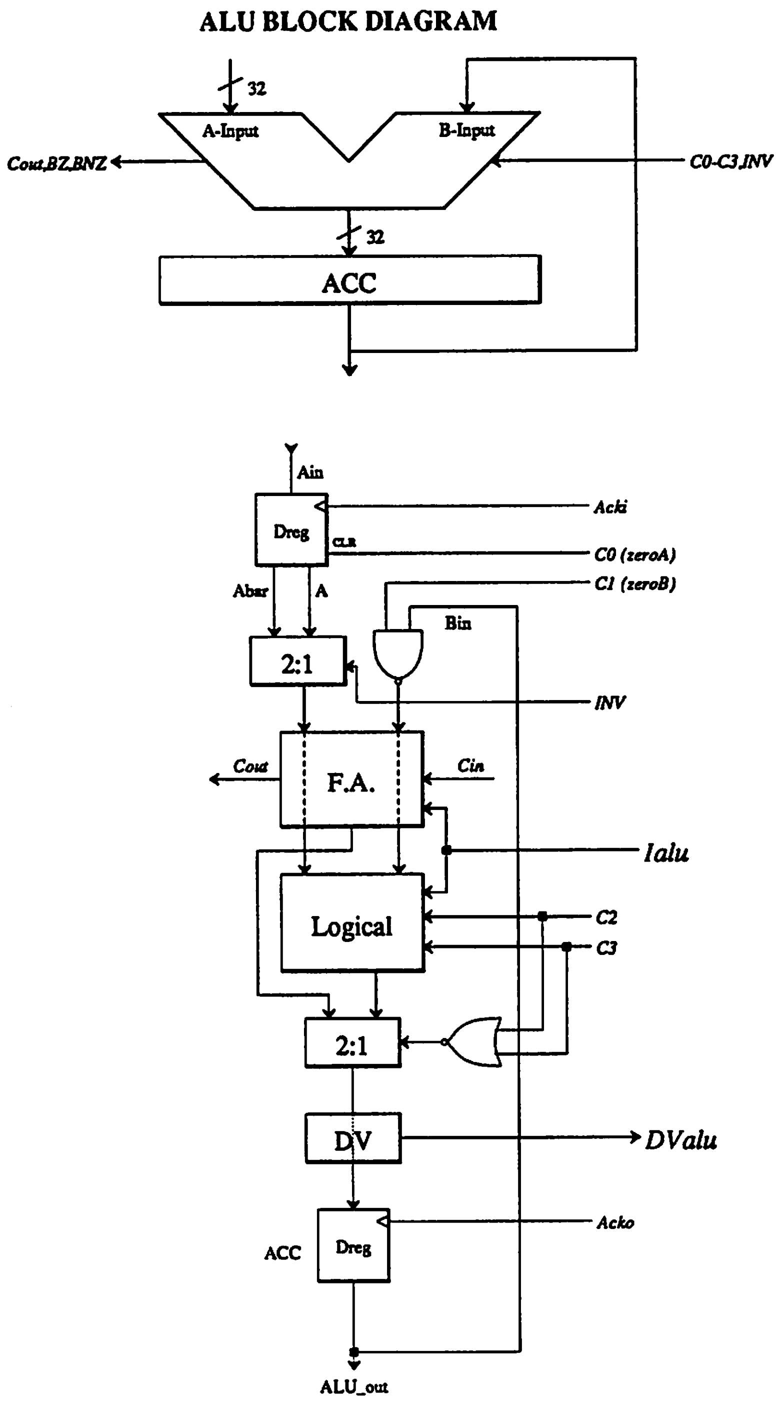


Figure 6.5: Block diagram of the self-timed ALU.

The logical functions and the addition are performed by separate DCVSL gates, the outputs of which are fed to a 2:1 MUX. Depending on the desired ALU function, one of the two gate outputs is passed to the accumulator. Since the completion time for addition is dependent on the length of the carry propagation, all bits are examined to form the ALU data valid completion signal. In this way, the completion time of the ALU will be data dependent in addition or subtraction modes while it is constant during the performance of logical functions. The ALU also provides a Carry output and a "Branch on Zero" output that can be used to control branching in a signal processor.

The full adder was designed as two DCVSL gates, one for sum and the other for carry. The *ntree* program described in Chapter 3 was used. Below, the input and output files for each gate are shown:

Input file for SUM

```
# FULL ADDER SUM stage (3-way xor circuit)
#
(sum  (xor 1 2 3))
#
```

Output file for SUM

```
        NMOS Tree for (sum  )
*
* Logic Expression: (xor 1 2 3)
*
* This file generated by ntree on Sun Jul  5 22:29:06 1987
*
*  NODE ASSIGNMENTS:
*  GND  =  0      Vdd  =  100
*  Pbulk = 102   Nbulk = 101
*  (Complement of )Input Number 1  is node (11) 1
*  (Complement of )Input Number 2  is node (12) 2
*  (Complement of )Input Number 3  is node (13) 3
*  F_OUT = 21   F_BAR_OUT = 20
*
*    D   G   S   B
m1   24  12   0  101  NMOS
m2   25  13  24  101  NMOS
m3   20  11  25  101  NMOS
m4   21   1  25  101  NMOS
m5   28   3  24  101  NMOS
m6   21  11  28  101  NMOS
m7   20   1  28  101  NMOS
m8   29   2   0  101  NMOS
m9   28  13  29  101  NMOS
m10  25   3  29  101  NMOS
***end***
```

Input file for CARRY

```
#
# FULL ADDER carry stage (majority gate)
#
(carry_out 100 (or (and 1 2) (and 1 3) (and 2 3)))
#
```

Output file for CARRY

```
        NMOS Tree for (carry_out 100 )
*
* Logic Expression: (or (and 1 2) (and 1 3) (and 2 3))
*
* This file generated by ntree on Tue Mar 10 11:24:16 1987
*
*  NODE ASSIGNMENTS:
*  GND  = 0       Vdd  =  100
*  Pbulk = 102    Nbulk = 101
*  (Complement of )Input Number 1  is node (11) 1
*  (Complement of )Input Number 2  is node (12) 2
*  (Complement of )Input Number 3  is node (13) 3
*  F_OUT = 21    F_BAR_OUT = 20
*
*      D   G   S   B
m1    24  11   0  101  NMOS
m2    20  12  24  101  NMOS
m3    26   2  24  101  NMOS
m4    20  13  26  101  NMOS
m5    21   3  26  101  NMOS
m6    28   1   0  101  NMOS
m7    26  12  28  101  NMOS
m8    21   2  28  101  NMOS
***end***
```

The schematics for the full adder gates are shown in Figure 6.6. One important note about the adder is that the completion time is dependent on the permutation of the inputs for the carry gate. The logical equation for the carry gate is

$$C_{out} = AB + BC_{in} + AC_{in} \tag{6.1}$$

While the logical result is independent of the ordering of the inputs, the firing of the DCVSL implementation of the gate is not. The completion time for a full adder depends on the number of stages through which the carry must propagate. If either the A or B inputs (but not both) are high, then the C_{out} result depends on the C_{in} input, hence the carry propagation. If, however both the A and B inputs are high, then a carry is generated for that bit regardless of the C_{in} input. As drawn in Figure 6.6, the carry gate output becomes valid if both A and B are high as desired. If a different ordering was used, (i.e., C_{in} placed on one of the lower transistor pairs) then the gate output will not become valid until C_{in} is valid. Because each bit waits for the previous bit's carry result, the completion time for the adder is the *worst-case*, which is the carry propagation time for the full 32 bits. Some of the early adder test chips built contained the different ordering and showed a constant worst-case completion time. The technique could be used if data-dependency is not desired in an adder.

The logical functions are performed in a single complex DCVSL gate. It demonstrates the advantage of being able to perform a collection of logical operations with a single gate in DCVSL. The *ntree* input file for the logical block is:

Input file for LOGICAL

```
# ALU logic circuits
#  Complementation is accomplished by muxes on input
#  choosing between 1, 1*, 2, 2*
#  Inputs: 1,2
#  Control: 3,4 -> binary coded:
# 0  not-used
# 1  or
# 2  and
# 3  xor
#
#
(logic outputs
(or
    (and 3 (not 4) (or 1 2))
    (and (not 3) 4 (and 1 2))
    (and 3 4 (xor 1  2))
)
)
#
```

The schematic for the DCVSL gate for performing the logical operations is shown in Figure 6.7. The 2:1 MUX circuits are very similar to the version used in the barrel shifter and their schematics along with the schematics of miscellaneous other cells used in the ALU can be found in the appendix.

Completion Signal Generation

In the ALU, the completion signal generation must use all of the bits of the output word because of the data dependency of the addition operation. A 32-bit AND gate is implemented to do this but in the interests of speed and keeping the architecture in bit-sliced form, the large gate is implemented as a tree of 4-input gates. Figure 6.8 shows the schematic for the Data Valid signal generation circuitry which spans each 4 bits in the ALU. (This cell design constrains the wordlength of the ALU to be a multiple of 4 bits) The top row of 4 NOR gates simply forms the $\overline{DV}$ signal for each bit from the true and complement outputs of the last DCVSL stage in the ALU gates. Those four signals are fed to a 4-input NOR gate which makes up the first level in the tree structure. The NOR output is "dvA" which is a completion signal for the four bits spanned by each *DV.mag*

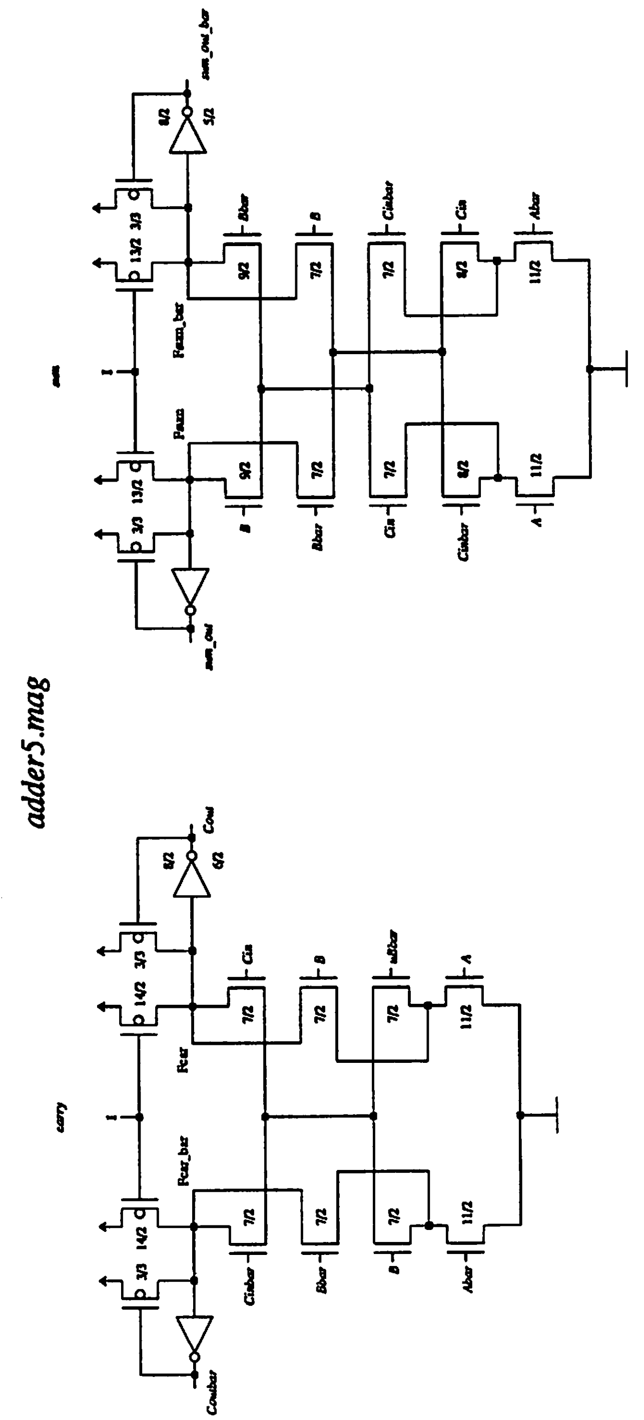


Figure 6.6: Schematics for DCVSL full adder used in the ALU.

logic2.mag

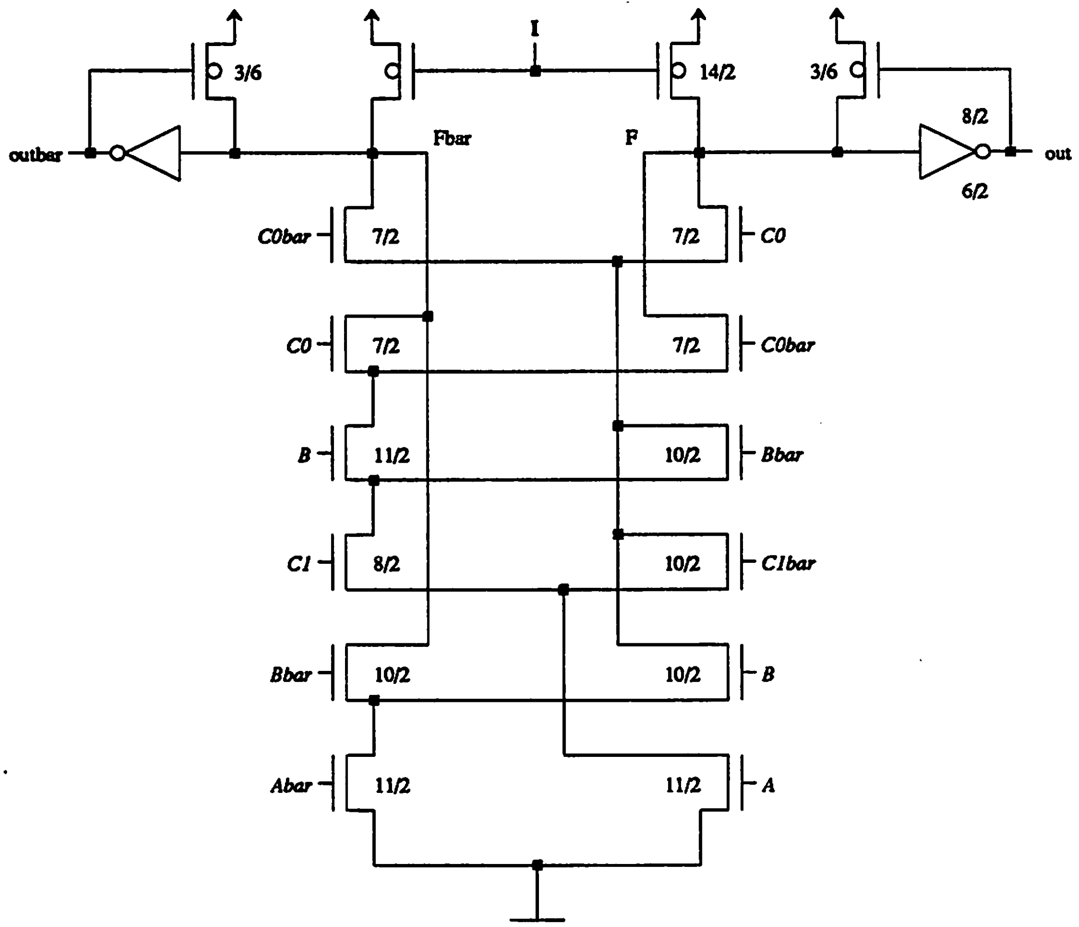


Figure 6.7: Schematic for the DCVSL gate performing logical functions in the ALU.

cell. At the bottom are devices and metal lines which are used to make up a NAND gate for the dvA signals. The "X"s show optional connections which are made depending on the wordlength of the ALU. The dvB and dvC busses are meant to each span 16 bits, that which causes the NAND gate to have four inputs. The maximum size of the ALU is limited to 32 bits. The dvB and dvC signals are sent to a 2-input NOR gate in the control slice of the ALU to generate the final **DV** signal.

The following table provides a complete description of the ALU functionality in terms of its control inputs. Note that the B-input of the ALU is the accumulator output. Also, since the functions of control bits **C0** and **C1** are just to zero the A-input and B-input of the ALU respectively, there is some redundancy in the mapping of control bits versus function in the table. The symbol $\sim$ denotes a one's complement inversion while the symbol $-$ denotes a full two's complement inversion (negation of a signed quantity). $\wedge$, $\vee$, $\oplus$ mean AND , OR , and XOR .

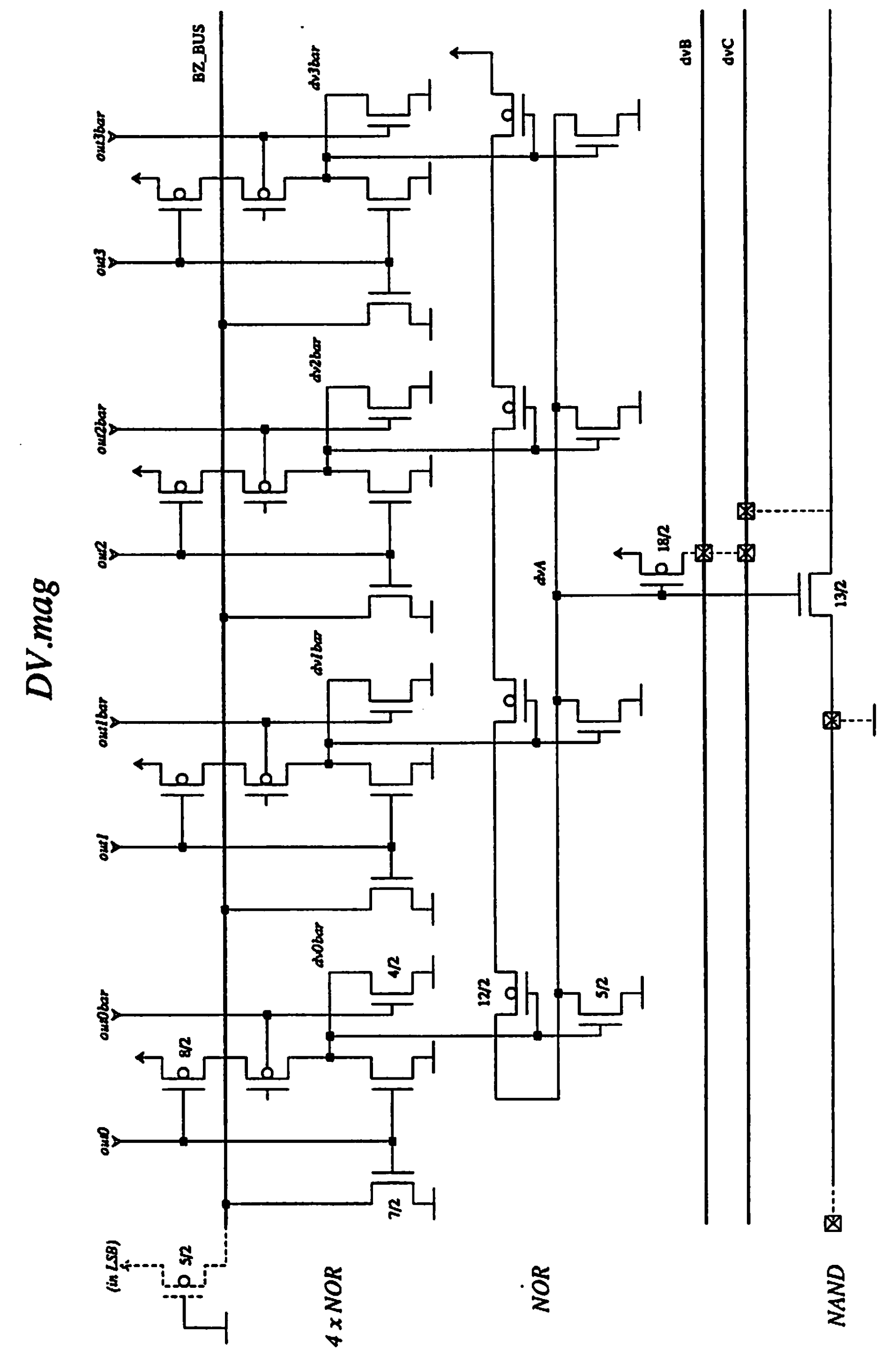


Figure 6.8: Schematic for the Data Valid signal generation circuitry in the ALU.

ALU Function vs. Control Inputs

INV	C0	C1	C2	C3	Function
0	0	0	0	0	$A+B$
0	0	0	0	1	$A \wedge B$
0	0	0	1	0	$A \vee B$
0	0	0	1	1	$A \oplus B$
0	0	1	0	0	A
0	0	1	0	1	0
0	0	1	1	0	A
0	0	1	1	1	A
0	1	0	0	0	B
0	1	0	0	1	0
0	1	0	1	0	B
0	1	0	1	1	B
0	1	1	0	0	0
0	1	1	0	1	0
0	1	1	1	0	0
0	1	1	1	1	0
1	0	0	0	0	$A-B$
1	0	0	0	1	$(\sim A) \wedge B$
1	0	0	1	0	$(\sim A) \vee B$
1	0	0	1	1	$(\sim A) \oplus B$
1	0	1	0	0	$-A$
1	0	1	0	1	0
1	0	1	1	0	$\sim A$
1	0	1	1	1	$\sim A$
1	1	0	0	0	B
1	1	0	0	1	B
1	1	0	1	0	B
1	1	0	1	1	B
1	1	1	0	0	0
1	1	1	0	1	0
1	1	1	1	0	0
1	1	1	1	1	0

3. Iterative Multiplier

For a multiplier, the repetitive nature of the computation allows a single piece of hardware (Booth decoder and carry-save adder) to be used repeatedly to form each partial product [78]. The iterations are controlled by two additional internal handshaking circuits.

The multiplier block diagram shown in Figure 6.9 illustrates the self-timed system model on a smaller scale. The multiplication operation is broken down into three stages: When new operands X, Y are ready, the partial products are each calculated by forming

ITERATIVE MULTIPLIER BLOCK DIAGRAM

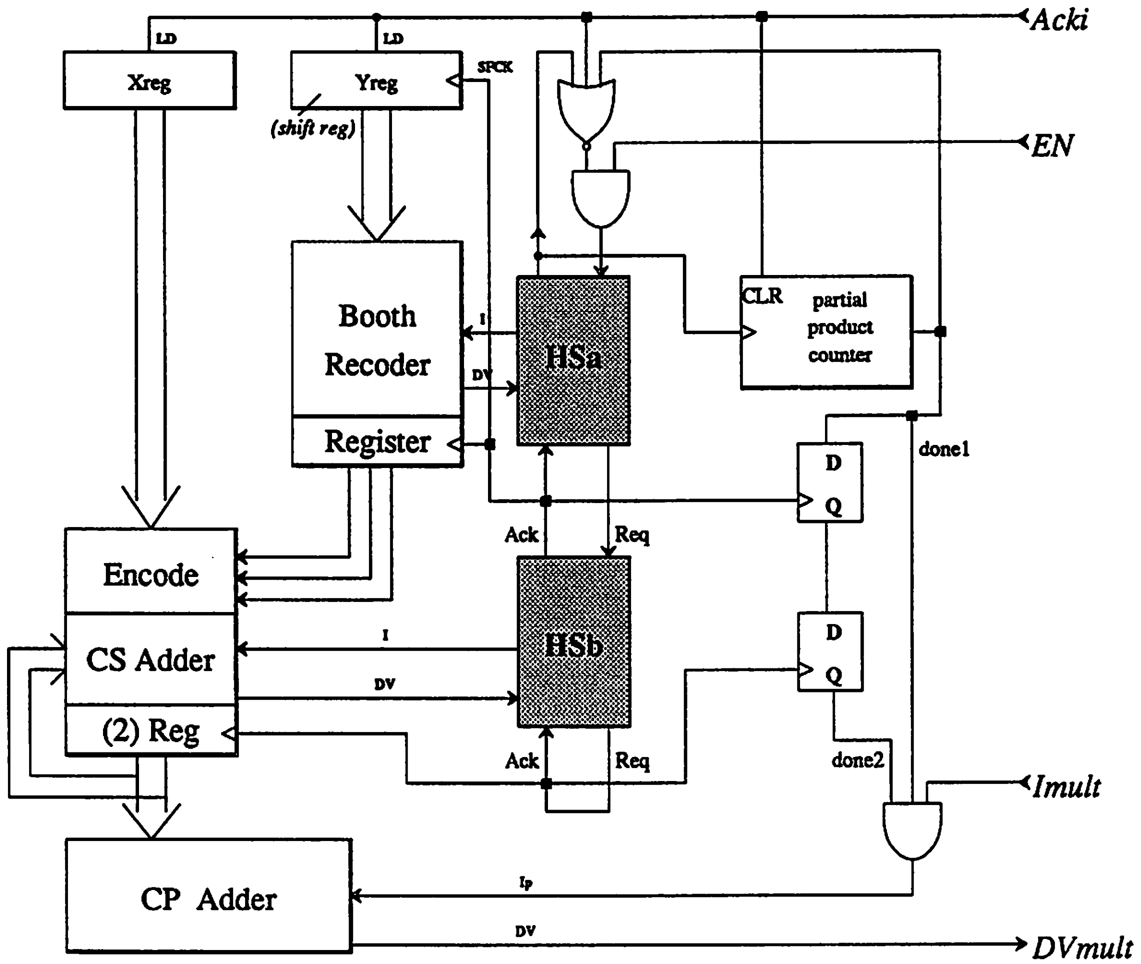


Figure 6.9: Block Diagram of the iterative self-timed multiplier chip.

a Booth coefficient and performing a carry save (CS) addition. Then a final product is formed by assimilating the carries in the carry-propagate (CP) adder. The three stages are all self-timed sub-blocks. Remembering that to the external world, the multiplier just appears as another DCVSL block, it accepts the standard **I** signal and generates a **DV** signal. The **I** signal enables the final CP adder from which **DV** is generated. Both *Imult* and and *Acki* as shown on the block diagram are assumed to originate from some external handshake circuit that controls the multiplier. *Acki* latches new operands as usual. Between the time that the new operands are latched and the final addition is performed however, the **I** signal for the CP adder is held low so that the internal self-timed system for calculating the partial products can do its work. For each partial product, **HSa** enables the Booth Recoder to compute a new Booth coefficient. Upon completion of this task, **HSa** informs **HSb** that it is ready and **HSb** enables the next stage to perform the Booth encoding of the multiplicand and the carry save addition. The result of the carry save addition is stored in two sets of registers (one for *carry* and one for *sum*) for the next partial product. All of the handshake circuits are basically the 4-cycle type that was shown in Figure 4.12.

A partial product counter indicates when all the partial products are calculated and this allows the final adder to operate. With the Booth algorithm, $m = n/2$ partial products are required for a n-bit multiplier (n must be a multiple of two). The count that generates a "done" signal in the partial product counter is programmable via the tiling of the counter cells. The counter is 5 bits wide allowing support for up to $2^5 - 1$ partial products. Since the stages are in effect pipelined, note that it is necessary to add a two-stage delay line between the partial product counter "done1" output and the "done2" signal feeding the AND gate controlling *Imult*. This demonstrates how self-timed systems naturally follow a so-called Data Stationary architecture. Because the actual time of data transfers is not controlled, moving the control signals along with the data (via the same handshake interface circuits) ensures proper operation. More about this type of architecture for a full processor will be explained in the next chapter.

The internal system for calculating partial products is really just a free running self-timed pipeline. By connecting the *Reqo* to the *Acko* of **HSb**, and the *Reqi* to the $\overline{Acki}$ of **HSa**, the pipeline free runs. Some means of controlling it is required however for initialization of new multiplications and for assuring that data fed to the CP adder is valid. When the partial product counter raises its done signal output, the NOR gate above **HSa** disables new requests so that the pipeline stops. The multiplier *Acki* signal is used for initialization. Upon the assertion of the *Acki* signal, the partial product counter is cleared (as well as the two D flip-flops that delay the counter done signal). This in turn lowers the counter done output. The *Acki* signal itself however is also fed to the NOR gate so that the partial product calculations do not begin until it goes LOW . This is necessary because the *Acki* signal holds the partial product counter cleared until it is lowered. Another enable signal (EN) can also be used to prevent the internal pipeline from operating. This is useful when the multiplier must be fully disabled in spite of the state of the *Acki* signal which can cycle if it also controls other blocks. A more detailed explanation will be presented in the next chapter about this.

The core of the multiplier is bit-sliced, although multiplicand wordlengths must be a multiple of four due to the way completion signal circuitry is spread across four bits in the layout. For the DSP chip described in Chapter 7, a *single precision* 16-bit product is generated for 16-bit multiplier and multiplicand inputs.

3.1 Booth Algorithm

The Booth algorithm for multiplication takes an n-bit 2's complement multiplier $Y = (y_{n-1}, \cdots, y_1, y_0)$ (MSB $= y_{n-1}$) and an j-bit 2's complement multiplicand $X = (x_{j-1}, \cdots, x_1, x_0)$ and generates a product (of up to $n + j$ bits) in $m = n/2$ cycles where each cycle represents the calculation of a single partial product. The multiplier is recoded into a radix-4 multiplier $z = y$ by a triplet scanning method using the modified Booth algorithm:

$$z = (z_{m-1}, \cdots, z_1, z_0), \quad z \in \{-2, -1, 0, 1, 2\} \tag{6.2}$$

where

$$z_i = y_{2i+1} + y_{2i} - 2y_{2i-1} \quad for \ i = 0, 1, \cdots, m-1. \tag{6.3}$$

and $y_{-1} = 0$. The product is obtained by the following recursion:

$$P(k+1) = \frac{1}{4}(P(k) + X \cdot z_k) \quad k = 0, 1, \cdots, m-1 \tag{6.4}$$

where $P(k)$ is the k^{th} partial product and $P(0) = 0$. Carry-save adders save the partial products so each one is represented by a pair of bit-vectors $< C(k), S(k) >$ where $C(k)$ is the carry word and $S(k)$ is the sum word.

The register which holds the multiplier Y loads in parallel fashion at the start of a new multiplication. It then shifts two bits to the right per partial product. The three LSBs are fed to the Booth Recoder which implements Equation 6.3. The actual implementation of Equation 6.3 depends on the specific hardware which makes use of the Booth coefficient. The three-bit window input to the recoding circuitry defines a coefficient $z_i \in \{-2, -1, 0, 1, 2\}$. In this design, the coefficient is expressed in the form of a new three bit signal vector $B_i = [1x, 2x, inv]$. Each bit denotes the following:

1x: Pass multiplicand unshifted

2x: Pass multiplicand shifted (left) by 1 bit

inv: Invert multiplicand

If **1x** and **2x** are both low, then the multiplicand is zeroed. The description above for the bit functions is actually what takes place in the Booth encoding of the multiplicand in each bit slice. The logic to perform this is shown in Figure 6.10. The recoding logic which maps the three bit multiplier windows into the bit vector B_i is shown in Figure 6.11. It resides in a control slice adjacent to the multiplier bit slices.

3.2 Multiplier Cell Design

The layout for the iterative multiplier consists of a MSB slice, a series of bit slices, and a control slice. The control slice contains all of the handshaking circuits, the partial product timer, and the Booth recoding circuitry. A block diagram of the bit slices (minus the carry propagate adder) is shown in Figure 6.12. The slice covers 2 bits since the shift register has even and odd cell types due to the shift-by-two nature of Y input register. A normal register stores the X input bits. The X bits are fed through Booth encoding circuitry (see Figure 6.10) which consists of a single DCVSL gate. The schematic for the Booth encoding DCVSL gate is shown in Figure 6.13. The carry save adder is implemented as two DCVSL gates, one for sum and the other for carry, just as in the ALU. Register cells hold the partial product results of the CS addition. The required shifting of two bits (multiply by 1/4) between partial products is implemented by routing between the slices. Schematics for multiplier cells not shown in this chapter appear in the appendix.

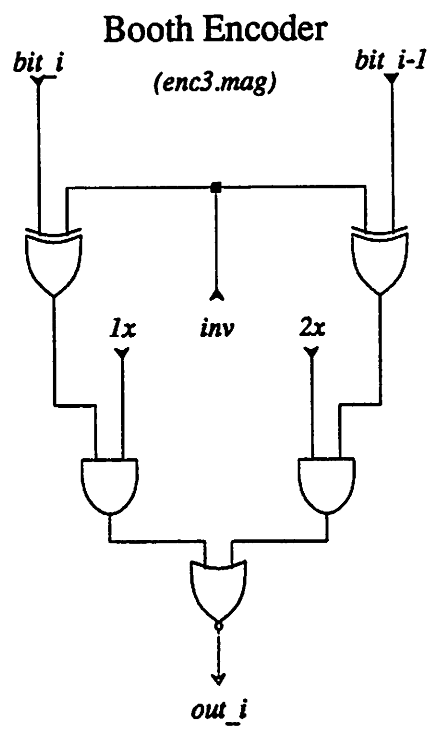


Figure 6.10: Logic to encode the multiplicand from the three Booth control signals.

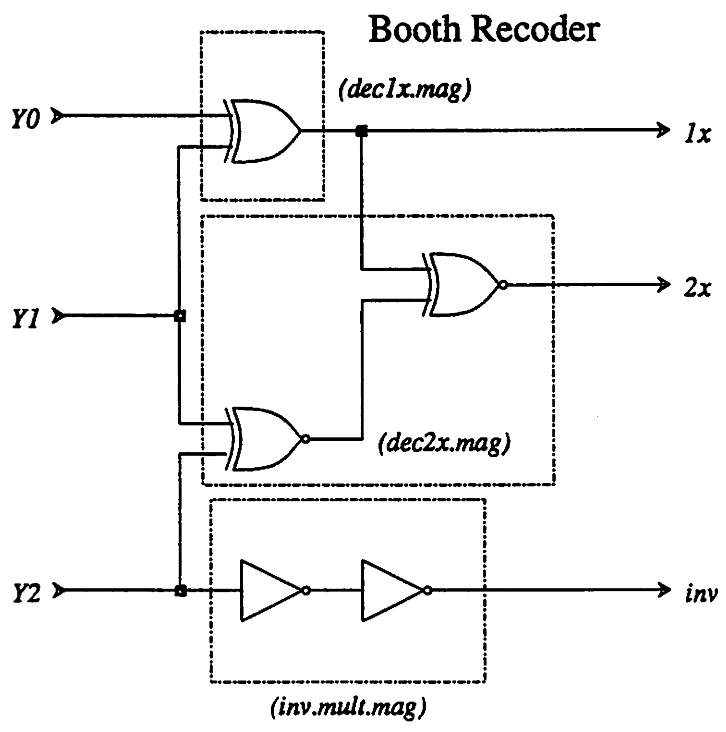


Figure 6.11: Booth Recoding logic.

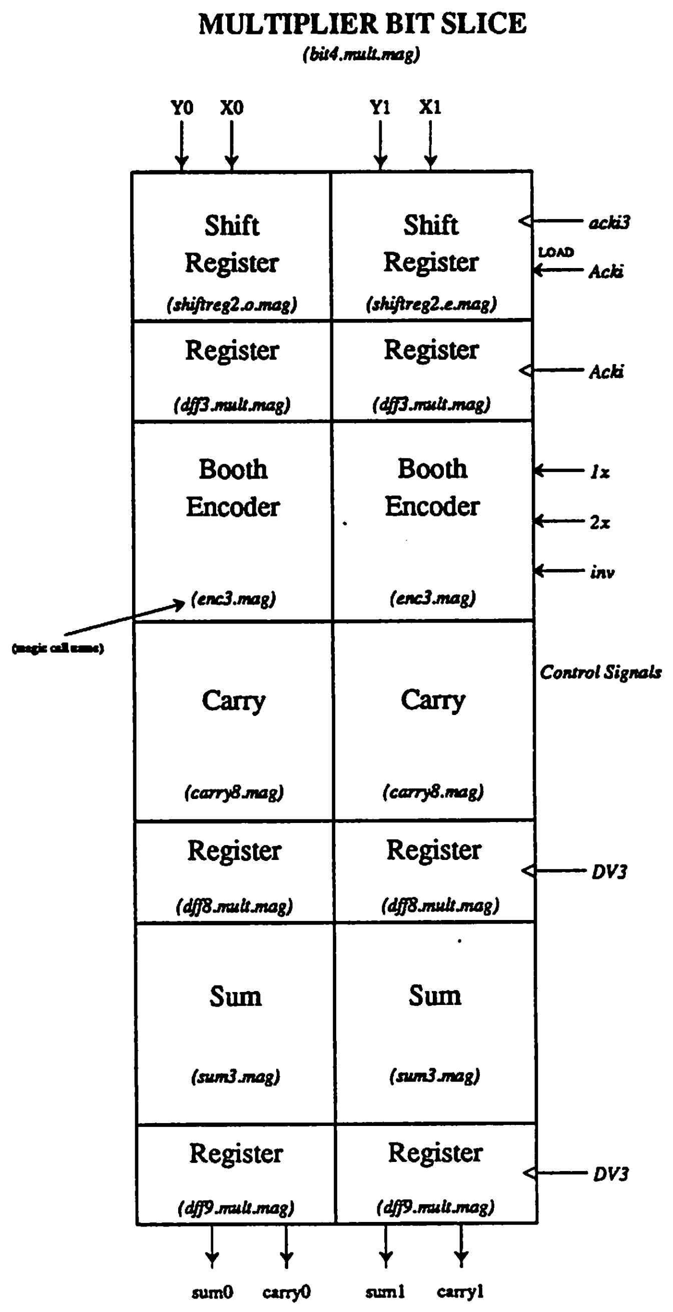


Figure 6.12: Bit slice of multiplier minus carry propagation adder section.

enc3.mag

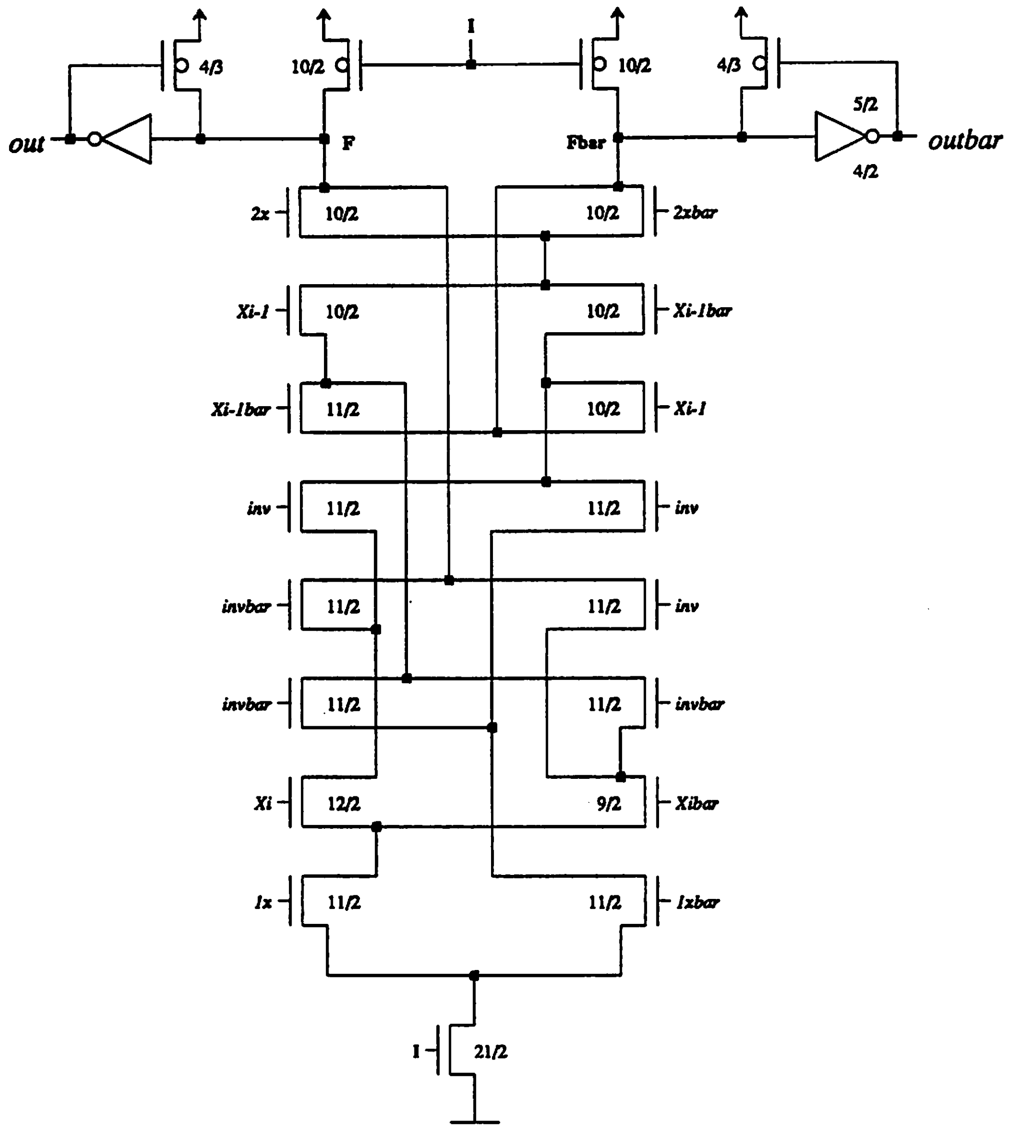


Figure 6.13: Booth encoder DCVSL gate used in the multiplier.

4. ROM

In the self timed paradigm, every stage must generate a completion signal. This applies also to ROMs used for program storage or coefficient storage. A DCVSL style ROM could have been designed from scratch, however it was felt that an existing library design with a small design modification would be sufficient, saving the rather long time required for layout in new designs. The existing library cell[74] ROM (also suitable for PLAs) consists of a core that is programmed by the presence or absence of one NMOS transistor per bit. A diagram illustrating the ROM operation is shown in Figure 6.14. During a pre-charge time, the bitlines are pre-charged high. The input address is typically decoded to select a single word in ROM during this time also. The bitline buffers are inverting so all of the outputs go LOW . During the evaluate time, the core transistors discharge the bitlines which read a logical 1 (NMOS transistor present) to Ground and do not affect the bitlines which read a logical 0 (NMOS transistor not present). The buffered outputs then change accordingly HIGH and LOW respectively.

The scheme to generate a **DV** or completion signal for the standard ROM was as follows: Presuming that the time to generate a 1 at an output is longer because it involves *changing* the state of the bitline from its pre-charge state by discharging it through an NMOS transistor, an extra bit can be added to each word which is always programmed to generate a 1 and that bit can be used for **DV** . This involves adding a "dummy" column to the ROM so that for an n-bit wide ROM, there is always at least $n+1$ outputs. The extra bit always is programmed to a 1. The one danger in the completion signal scheme is that, like other schemes which rely on matching, the delay time for all output bits to become valid must be less than or equal to the time for the extra **DV** bit. The library ROM design contains shorting metal busses which equalize the delay of the bits, reducing the danger. Also, since a **DV** signal itself must pass through at least one gate (in the handshaking circuitry) before causing any action to occur, the gate delay can alleviate problems due to mismatching in the delays between bits. However, caution must be exercised if the ROM outputs feed a register for example, because the set-up time requirements for the register may consume any margin given by the handshaking circuitry!

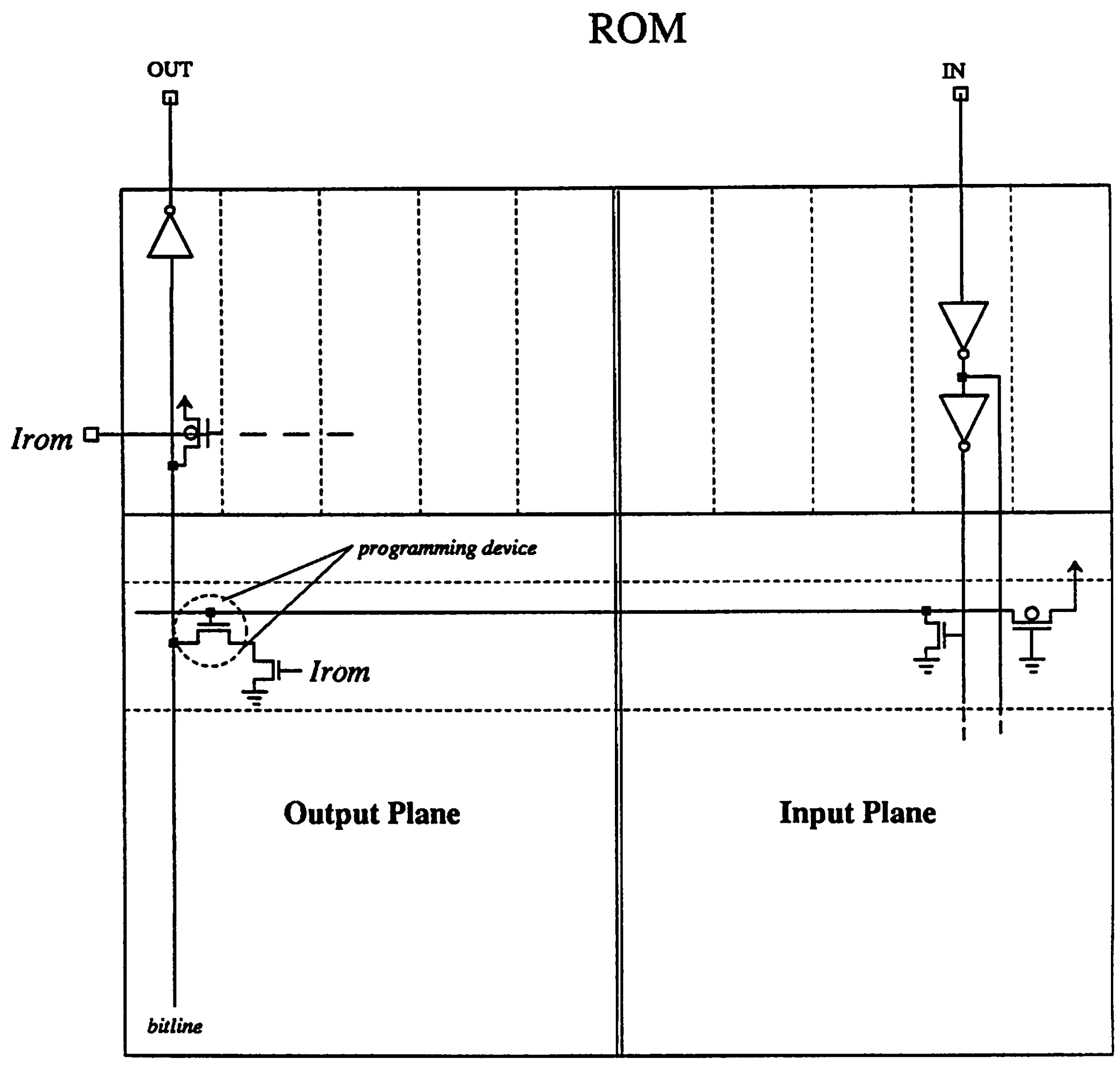


Figure 6.14: Diagram of the ROM adapted for self-timed applications.

5. RAM

Like the ROM described above, a RAM was adapted from an existing library cell[74]. The library RAM has separate *Read* and *Write* address and data ports. The storage core itself is made up of 3-T RAM cells and the bitlines for read and write are kept separate to support the two data ports. A block diagram of the self-timed version of the RAM is shown in Figure 6.15. The *Read* and *Write* address lines feed separate decoders and the internal read select and write select signals are effectively AND 'd together with an *Iread* and *Iwrite* signal. In the DSP design described next chapter, the RAM is read and written at the same time so the "I" signals are connected together. When *Iread* is low, the read bitlines are pre-charged, making the operation very similar to a DCVSL gate.

The generation of a completion signal is performed again (as with the ROM) by adding a special extra bit for each word. The bit always reads a logical 1 which is ensured by connecting the storage node of the 3-T cell not to the write bitline, but rather to V_{dd}. During pre-charge, the bitline is charged HIGH , so the inverted **DV** signal output is LOW as required. Thus, the self-timed RAM appears to the outside world as just another DCVSL stage.

Again, since DCVSL techniques are not strictly used in the RAM design, matching must guarantee that the *Read* time of the **DV** cell is at least as long as the *Write* time of the normal cells if **DV** is to indeed indicate when both reading and writing are complete. It is the case in most RAM designs that the *Read* time exceeds the *Write* time giving this approach a low risk factor.

6. Summary

The designs of some common datapath macrocells have been presented that meet the criteria of self-timed circuits. In all cases, the cells generate a completion signal that can be utilized by handshaking circuits to oversee data transfers. In most cases, a macrocell can be made by combining a set of DCVSL logic gates, the last of which contains completion signal hardware. The completion signal generation on a n-bit wide macrocell strictly requires an n-input c-element to examine all bits. However, due to the ability to closely match the pre-charge times of the gates, an n-input AND gate usually suffices. Further, for a macrocell in which the completion time is not data dependent,

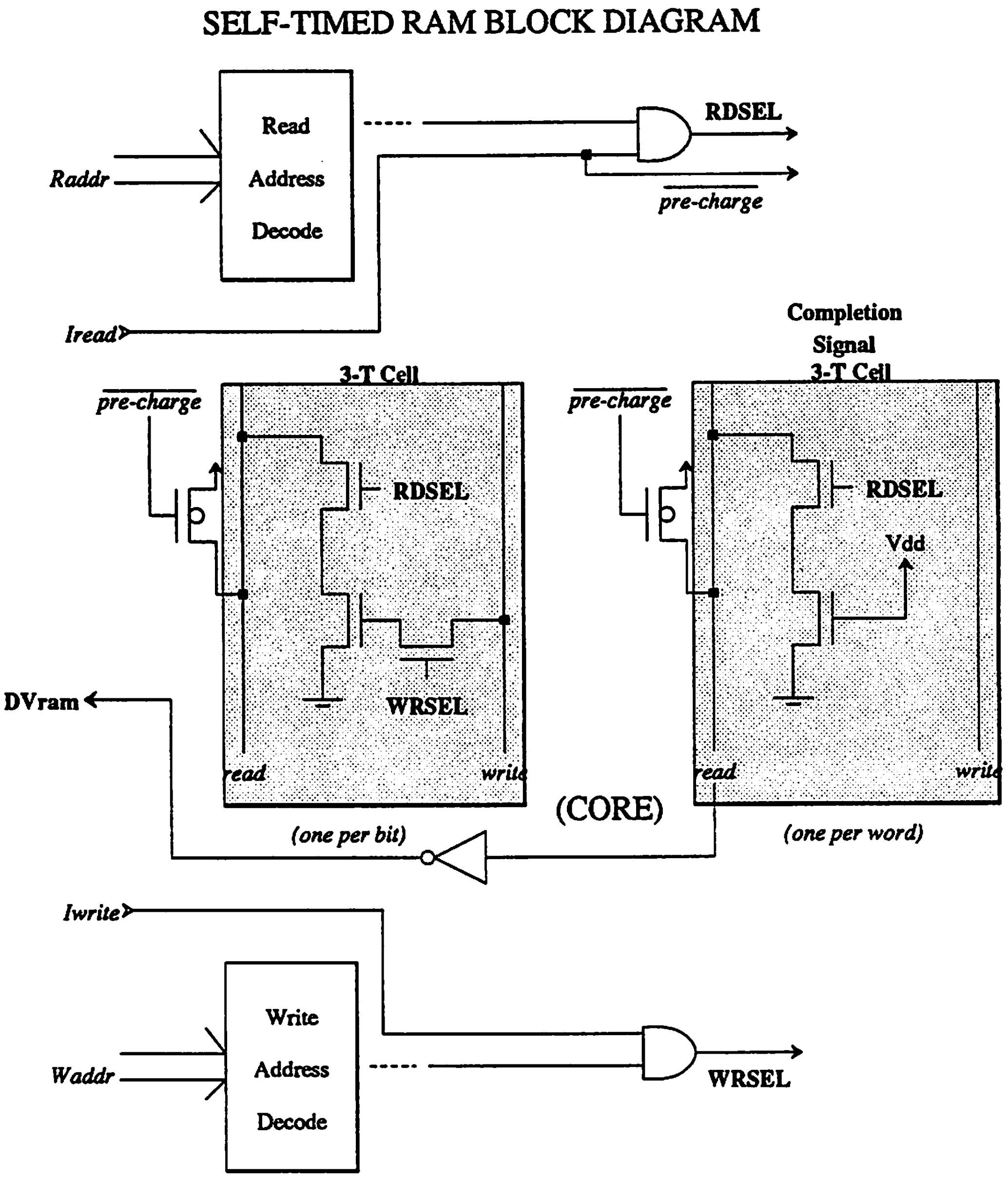


Figure 6.15: Diagram of the RAM adapted for self-timed applications.

circuit matching characteristics can be exploited to a greater extent allowing even fewer bits for the generation of a completion signal.

A barrel shifter and ALU were presented which are basically a cascade of DCVSL gates. The design of an iterative multiplier was shown in which the control of the calculation of the intermediate partial products was handled by a self-timed sub-system. Self-timed versions of a RAM and ROM were made by modifying existing synchronous designs. These contain a small overhead for the generation of a completion signal, however they depend on some circuit delay time matching to work properly.

Chapter 7

The Design of a Self-Timed DSP

The discussions in the last four chapters have covered how self-timed circuits are implemented, the synthesis of reliable handshaking circuits, c-element design, and the design of the macrocells that make up common datapaths. In this chapter, all of the techniques described are combined to design a fully self-timed programmable digital signal processing IC. The architecture of the DSP chip is presented along with the system timing. The resulting instruction set is given and an assembler for generating ROM code from the assembly code is discussed. Finally, details of the overall chip design are presented.

1. History of Signal Processing Applications and DSPs

If one examines the signal processing applications products produced in the last decade, there is strong competition between digital techniques and sampled data techniques. In fact, the introduction of sampled data techniques in the mid 1970's really revolutionized the integration of signal processing tasks. Switched capacitor integrated circuits have dominated until only recently. Ironically, it was the fact that sampled data filters could be integrated on a standard digital fabrication process that gave them credibility. Despite their huge success, sampled data circuits were targeted for extinction since many people in the design community still had their sights set on digital processing for many of the traditional reasons (ease of design, long term stability, re-programmability, more applicable to CAD based design). They promised that scaling of the digital process would soon eliminate the competition by switched capacitors.

In reality, the analog designers, while a minority, have been able to improve their skills and take advantage of the scaling of technology remarkably well. It is only quite recently that digital signal processors have begun to take hold of the market for signal processing products. The analog designers hit some fundamental limitations on the scaling of sampled data circuits while the digital processes have been scaled down sufficiently to allow a very high level of processing power. Also coupled to this is the complexity of the systems that are being developed on silicon. As the design rules have been reduced, more functionality has been introduced. In the past for example, a chip containing 40 poles of filtering that could be used as the front end of a modem was a viable product using switched capacitors. Currently however, the level of integration has grown to the point where the entire modem is on a single chip. Since many telecommunications products rely on signal processing algorithms suited for the digital domain such as adaptive filters and echo cancellers, most chips (or systems) on the market now are at least hybrids of digital and sampled data techniques. Also, the growth of *general purpose* digital signal processing circuits has been very steep in the last several years as the power of the digital processors has reached a usable level in terms of algorithm requirements.

The introduction of general purpose digital signal processing ICs such as the TMS320 series from Texas Instruments, DSP56000 from Motorola, and the DSP32 from AT&T made the technology widely available and suitable for many different applications. These DSPs provide an architecture that is powerful enough to fit a wide variety of algorithms along with being programmable by the user (either with off-chip program ROM or by mask programming). The success of these products and the products in which they are utilized has driven the industry to continue to pursue advancements in power and performance of such devices so that they will proliferate even more. Advancements in analog-to-digital conversion have also fueled the interest in digital signal processors. Using oversampling techniques, much of the A/D converter can be shifted into the digital domain. Of course, it is the continued scaling of the digital IC process which has made these developments possible. The traditional reasons for using digital techniques for signal processing system include long term stability, rapid modifications through re-programming, and the reliable prediction of behavior and noise performance.

Digital signal processors represent some of the highest performing micro-computers because of the incredible rates of computation they achieve as required by many real time systems. The speed of operation and the functionality offered are key aspects to their success. As the IC process technology is advanced, DSP chips strive to offer even more power in terms of the system they may implement and greater speed of operation. This typically is achieved by using smaller and faster transistors and placing more of the transistors on a chip to extend the functionality. General purpose DSPs on the market now run in the 10-30MHz range for each instruction. As continued scaling of the digital IC process proceeds, these clock rates will extend well beyond 100MHz. Even so, only a subset of the desired applications can actually be handled. For example, the general purpose DSPs mentioned above fall way short of the speed requirements for doing video signal processing. Thus, the motivation to obtain more speed in DSPs is will continue to be very great.

2. System Description

The goal of this research has been to apply the principles of self-timed circuit design to make a general purpose DSP. Given the magnitude of this undertaking, a relatively simple architecture for the chip was chosen. It resembles the datapath architecture for a *Texas Instruments TMS320210* without the address arithmetic capability. Figure 7.1 shows a diagram of the datapath. Dotted lines denote pipelining stage boundaries and shaded rectangles represent registers. The challenges of making the datapath self-timed include the existence of feedback and the programmability of the architecture (via the different types of instructions). The operations in each stage of the pipeline are detailed below:

Stage 1

The first pipeline stage in the datapath contains the memory. A RAM supplies a temporary storage area and it is both read and written to during the first pipeline cycle[1]. RAM values which are read are supplied to the datapath cells of the next stage as well as another pipeline that is used to store words that are to be written back into RAM for the purposes of moving data. Due to the pipelining delays, data *read* from the RAM is specified in the current instruction, while data *written* to RAM was specified two instructions earlier. The three registers in the feedback path around the RAM emulate the delays seen through the rest of the datapath to make the constraint for writing the RAM consistent for instructions that move data through the datapath or just around the

[1]Actually, the first pipeline stage of the entire system is the instruction fetch described in the controller section.

feedback path. The source for writing the RAM can be either the Accumulator, the RAM feedback delay line, or the Input register. The feedback path is used mainly for "data move" instructions such as shifting data down a filter shift register implemented by RAM. A coefficient ROM (CROM) also exists in the first stage for implementing fixed coefficient digital filters efficiently. The CROM is read concurrently with the RAM.

Stage 2

The second pipelining stage of the datapath contains the barrel shifter and the multiplier. These two cells are essentially in parallel which allows using either of them in an instruction but not both since they are preceded and followed by MUX circuits working in unison. The input to the shifter comes from the RAM. The multiplier can be configured to multiply a variable read from RAM by a fixed coefficient from CROM *or* it can multiply two variables from RAM. The Y input to the multiplier is taken from the CROM or the RAM feedback delay line as determined by a MUX, and this allows multiplying two variables from RAM which are read on consecutive instructions.

Stage 3

The last stage of the pipeline contains the ALU and Accumulator. A MUX selects the ALU A-input from either the shifter or multiplier. The B-input to the ALU is the Accumulator. Since the RAM is 16-bits wide while the ALU/Accumulator is 32-bits wide, provisions are made to write either the high or low byte of the Accumulator back into RAM. The feedback around the entire datapath allows values from the Accumulator to be written back into RAM or to the Output Port.

2.1 Data Stationary Architecture

In micro-coded digital signal processors, one encounters mainly two types of architectures. The most common is *Time Stationary* (TS), where a single micro-coded instruction contains the control signals that indicate what each element in the processor will do at a given *time*, or more precisely, the time during a given instruction cycle. Figure 7.2 illustrates this. The microcode in the program ROM output register indicates what each block does at t_1. The corresponding assembly code typically has separate fields per instruction, again to specify what happens at each stage of the datapath. The *data*

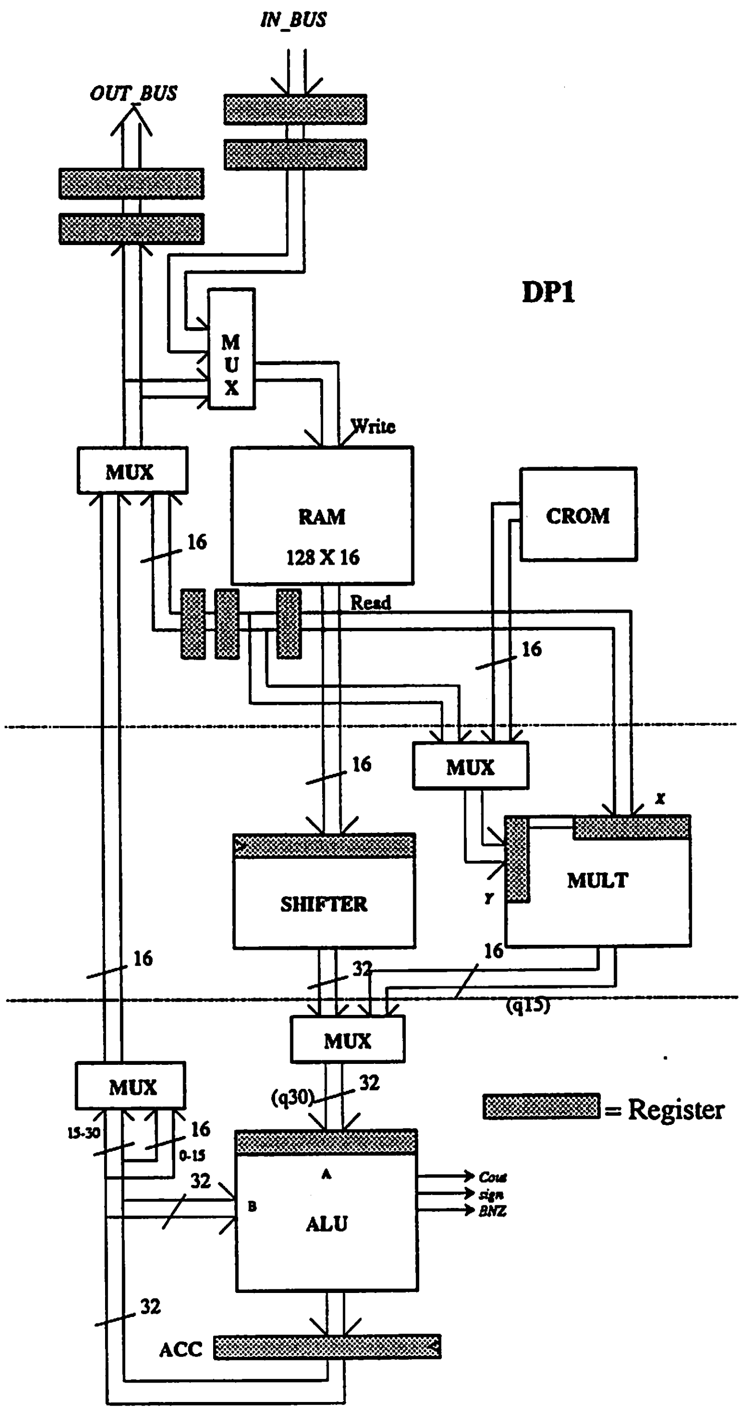


Figure 7.1: Block Diagram of datapath.

referred to in each field however, is different with the existence of pipelining. That is to say, in the figure, if the data is read from the RAM at t_i, then the data that has moved into the shifter is that read at t_{i-1} and the data in the ALU is that read at t_{i-2}. The assembly code can be fairly confusing under the circumstances. For a three stage pipelined datapath, each instruction must refer to three pieces of data.

Another processor architecture is called *Data Stationary* (DS), and as the name implies, the assembly code is written to describe what happens to a single data word as it proceeds through the pipeline. The bottom of Figure 7.2 shows that program ROM micro-code is passed though its own pipeline registers which mirror the movement of data words though the datapath. In this way, the control signals supplied to a block for a certain data word reach the block at the same time as the data. The assembly code is more coherent since each instruction now contains the information about what happens to a *single* data word as it moves through each stage of the pipeline. The hardware cost for implementing a DS processor is the extra registers for the micro-code. Also, the simplicity of the instructions breaks down for control oriented branching instructions where a test on a the accumulator (say $ACC \leq 0$) determines whether a branch should occur. In a DS processor as shown in the figure, the test is actually made on data read two cycles earlier. A commercial example of a TS processor is the *Motorola DSP56000* while the *AT&T DSP-32* is a DS processor architecture.

For a self-timed processor, the Data Stationary architecture makes sense for reasons beyond those described above. While a *sequence* of operations is imposed on the datapath macrocells, the exact temporal information about the operations is not obtainable since it is process and data dependent. For a certain series of instructions, we may know that a word of data will be read from RAM, then shifted three places left by the barrel shifter, and then added to the Accumulator. If the data transfers are self-timed however, one would have to to make sure that all transfers are completed before allowing the control signals (micro-code) to be updated for a new instruction if a TS architecture is employed. To do so would require communication lines between the controller and all of the datapath stages as shown in Figure 7.3. It is exactly that sort of global communication which is a problem for synchronous designs (in the form of the clock signal). A DS architecture more naturally fits the self-timed paradigm because because communication for both the data and control signals takes place only between adjacent stages. The controller itself only communicates with the first stage of the datapath pipeline

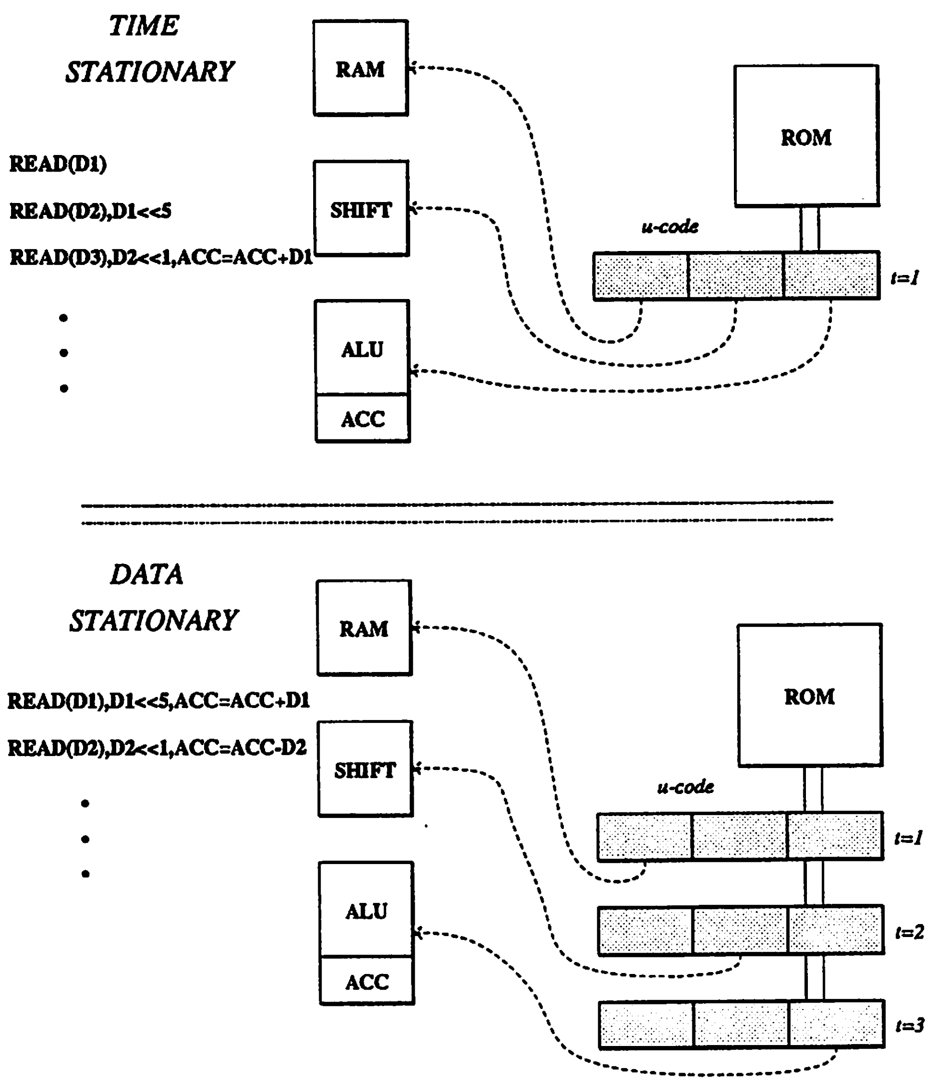


Figure 7.2: Data Stationary and Time Stationary processor architectures.

as illustrated in Figure 7.4. If one envisages controlling the instruction pipeline with the same handshake signals that control the data transfers, everything is synchronized. As a data word moves from one stage to the next, the control signals that affect computation on that data word follow along. For this reason, a Data Stationary architecture was employed in the self-timed DSP described in this chapter.

2.2 Instruction Set

Given a choice for the datapath architecture and basic functionality of a DSP, the instruction set can be derived. The architectural choice is often based on many things related to the applications of the chip. The most optimum choice for an architecture is another active research area which includes the study of parallelism in signal processing algorithms and the job of scheduling tasks when multiple processors are utilized. For the self-timed DSP, the architecture of Figure 7.1 was chosen since it is suitable for general purpose use.

An instruction set should be easy to read and write to be the most useful. Data Stationary architectures tend to simplify the instruction set by making the instructions look more like an equation operating on a single data word. The math instruction set for the self-timed datapath is given in Figure 7.5. There are basically two types of math instructions: those using the barrel shifter and those using the hardware multiplier. The multiplier instructions are split into two sets also depending on whether the multiplier Y-input originates from the CROM or the RAM delay line. The output of the first register in the RAM feedback delay line is referred to as the "T" register, a name that parallels the multiplier input register of a *Texas Instruments TMS32010.* A final sub-division of the instructions is made to differentiate where the data is directed. The "Z" field of an instruction is either a *write* location in RAM or the output port. The $Z = ACC$ construct directs a word in the Accumulator to the output port or back into RAM. Similarly, the $Z = X$ construct directs a word read from RAM to the same destinations. This is typically used for data moves in RAM.

There are a few additional instructions used specifically for control operations in the datapath. These are shown below:
The GOTO instruction just causes the program counter to jump to the instruction denoted by *label* in the program. Similarly, the conditional branch only executes the GOTO when

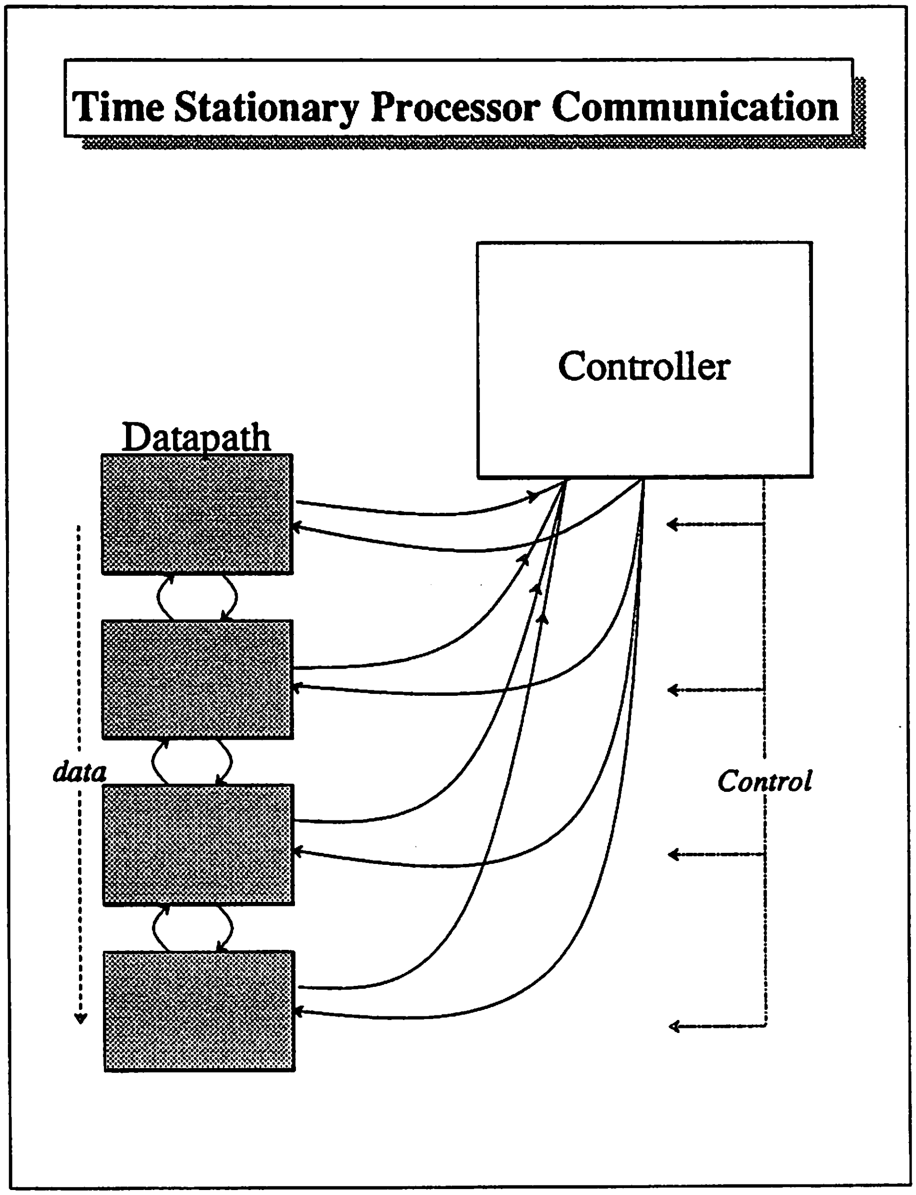


Figure 7.3: Time Stationary processor communication.

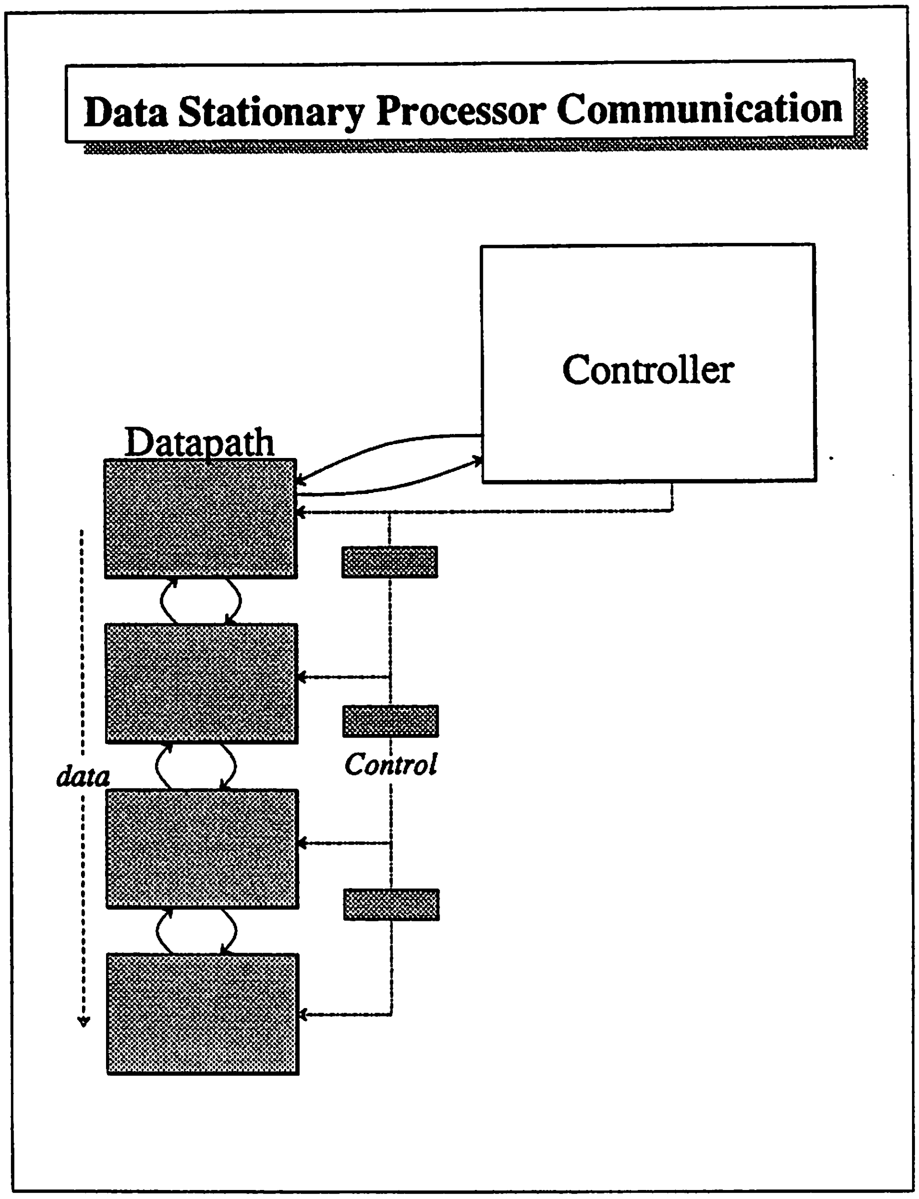


Figure 7.4: Data Stationary processor communication.

Math Instructions for Self-timed Datapath

KEY:

X = RAM Read Location
Z = RAM Write Location or OUT (output register)
T = MULTIPLIER Y Input Register = LAST RAM VALUE READ
B = Accumulator or "0 "
ACC = ACC (Accumulator bits 15-30) or L_ACC (Accumulator bits 0-15)
C = Coefficient ROM Location
"~ " = one's complement
OP = +, −, |, &, ^

( ) = required field
[ ] = optional field

———— DATAPATH MATH INSTRUCTIONS ————

[Z=]ACC = B;	Set Accumulator to Zero or Accumulator (NOP).
[Z=]ACC = B OP [~]X[<<0-15];	Add,Subtract RAM to Accumulator,[0] with optional shift. Output to Z.
ACC = B OP [~][Z= X][<<0-15];	Add,Subtract RAM to Accumulator,[0] with optional shift. Data Move.
[Z=]ACC = B OP [~]X * T;	Multiply, accumulate [invert], Y input to mult is RAM location read on last instruction. Output to Z.
[Z=]ACC = B OP [~]X * C;	Multiply, accumulate [invert], Y input to mult is CROM location Output to Z.
ACC = B OP [~][Z= X] * T;	Multiply, load ACC [inverted], Data move.
ACC = B OP [~][Z= X] * C;	Multiply, load ACC [inverted], Data move.
any above, (Ram = IN);	Any of the above instructions can have an INPUT to RAM specification as long as Z is not a RAM write location.

Notes:

Substituting "<< −" for "<<" turns off sign extension in the barrel shifter.

If Z not specified, Write Location of RAM will be the same as a specified Read Location.

If no Read Location specified, Write is DISABLED except on an INPUT specification.

All Writes to memory including INPUT instructions have a latency of three instructions.

Figure 7.5: Math Instruction Set for datapath.

Control Instructions for Self-timed Datapath

GOTO *label*;	Unconditional Branch to instruction line with *label*.
BRANCH *label*;	Unconditional Branch to instruction line with *label*. (same as above)
if(*cc*) GOTO *label*;	Conditional Branch. *cc* is condition code selected.

Figure 7.6: Datapath Control Instructions.

the condition is satisfied. The four condition codes implemented in the self-timed DSP are:

1. UC	Unconditional	Tied to V_{dd}.
2. BZ	Branch on Zero	High when $ACC = 0$.
3. BNZ	Branch not Zero	High when $ACC \neq 0$.
4. SGN	Branch on sign	High when $ACC \geq 0$.

The mnemonics such as BNZ are symbolic only. The instructions must refer to the number of the condition code unless a label is assigned to them, in which case the mnemonic can be any arbitrary string. The assignment of labels is discussed in the next section on the assembler.

2.3 Assembler

The main purpose for an assembler is to translate the higher level assembly code of a processor into the actual program ROM bits. In that respect, an assembler is mainly a parsing program. Other enhancements to assemblers are usually provided in order to make the assembly code more easily written and read. For example, EQUuate statements allow the user to equate strings to constants so that more meaningful labels can be used. A filter coefficient may be stored in CROM location 8, and it is necessary to multiply it by the input sample stored in RAM location 1f. Rather than writing the instruction

ACC = ACC + 1f * 8;

by using two **EQU**ate statements, the same instruction can be written more clearly as

ACC = ACC + input * coeff1;

Text labels for line numbers also make branch statements clearer.

Another task of many commercially available assemblers is to check for errors in the input assembly code file. The errors can range from syntax errors to timing errors or warnings. An example of this is a branch instruction. In a data stationary processor, the branch may key off a data word read several instructions previous to the current instruction. The assembler can spot situations where the programmer may have overlooked this latency and issue a warning.

An assembler was written for the Instruction Set presented above so that programming the self-timed datapath could be more efficiently completed for a variety of applications. The program, named *asm*, was written in C language using the UNIX [2] utilities LEX and YACC , which were written expressly for constructing parsers quickly. The program LEX recognizes lexical structures in the input stream and then either takes actions or passes the structures, called "tokens" to YACC . The YACC program accepts a special grammar language input to specify the correct grammar expected between tokens. It reports errors automatically and generates a C program that gets compiled into your main program (YACC stands for "Yet Another Compiler Compiler"). A diagram of the structure of *asm* is shown in Figure 7.7.

Valid assembly code consists of the following: 1) A declarations section which contains a number of **EQU**ate statements that equate character strings to constants, and 2) a code section which contains the actual instructions. Instructions are terminated by a semi-colon and comments are allowed either after the semi-colon of any instruction (up to the carriage return) or in C language style using "/*" and "*/" to begin and end the comment. The declarations section is begun by the word DECL at the beginning of a line. The declarations section is ended and code section is begun by the word START at the beginning of a line. The code section must contain instructions of the form presented above, however any integer constant may be replaced by a string if it appears in the declarations section. Valid labels are a character string followed by a colon (such as *Label:*). The set of valid keywords and operators that make up instructions is:

[2]UNIX is a trademark of AT&T

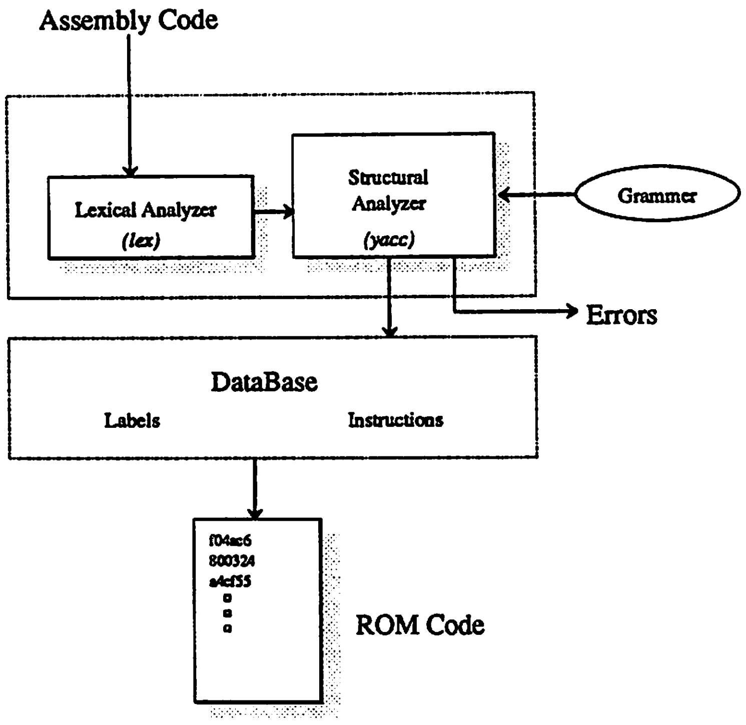


Figure 7.7: Diagram of the assembler written for the self-time DSP.

$$\begin{aligned} Keywords &\in \{ACC, L\_ACC, OUT, IN, GOTO, BRANCH, T, IF\} \\ Operators &\in \{+, -, |, \&, \hat{}, \ll, \ll-, *, \tilde{}\} \end{aligned}$$

The keywords may be in all upper case or all lower case letters, however not a mixture. Labels on the other hand, are read literally and may contain any mixture of upper and lower case letters. The lexical analyzer recognizes digits as any combination of numbers from 0 to 9 and words as any combination of letters and numbers that follows a leading letter. As mentioned, a label is a word followed by a colon. White space is ignored in the input. An example of assembly code is shown next to clarify these rules. The assembly code for an IIR filter implemented on the DSP chip is shown below:

```
/*   iir2.asm    */
/*
 *   FILTER ASSEMBLY CODE FOR 8 pole BPF in file "iir_design"
 *      (using multiplier)
 *
 */

/*************************************************************************
```

```
**************************************************************************/
DECL
        /* Filter State variable registers */
        EQU     del1    0
        EQU     del2    1
        EQU     del3    2
        EQU     del4    3
        EQU     del5    4
        EQU     del6    5
        EQU     del7    6
        EQU     del8    7
        EQU     del9    8
        EQU     del10   9
        /* others */
        EQU     tmp     10
        EQU     input   11
        EQU     tmp2    12
        /* filter coefficients in CROM */
        EQU     a1      8
        EQU     a2      9
        EQU     b1      10
        EQU     a1b     11
        EQU     a2b     12
        EQU     b1b     13
        EQU     a1c     14
        EQU     a2c     15
        EQU     b1c     16
        EQU     a1d     17
        EQU     a2d     18
        EQU     b1d     19
START

INIT:   acc = 0;
        /* clear all state registers to zero */
        (del1=acc)=0;
        (del2=acc)=0;
        (del3=acc)=0;
        (del4=acc)=0;
        (del5=acc)=0;
        (del6=acc)=0;
        (del7=acc)=0;
        (del8=acc)=0;
        (del9=acc)=0;
        (del10=acc)=0;
        /* done, begin filter program */
sample: acc = 0 | del2 * a2,(input=in); read input sample
        acc = acc + (tmp=del1)<<15;     coeff a1 > 1
        acc = acc + del1 * a1;          fractional part of a1
        (del1=acc) = acc + input<<15;
        acc = acc + del2<<15;
        /* acc = acc + tmp<<15;         coeff b1 > 1 */
        acc = acc + (del2=tmp) * b1;
        /*  acc is output of first section */

        /* section 2 */
        (tmp=acc) = acc;
        acc = 0 | del4<<15;
        acc = acc - del3<<15;
        acc = acc + (del4=del3) * b1b;   data move
        acc = acc + (del3=tmp)<<15;
        (tmp2=acc) = acc;

        acc = 0 | del6 * a2b;
        acc = acc + (tmp=del5) * a1b;
        (del5=acc) = acc + tmp2<<14;     scale by 0.5
        acc = acc + del6<<15;
        acc = acc - (del6=tmp)<<15;
        acc = acc + tmp * b1c;

        acc = acc + del8 * a2c;
acc = acc + (tmp=del7)<<15;
(del7=acc) = acc + del7 * a1c;

acc = 0 | del8<<13;              scale by 1/4
acc = acc + (del8=tmp) * b1d;
acc = acc + del7 << 13;

acc = acc + del10 * a2d;
acc = acc + (del10=del9)<<15;
(del9=acc) = acc + del9 * a1d;
(out=acc) = acc;                 send result out
goto sample;
```

The IIR filter code uses a combination of multiplies and shift/adds so it is a good example. Notice that the math is performed in the upper Accumulator bits for this program. Therefore, if a number from RAM is to be added to the Accumulator, but divided by 2 first with the shifter, then the shift amount should be 14 places. A shift of 15 places left justifies the number in the Accumulator assuming $q.15$ format in RAM and $q.30$ (double precision) format in the ALU. The network diagram of the filter is given later in Section 6.5.

The assembler automatically checks for syntax error in the input file because the grammar must comply to the grammar rules given to YACC . Beyond that, error checking exists to ensure that the user does not exceed hardware limitations such as writing to a RAM address that is too high or shifting more than 15 places. The assembler writes two ROM code output files that interface directly to the system simulator THOR that was used to test the architecture. Those output files can then be converted to a proper parameter file for the LAGER layout system [72, 73] used to complete the chip design by running the program *ROMconvert*. The assembler also writes an output file that gives more complete information about each instruction such as the program counter value and the condition of all of the control signals. Therefore, the command

```
asm  code
```

uses/generates the following files:

1. Reads assembly code from file code.asm.
2. Generates output file code.out.
3. Generates THOR Hex ROM-code file code.ROM_L (lower 32 bits).
4. Generates THOR Hex ROM-code file code.ROM_H (upper 8 bits).

The THOR simulator is limited to 32-bit values so two ROM-code files must be generated to simulate the 40-bit wide control word of the system. More information about the assembler can be found in Appendix B.

3. System Timing

The datapath drawing in Figure 7.1 has an implicit idea of timing in it by the way the macrocells are interconnected, the location of pipeline registers, and the resulting data flow. However, the actual system timing is something that the designer imposes on the circuit blocks and in a self-timed system, this is done via the interconnection of a set of handshaking circuits. In fact, the system design is really just the design of the handshaking circuits since the datapath macrocells simply act as latencies added to the handshake signals between stages. Knowing what we'd like to have the datapath do at this stage of the design, a rough sketch of the control/handshaking circuitry can be made. From the desired data flow in Figure 7.1 and the decision to employ a Data Stationary architecture, the drawing of the control and handshake circuits for the datapath was formed and it is shown in Figure 7.8. The core of the controller is the familiar micro-coded control store consisting of a program ROM (PROM) and program counter (PC). The PC increments for each instruction and its output addresses the PROM. The actual timing of the controller is done with a free running 4-cycle handshake circuit. Provisions for branching are also present. A high-level view of the controller is really that of a self-timed signal generator. The signal is the micro-code and the timing consists of a series of requests which occur as each instruction is fetched from PROM.

The requests from the controller are fed to the datapath handshaking circuitry. There is one handshake circuit for each of the datapath pipeline stages as shown. The first stage RAM/CROM HS circuit is the most complicated since it must handle requests from the controller, the ALU (bottom of the datapath), and the I/O circuitry. Because the multiplier and barrel shifter are configured in a parallel fashion, there are two sets of handshake signals between the second and third pipeline stages. In effect, the *Req* and *Ack* signals from the first stage are de-multiplexed to drive the multiplier/barrel shifter, and then multipliexed to drive the ALU.

An instruction pipeline mirrors the datapath stages as required by the choice of a Data Stationary architecture. Control signals for the datapath cells are tapped off at appropriate points and they are shown in Figure 7.8 as dotted lines. Note that since the *write* operation in the RAM coincides with data coming out of the bottom of the datapath, the write address (*Waddr*) comes from the last stage in the instruction pipeline (ctl_D). The clocking of the instruction pipeline is done by the same signals which clock data

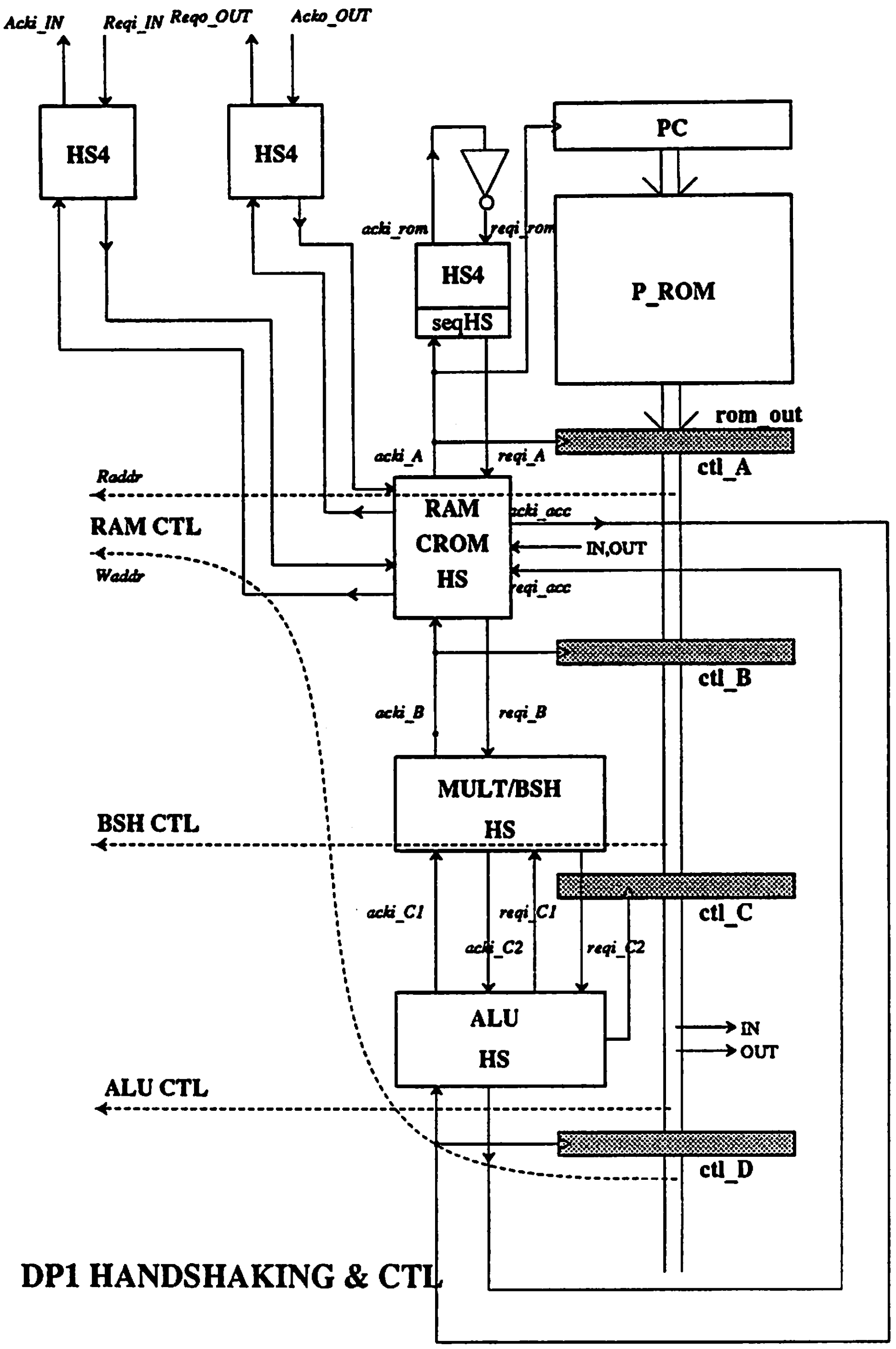


Figure 7.8: Handshaking and Control circuitry block diagram for the self-timed DSP.

through the datapath. Thus, the instruction control signals follow along with the data. Names have been added to the diagram to begin the important process of clearly labeling the signal names of the DSP. Handshake signals have a post-fix of "A,B,C" respectively for the first, second and third pipeline stages. The design of the handshake circuits shown in Figure 7.8 is critical to the system timing and performance. The next section discusses these handshake circuits in detail.

4. Datapath Handshaking

In this section, the specification and design for the datapath handshaking are presented.

4.1 Controller

The program store for the self-timed DSP contains horizontal micro-code which eliminates the necessity for an instruction decoder. The controller consists basically of a program counter (PC) and program ROM (PROM) as shown in the block diagram of Figure 7.9. Of course, rather than using a global clock to increment the PC, a free-running handshake circuit acts as a timing generator so that the controller issues instructions at exactly the rate at which the datapath demands. The enable signal $ENrom$ allows the generator to be switched off or stepped for the purposes of testing. Given that $ENrom$ is high, the HS4 circuit free runs because its $Reqi$ signal is just $\overline{Acki}$. The rate at which it receives acknowledge signals at the output port therefore determines the rate at which the handshake circuit runs. Each new cycle of the $Reqo$ ($= Irom$) signal from the HS4 block enables the PROM to read an instruction. The falling edge of $Irom$ increments the program counter to the next instruction address. As with any DCVSL stage, the completion signal from the PROM becomes the $Reqo$ signal sent to the next stage. It is $reqi\_A$ on the diagram of Figure 7.8. Similarly, $Acko$ is $acki\_A$ in Figure 7.8.

The branching instructions for the processor require that the controller be a bit more sophisticated than shown in Figure 7.9. A program branch implies that the program counter must be loaded with a new address. One bit of the micro-code instruction output acts as a "branch bit", signalling a branch instruction. The necessary sequence of events for executing a branch includes sensing this bit and in the case of it being asserted, loading the program counter with an address derived from the instruction itself. Conditional

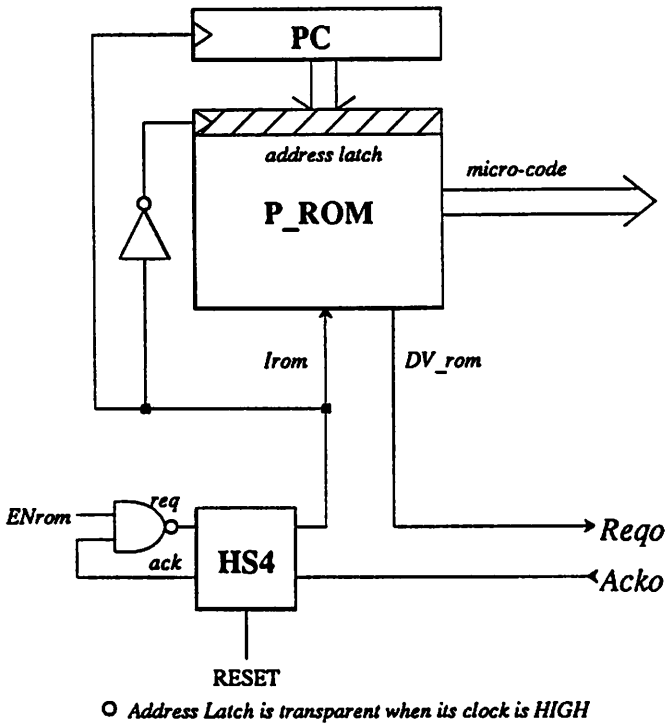


Figure 7.9: Block diagram of the controller used for the DSP chip.

branches perform the loading based on the state of signals originating from other parts of the datapath (usually the ALU). The sense/branch operations could be performed in another stage following the basic controller, however the tasks are small and should be included in the basic PROM cycle to avoid another stage of pipelining. The branch bit however is not valid until the the *DVrom* signal goes high.

A *sequential handshake* circuit was employed to impose the sequence of Read PROM/branch operations during each cycle without any more pipelining. The guarded command for a sequential HS circuit is:

$$*[Reqi^+ \rightarrow Reqo^+; Acko^+ \rightarrow Acki^+] \tag{7.1}$$

The *Acki* signal waits until *Acko*$^+$ which implies that the next circuit has completed. Therefore a sequence of two operations occur in a single handshake "cycle". The sequential handshake circuit is shown in Figure 7.10.

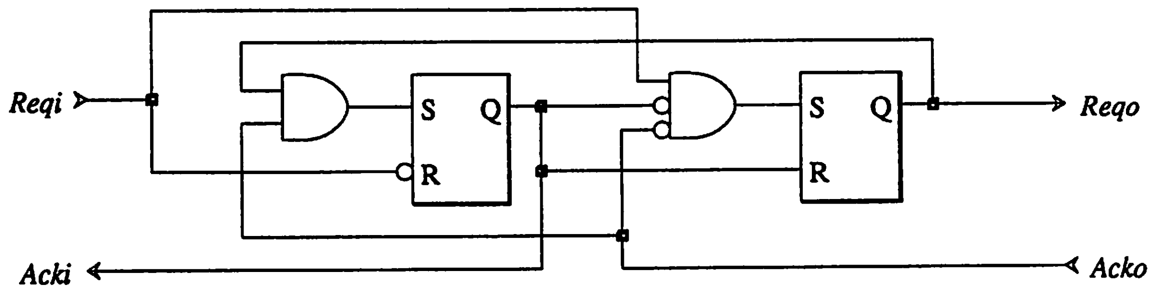


Figure 7.10: Sequential handshake circuit.

Figure 7.11 shows a more detailed block diagram of the chip controller incorporating the branch circuitry. The *DVrom* signal acts as the *Reqi* of the sequential handshake circuit. The *Reqo* of the sequential handshake circuit clocks a register which stores the branch bit from the PROM micro-code output. When a branch instruction occurs, the cycle in which it occurs is used to load the PC with the branch address. No output request is sent from the controller until the next non-branch instruction is read. This scheme is required because the branch address bits in the micro-code are shared with other control bits used in normal instructions. Also, the branch instruction is stand-alone; no data-path operations are specified in the assembly code on a branch instruction. Therefore, the datapath performs no operation during a branch which can be achieved by simply not supplying a request to it. The output of the branch register (labeled "branch" in the figure) controls a 1:2 demultiplexer which routes the *Reqo* of the controller either to the datapath as the *reqi_A* signal, or back to the controller *Acko* signal. The de-selected output of the demultiplexer stays LOW . The operations of the controller can be described in pseudo computer source code as follows:

```
controller() {
    while(TRUE) {
        Read Program ROM;
        Latch Branch bit;
        if(branch bit == 1) {
            Acko = 1;
            if(condition code == 1)
                Load PC with Branch Address;
        }
        else {
            Reqo = 1;
            Increment PC;
            while(Acko == 0)
                wait;
        }
        Pre-charge Program ROM;
    }
}
```

The sequential handshake circuit ensures that the branch bit is not latched until the program ROM is read first. The block labeled "Reg/demux" containing a *DV* signal output is a delay block that duplicates the circuitry of the branch register and demultiplexer in order to generate a Data Valid signal which indicates when the output request is ready. The lack of a register with completion information and the use of a standard CMOS demux circuit necessitates the delay. In Figure 7.11, the HS block at the output and the register which latches the micro-code instruction constitute the beginning of the instruction pipeline and they are included for clarity. The HS circuit corresponds to the RAM HS block in Figure 7.1. A master RESET signal clears the PC to zero and initializes the handshake circuits.

4.2 Instruction Pipeline

In Section 2.1, the Data Stationary architecture was presented for use in self-timed DSPs. A requirement for having a DS architecture is a pipeline for storing instructions as they proceed through the datapath. In a Time Stationary processor, a controller, or a collection of controllers, must send signals to each stage in the datapath for every

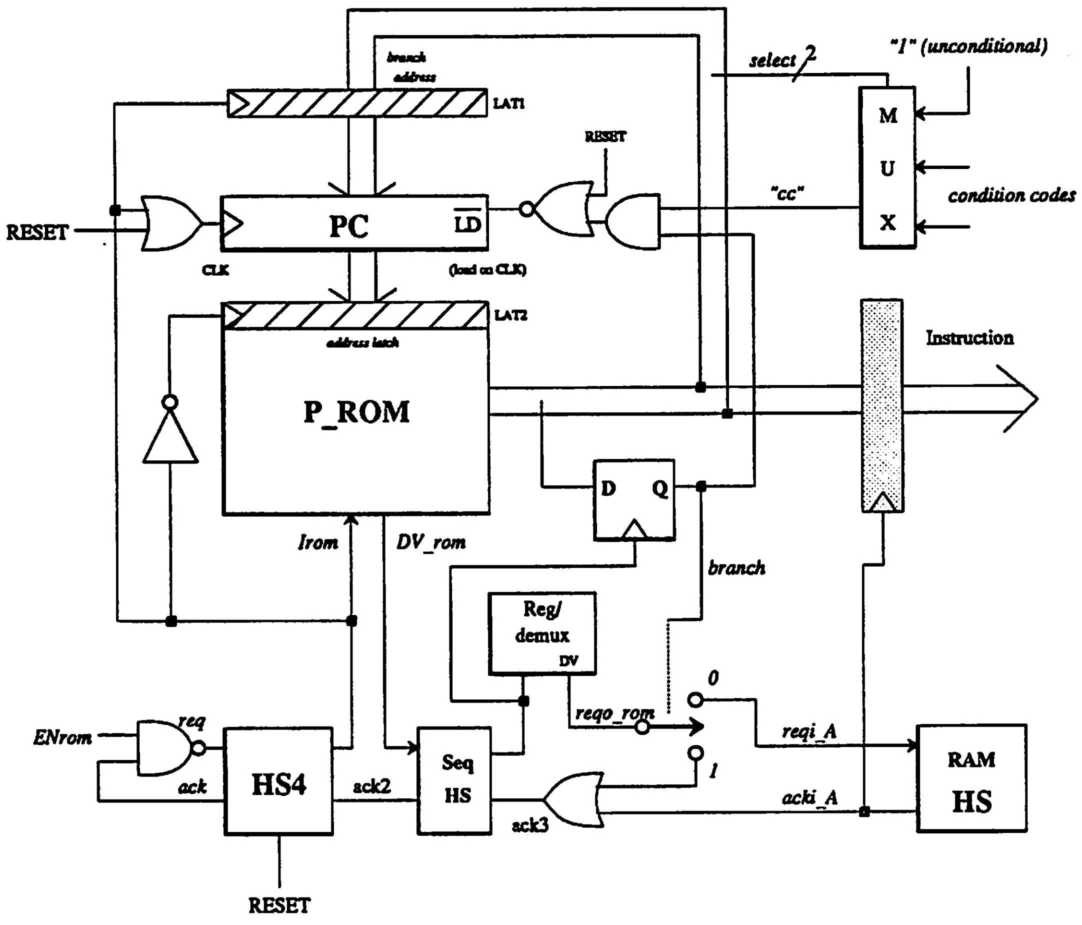


Figure 7.11: Detailed diagram of chip controller circuit.

instruction cycle. A self-timed version of the scheme would most likely contain handshaking between the controller and each datapath stage since the completion of any stage is not synchronized with a global clock. As mentioned before, the elegance of using a DS architecture is that the control signals can follow along with the data as determined by the handshaking existing inside the datapath. A difficulty arises however for control signals that actually re-configure the datapath and therefore the associated handshake circuitry. An example is shown below in Figure 7.12. In a MUX or DEMUX stage, a control signal routes the handshake signals to their proper places. Figures 4.19 and 4.20 illustrated this. The control signal in a DS type processor emerges from the instruction pipeline. If the pipeline is clocked by the same handshaking signals as the datapath as in Figure 7.12, then there is a potential race condition between the *Acki* signal and the *CTL* signal affecting the circuit state. In the figure, *Acki* may pass through the DEMUX before the *CTL* signal is able to change the DEMUX state. The synthesis of the handshake circuit constrained

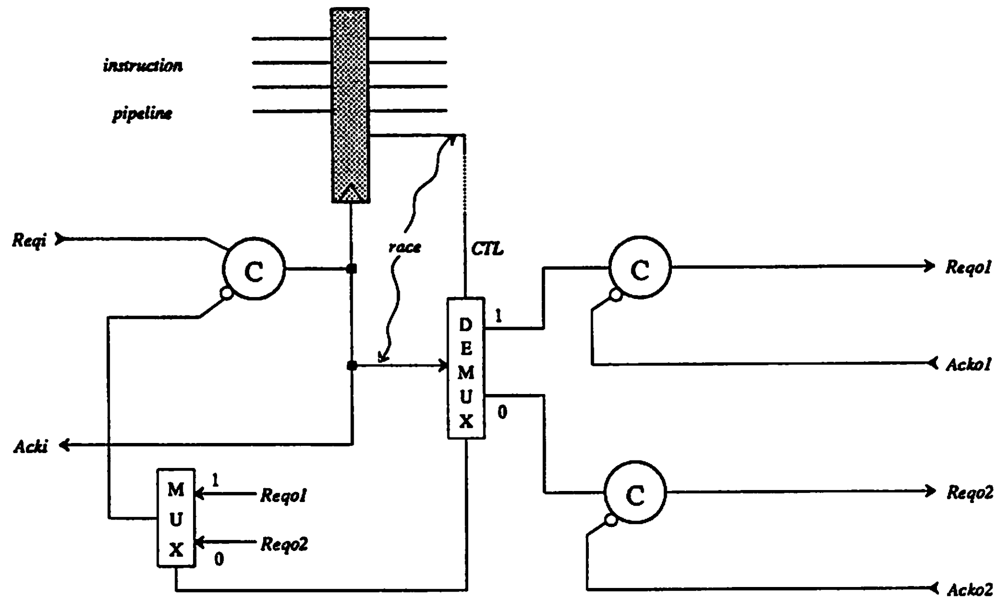


Figure 7.12: An simple example of a control signal which affects the state of a handshake circuit.

the *CTL* signal to be in a given state. The most obvious way around the race condition is to have the instruction pipeline timing separate from that of the datapath. Each stage would have its individual HS4 circuit. The register for a stage would be clocked on the

Acki signal and the *Reqo* signal (normally controlling computation) would be sent to the datapath stage that uses the control signals from the register. In that way, the datapath is ensured of receiving valid control signals from the instruction pipeline by the handshaking operation.

The overhead in hardware required to use handshaking between the instruction pipeline and the datapath is undesirable. Also it was felt that since the control signals are available in the pipeline before they are needed (i.e. they are in a register further up the pipe), enough information was there to avoid the use of controller handshaking. Thus, the DSP chip design presented in this chapter does *not* use any special handshaking for control signals. In the few cases where a requirement as shown in Figure 7.12 arises, the notion of a register completion signal is employed for the instruction register. Figure 7.13 shows the same handshaking with a small amount of extra circuitry added to alleviate the race problem. The technique relies on circuit matching by using a "dummy" register circuit delay in series with the *Acki* signal before it is supplied to the DEMUX. The *Reg delay* block must be designed so as to have enough delay to compensate for the settling time of the instruction register as well as delays associated with the signal lines between the register and handshake circuitry. This solution is only required in several places on the chip and they will surface in the description of the handshaking blocks below.

The allocation of bits in the control word for the DSP is shown in Table V. A maximum of 128 locations are allowed in the RAM and CROM with the number of bits shown. A 40-bit wide ROM is required to store the program instructions.

Since control signals are used in each stage of the datapath, a shrinking number of the entire set needs to be forwarded down the instruction pipeline after each register. In order to reduce the area and routing required for the pipeline, the register size is reduced after each stage. Figure 7.14 shows a detail of the bits for each stage of the instruction pipe.

4.3 I/O Scheme

A method for getting data into and out of the DSP chip was chosen that takes full advantage of the self-timing characteristics. The input register feeds a MUX which can select it as the *write* input to the RAM. The output register is driven from the feedback path around the datapath. Thus, as shown in Figure 7.1, it can send either data words

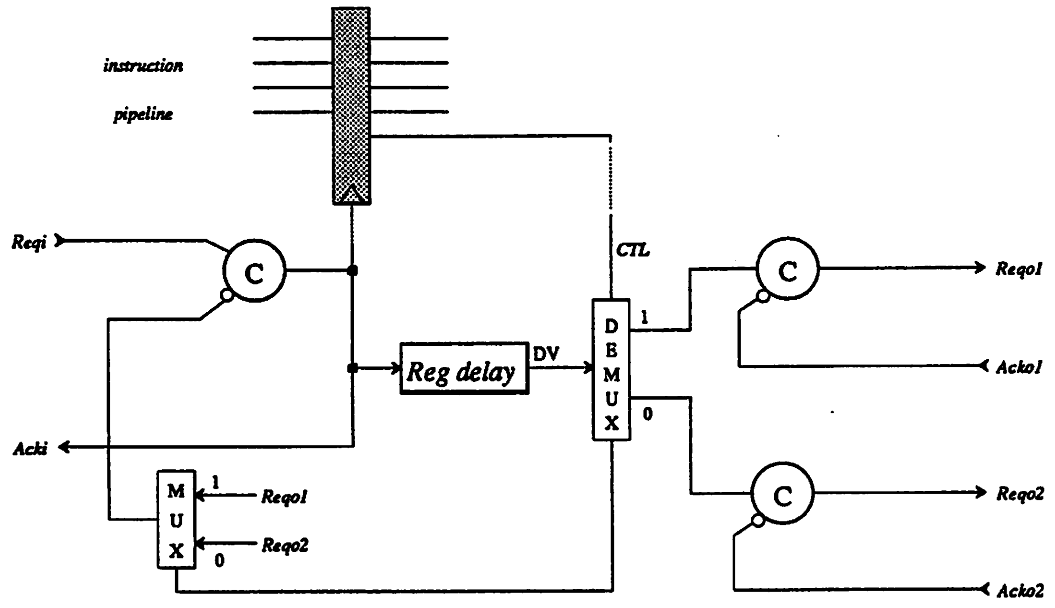


Figure 7.13: Using a register delay to ensure that a control signal is utilized properly in the handshaking circuit.

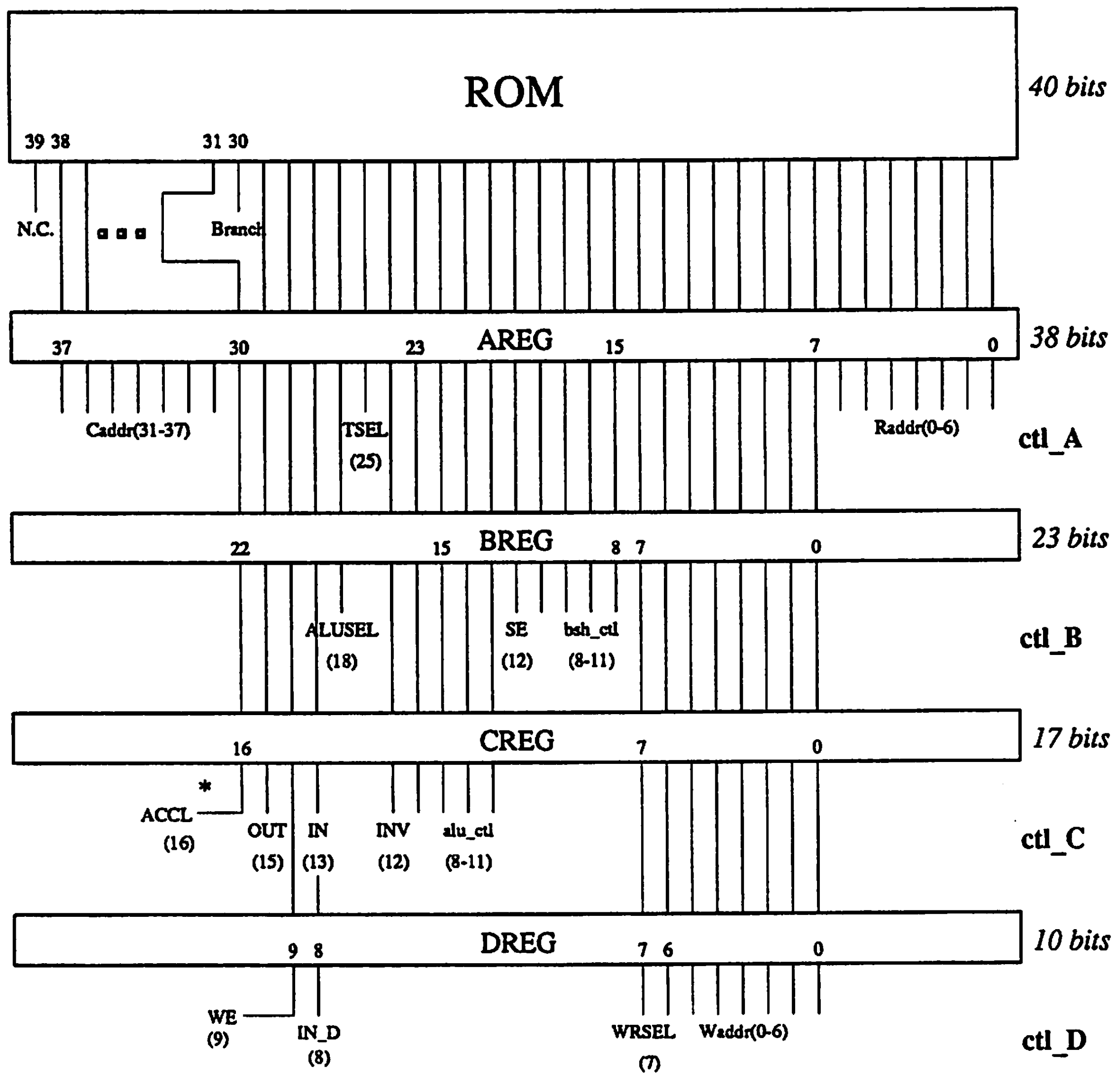
from the Accumulator or the RAM (via the 3-register delay line around the RAM) to the outside world. The times when an input word would be read or an output word would be sent are the same as a RAM operation and therefore the I/O handshaking is part of the RAM handshaking circuit. In an early design of the I/O block, a single register was employed to store the input word during the RAM operation. By designing the RAM handshaking to wait for a request from the input port during an input instruction, the timing constraints are satisfied. Similarly, the handshaking can be designed to wait for an acknowledgment signal from the outside world before proceeding during an output instruction. With this scheme however, the timing of the external signals which serve as handshaking signals for I/O operations can strongly influence the program operation. It is more desirable to have the capability of writing (reading) the output (input) register and then proceeding immediately with the program operation. While the external circuitry must consume (supply) the data word before the next I/O operation, the timing is less critical.

A FIFO type of register was substituted for the single register in each of the I/O paths to obtain the greater flexibility. The input register (*i-reg*) and the output register

Table V. Processor CONTROL WORD

Bit(s)	Mnemonic	Description:
39	NC	No-Connection
38-32	Caddr	CROM Coefficient address
31	ACCL	Select Accumulator LOW byte
30	Branch	Branch Bit
29	OUT	Enable Output Port
28	WE	Write Enable for RAM
27	IN	Enable Input Port
26	ALUSEL	Select Multiplier to ALU
25	TSEL	Select CROM to MULT Y-input
24-20	alu_ctl	ALU function select bits
19	SE	Enable sign-extension in Shifter
18-15	bsh_ctl	Shifter amount
14	WRSEL	Write RAM from Accumulator
13-7	Waddr	RAM Write Address
6-0	Raddr	RAM Read Address

Instruction Pipeline



* *ACCL delayed by one in ACC_MUX*

Figure 7.14: Detail of the instruction pipeline registers.

(*o-reg*) in the block diagram are actually two stage shift registers. The outer-most register in the FIFO is controlled by its own simple 4-cycle handshake circuit which talks to the outside world. Each of the input and output operations will now be examined in detail.

Input Instruction

On an input instruction, the input register acts as the source for data on the RAM *write* port. Therefore, as stated above, the input port handshaking must work in conjunction with the RAM handshaking. Figure 7.15 shows a simplified drawing of the input FIFO and associated handshaking. The upper two c-elements make up the RAM handshaking. When the ALU completes an operation, it asserts the *reqi_ACC* signal. Assuming that the controller has another instruction ready, the RAM handshake circuit will raise *acki_ACC*, clocking the Accumulator itself - which is normally the RAM input register. The circuit of Figure 7.15 is configured for an input instruction. Therefore, the RAM handshaking circuit must wait for a request from the input FIFO (the third input to the c-element) before proceeding. The input FIFO is actually a self-timed pipeline, the output of which feeds the RAM handshaking input. There are several timing cases to consider: If the off-chip source requests an input transfer at some point before the impending processor input instruction, the *Reqi_IN* signal is raised. Assuming the last data input was successfully transferred to the second FIFO register, then *Acki_IN* will be raised and the new data word will be stored in the first FIFO register. If the last data input is still in the first register, the FIFO is full (implying that the request is two input instructions early) and the *Acki_IN* signal remains low until there is space. When *Acki_IN* is raised, then the *reqi_INP* signal is raised to signal the RAM handshake circuit that new input data is ready. During the actual input instruction, *acki_INP* is then raised clocking the data into the second FIFO register which feeds the RAM.

In the case where the processor reaches an input instruction before there has been a request on the input port, the *acki_ACC* signal stays low in the absence of the *reqi_INP* signal. When the off-chip source finally raises *Reqi_IN*, the input data will ripple through the FIFO, raising *reqi_INP* and allowing the processor to proceed. This in turn raises *acki_ACC* and clocks the data into the second FIFO register where it is supplied to the RAM input. The FIFO arrangement allows an input generator to run out of phase with the processor input instructions which consume the input data. In the actual circuit, the configuration of the RAM handshake circuit must be changed during normal instructions so that it does not wait for input port requests before proceeding.

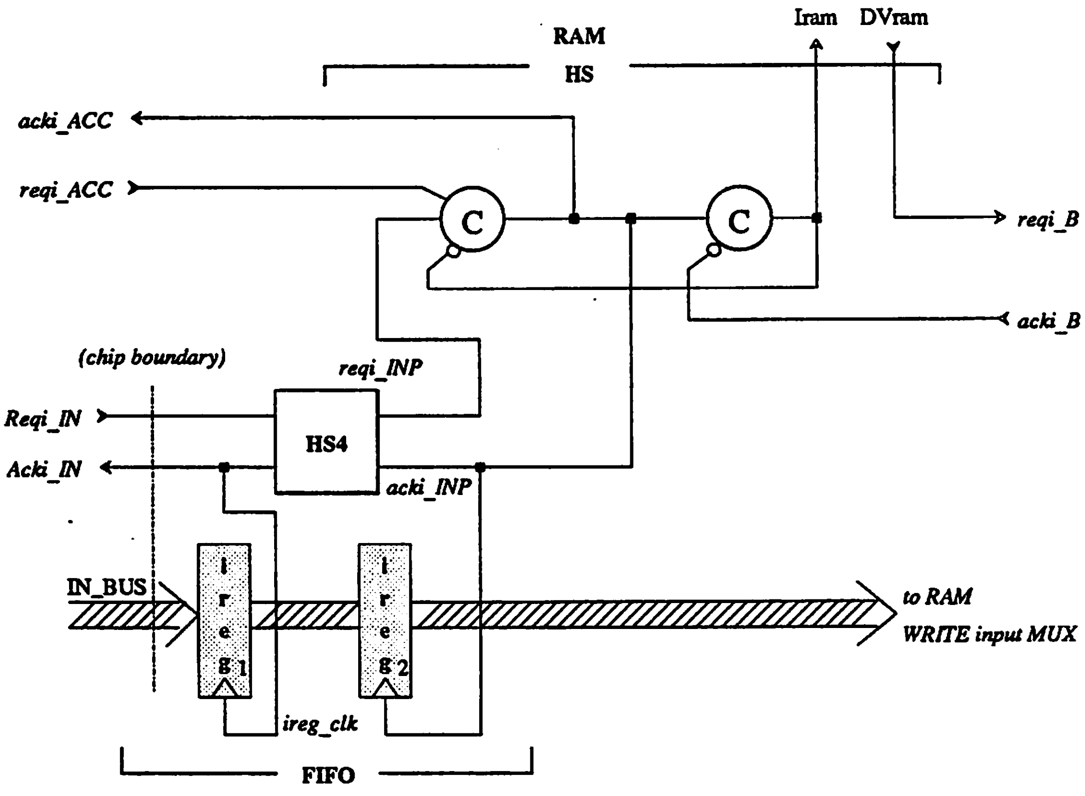


Figure 7.15: Simplified drawing of the Input Port.

Output Instruction

The output instruction timing is the dual of the input timing described above and a simplified drawing of the FIFO and associated handshaking is shown in Figure 7.16. Where the input port acts as another source to the RAM, the output port works in parallel with the RAM itself, sending out a data word during a RAM operation. The parallelism comes from the fact that the sources for writing into the RAM are the same for writing off-chip. When the *acki_ACC* signal rises at the beginning of an output instruction, valid data is ready at the inputs to the RAM and output FIFO. The right-most c-element of the RAM handshaking is then triggered to initiate a RAM operation. During an output instruction, another c-element is placed in parallel with the RAM handshaking to simultaneously trigger the output FIFO. Analogous to the *Iram* signal, the *reqo_oreg* clocks data into the first FIFO register. It also sends a request to the HS4 circuit associated with the second FIFO register. Assuming that the off-chip destination is ready to receive another data word, *Reqo_OUT* is raised and the second FIFO register is clocked. The first c-element of the RAM handshaking circuit waits for both the *Iram* signal and the *reqo_oreg* signal before proceeding, a requirement of the output port and RAM working in parallel. The skeletal circuit for the RAM handshaking during an output instruction is really identical to the handshaking circuit for a stage with two destinations as shown in Figure 4.18. As with the input port, the FIFO allows the output destination circuitry to operate out of phase with the output instructions in the DSP program.

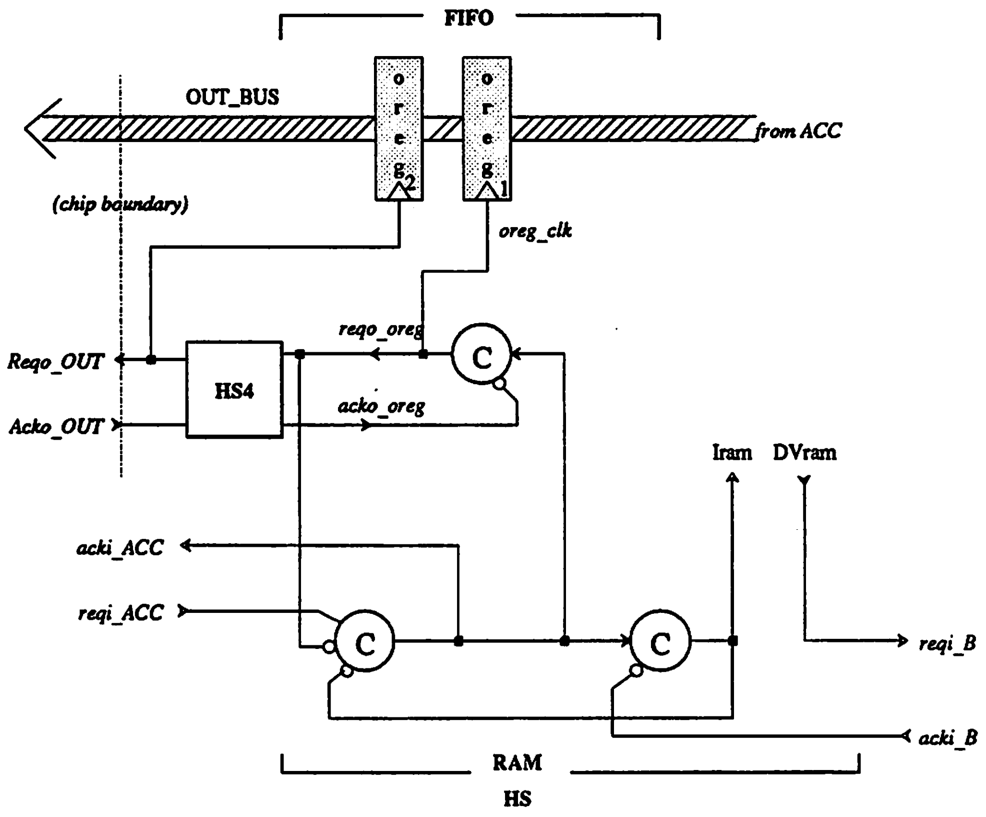


Figure 7.16: Simplified drawing of the Output Port.

FIFO Delays

There is a danger in the FIFO connections shown for the input and output ports. Along with the accompanying HS4 circuit, the input and output registers form a single stage in a self-timed pipeline. The absence of a completion signal on a simple D-register necessitates the use of some other method for ensuring that the register outputs are valid. In the DSP design, a "dummy" register (*regDV*) delay is used in several locations. This was shown in Figure 7.13 where is was used in lew of a completion signal on control signal registers.

Referring again to Figure 7.15, the input FIFO is like a self-timed stage containing no computation (DCVSL) logic. The first register is clocked on *Acki_IN* and its function is the same as the input register on any self-timed stage. In the absence of any computation, the output of the second c-element of the HS4 circuit is sent directly out as the output request rather than the **I** signal for a DCVSL circuit. Because the second register in the FIFO is clocked by the output acknowledge signal, one must ensure that the first register outputs are settled before clocking the second register. Placing a *regDV* cell in series with the *reqi_INP* signal accomplishes this.

For the output port in Figure 7.16, the output registers in the FIFO correspond to computation circuits in a normal self-timed stage. The first register $oreg_1$ works in parallel with the RAM and the signal *reqo_oreg* is its "I" signal. To ensure that its outputs are settled a *regDV* circuit is placed in series with *reqo_oreg* to generate a completion signal for the register. Finally, since $oreg_2$ is clocked by *Reqo_OUT*, and the request signals when data is valid on the output pins, another *regDV* is placed in series with it to ensure that the register outputs are settled before an off-chip device latches the data.

4.4 RAM Handshaking

As shown in the last sub-section, the I/O handshaking is integrated with the RAM (or first stage of the datapath pipeline) handshaking. The combination of the different functions makes the RAM handshake circuit (RAMHS) the most complicated in the DSP. A diagram showing the complete circuit is given in Figure 7.17. The task of properly initializing the circuitry complicates things slightly and that is discussed below. The figure combines the simplified drawings for the input and output ports. However, there is means for selecting whether the I/O circuitry is activated or not. Also, there is a

4-cycle handshake interface to the controller.

In its simplest configuration the RAMHS circuit receives a request from the controller via *reqi_A* and performs a RAM operation when the datapath is ready. The RAM is written from the feedback registers from the *read* port. Typically however, the RAM is written from the Accumulator in which case the *reqi_ACC* must be examined also. The configuration is controlled by the WRSEL2 signal in the diagram.

The OUT' signal is asserted when the output port is active and the IN' signal is asserted when the input port is active. When the signals are LOW , the processor just continues with a normal RAM operation without waiting for the ports. The method of *de-selecting* an input to a c-element is a little different than a typical logic gate. For example, to make a two-input OR gate transparent to one input, the other input is just set LOW . For a two-input c-element however, to make the output follow one input, the other input must always be the same state as the first input. Examining the input port connection, the first c-element receives *reqi_ACC*, $\overline{Iread}$, and *reqi_INP*. If the instruction is not inputting data, (IN' LOW) then a MUX switches the *reqi_INP* input to the c-element to *reqi_ACC*. That shorts two of the c-element inputs which effectively just collapses them into a single input. Similarly, on the output port side, one of the c-element inputs is driven by either $\overline{reqo\_oreg}$ or $\overline{reqi\_ACC}$ depending on the state of the OUT' signal.

RAM Input MUXes

The RAM input MUXes choose whether the RAM is written from the local feedback loop, the Accumulator, or the Input port. The location of these MUXes however, is such that it violates the model for a typical self-timed system as shown in Figure 4.1. Specifically, the MUXes are between the storage register for the stage and the computation block. The RAM is the computation block and the storage register is actually split between several locations (hence the MUXes). One assumption made for the datapath self-timed blocks is that the outputs of the the storage register for a stage are settled before the stage begins computation, as dictated by the DCVSL logic. The RAM input registers reside at various locations on the chip and they are separated from the RAM by long wiring busses and the MUXes themselves. In order to accommodate this architecture, the *acki_ACC* signal is passed through all of the RAM input MUXes so that it sees the same delay as the data from the registers. This is shown symbolically in Figure 7.17. In the actual circuit, an extra MUX cell is placed on each MUX with its two inputs shorted together so that the output follows the input in spite of the MUX control signal. The *acki_ACC* signal is routed through the cascade of extra MUX cells and the signal at the end of the chain is denoted *DVacc*.

The RAM and CROM are operated concurrently to generate both a variable and constant for the multiplier in the next pipeline stage. Therefore, the I_{read} and I_{crom} signals are identical. The WE signal enables the write operation in the RAM (It is always read). To form the completion signal for the stage, the DV signals from both the RAM and CROM are fed to a c-element.

Initialization

The existence of both pipelining and feedback in the datapath requires that initialization of the handshaking circuits be performed in a special way. In the section describing different c-element designs, it was stated that the ability to either *set* or *reset* a c-element output was desirable for initialization. The ability to set a c-element output HIGH or LOW during Reset, allows the designer to place the set of handshaking stages in a processor into a specific state. For a simple asynchronous pipeline, it is normally sufficient to just clear all of the c-elements. Upon start-up, the pipeline will get filled - the later stages just waiting for an input request before doing anything. In the DSP design presented in this chapter however, there is a feedback path from the Accumulator to the RAM. If the initial state chosen for all of the handshake stages was such that they were all cleared, then the operation of the chip would get "stuck" at the RAM because it would wait for a request from the Accumulator before proceeding (and there is no data that far down in the pipeline upon initialization). The proper initialization involves selecting a state that prevents the lockup. By initially setting *acknowledge* signals HIGH in stages of the pipeline that follow the RAM, the data will fill the pipeline normally.

In the early stages of the design of the DSP, a full set of c-elements that could be initialized HIGH or LOW was not available. Therefore, a different but equivalent approach was taken for performing the correct initialization of the datapath handshaking circuits. Referring again to Figure 7.17, the circuit configuration was actually made alterable. Rather than setting a specific state upon initialization, all of the c-elements on the chip are cleared with the Reset pulse. In the schematic, the register generating WRSEL' is also

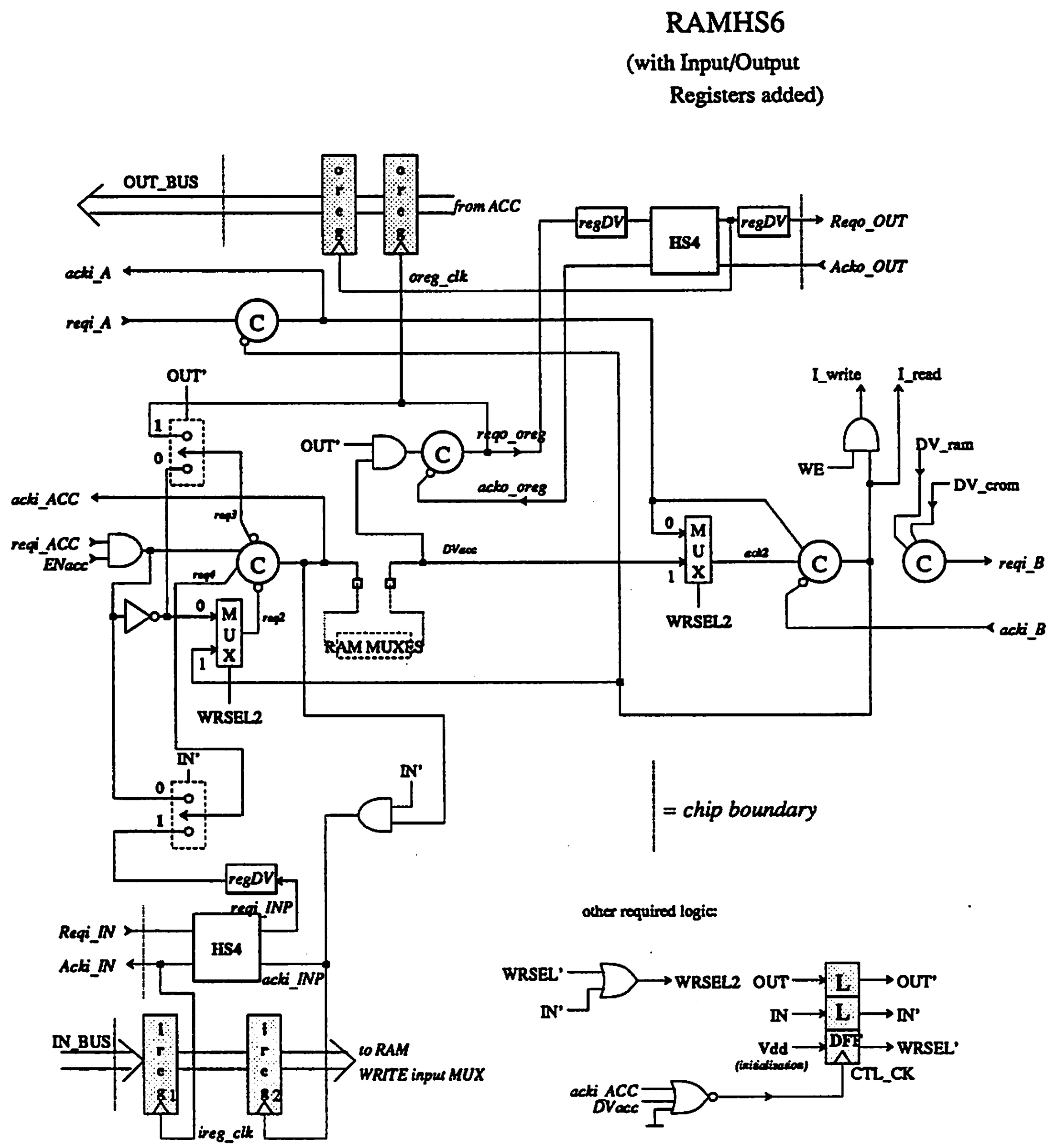


Figure 7.17: Complete drawing of the RAM and I/O handshaking. The I/O registers are included for clarity.

cleared. This in turn sets WRSEL2 LOW which changes the position of two MUXes in the circuit. When WRSEL2 is LOW , the RAM handshake circuits ignores requests from the Accumulator (remember, shorting two c-element inputs collapses them into a single input) and also allows the *ackiACC* signal to follow the *reqiACC* signal. After the first handshake cycle in the Accumulator stage, the WRSEL2 signal gets set HIGH and remains that way until the chip is powered down. With WRSEL2 HIGH , the circuit is configured normally as explained in the sections above.

Control Signal Interface

The absence of handshaking between the instruction pipeline and the datapath handshake circuits requires the control signals to be interfaced more carefully. The IN and OUT control signals shown in Figure 7.17 determine the configuration of the circuit for input and output instructions. For the handshaking to operate correctly, the circuit must be configured *between* instructions. If the IN and OUT signals are taken from **ctl_D** in the instruction pipeline (clocked by *ackiACC*), the re-configuration would be too late since the RAM handshake circuit would have already received an input request. The IN,OUT signals are taken therefore from the preceding stage, **ctl_C** in the pipeline. Taken from there however, they could change state while the RAM handshaking is still active in a handshake cycle. Therefore, several latches are added as shown at the bottom of the schematic. The IN,OUT signals are fed to latches which are clocked by CTL_CK which rises at the end a any given handshake cycle in the RAM. This occurs when both *ackiACC* and *DVacc* have gone LOW . by strobing the control signals at a time between handshake cycles, the circuit configuration can be changed without causing any spurious signal generation.

Note that the OR gate generating WRSEL2 in the schematic is a holdover from a previous design and while it exists in the circuit, its function is extraneous since WRSEL' is always HIGH after initialization.

4.5 Multiplier,Shifter Handshaking

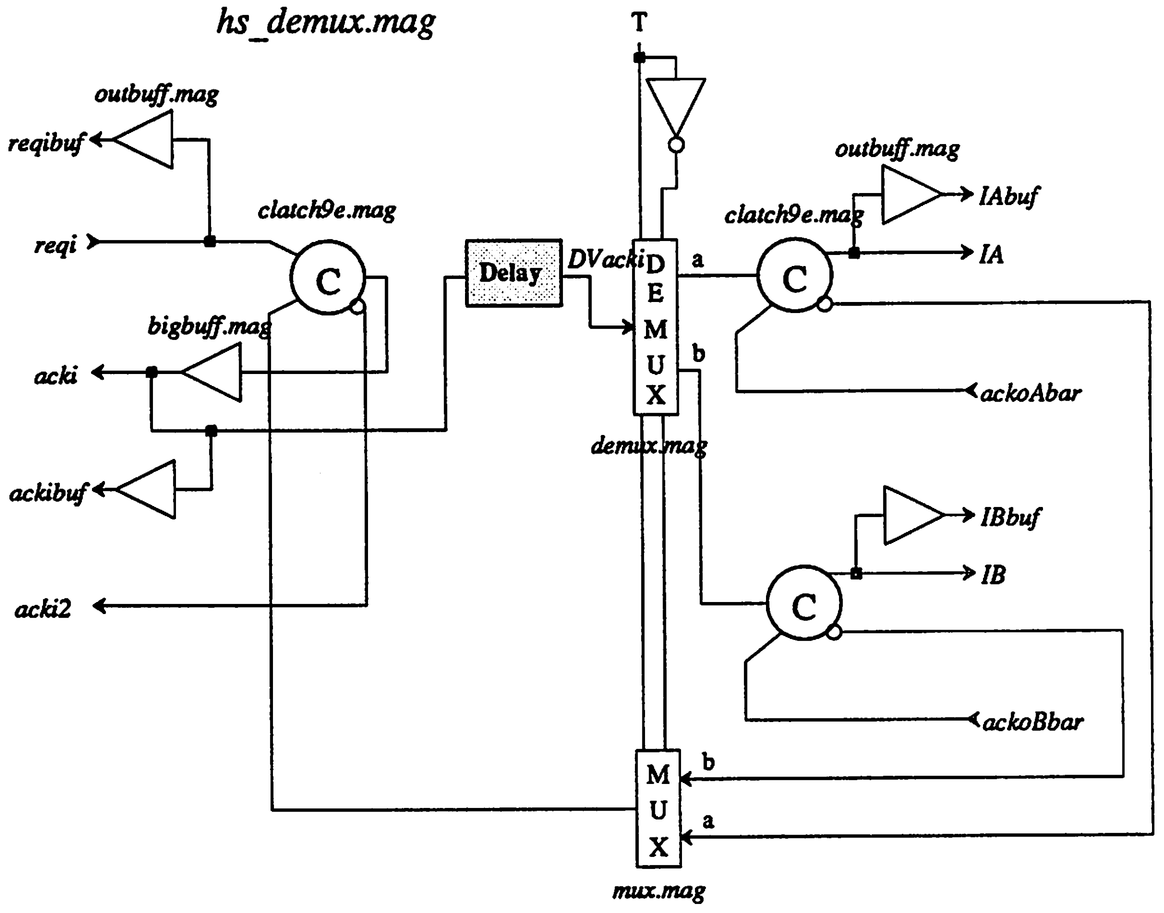
The second stage of the datapath pipeline contains the multiplier and barrel shifter. Either one of the cells may be used during and instruction but not both. Since one of two destinations is chosen from a single source (the RAM and CROM), the handshaking circuit is essentially that for a demultiplexer stage as shown in Figure 4.20. The control signal T in the figure determines which destination is chosen and in effect, it re-configures the handshake circuit into one of two states. When T is HIGH , the multiplier is selected. A more detailed schematic of the actual circuit (*hs_demux.mag*) is shown in Figure 7.18. MAGIC subcell names are shown. Signal names ending in "buf", are just buffered outputs that are sent out to pads for testing. They are not integral to the functioning of the DSP. The table below maps the signal label names in the handshaking cell to the names used in the block diagram of the DSP.

hs_demux label	Corresponding Signal
reqi	*reqi_B*
acki	*acki_B*
T	$ALUSEL$
IA	$Ibsh$
IB	$Imult$
ackoAbar	*acki_C1bar*
ackoBbar	*acki_C2bar*
acki2	clock for ctl_B reg.

The delay in the *hs_demux* circuit is necessary again because there is no handshaking with the controller. The *acki* signal of the circuit clocks the B-register the output of which is the ALUSEL signal that determines the state of the MUXes in the latter part of the handshake circuit. Since the control register generates no completion signal, the delay ensures that the state of the MUXes is determined before the request propagates to them.

4.6 ALU Handshaking

After the data is split and applied to either the barrel shifter or the multiplier, it is sent to the ALU. The corresponding handshake circuit must deal with requests from either the shifter or multiplier depending on the value of the ALUSEL control signal. Similarly, the data outputs from the shifter and multiplier are sent to a 2-input MUX, the output of which feeds the ALU input. Nominally, the 2-input MUX handshaking should be used here and it was shown in Figure 4.19. In the actual implementation, a question arises about what node should be interpreted as the *Acki* signal. Having two sources of requests, there are two individual acknowledge signals. The output of the MUX feeding



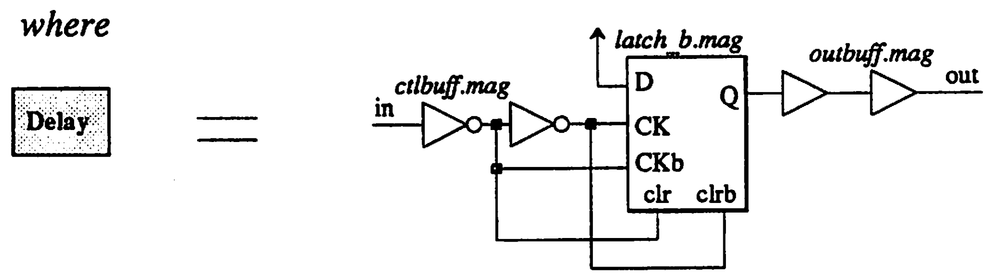


Figure 7.18: Detailed drawing of the handshake circuit used for the multiplier and barrel shifter.

the second c-element and parenthetically labeled *acki* in Figure 4.19 was originally chosen as the acknowledge signal for the ALU stage, clocking the control C-register.

The absence of handshaking for the control signals again caused a problem to arise in the ALU handshaking circuit. When the circuit of Figure 4.19 was employed and the *acki* signal taken from the node described above, there were situations where a double acknowledgment pulse was generated. The T signal (ALUSEL) was taken from the output of the B-register in the control pipeline. This was done so that its state would be determined and the MUXes in the handshaking circuit settled before requests came to the input. In the case where the *acko* signal is delayed for some reason (such as during an input or output instruction), an input request will be queued in the MUX handshaking circuit. After the delay time, there can be an *acki* pulse, followed by a change in the ALUSEL signal, followed by a duplicate *acki* pulse. The effect comes from changing the state of the MUXes in the handshaking circuit at the incorrect time. If the ALUSEL signal was controlled by handshaking, then the MUXes could be set up properly to avoid the problem, although the efficiency of the circuit would suffer because ALUSEL would have to settle before any datapath handshaking could occur.

It was observed that the *de-selected* cell in the second stage of the datapath (either the shifter or multiplier), sits in pre-charge state while waiting for its next operation. This translates to having a LOW DV signal and therefore no request to the ALU handshaking. Under the condition that only a single request of the pair is HIGH at any time, the ALU handshaking circuit can be simplified. The detailed diagram of the actual circuitry used is shown in Figure 7.19. Again, signal names ending in "buf", are just buffered outputs that are sent out to pads for testing. Rather than having separate c-elements for each request form the preceding stage, the two are collapsed into a single c-element having the input request multiplexed. The two *acki* signals are now the same node and there is no ambiguity in its generation or labeling. Note that the simplification depends on having only one of the input requests cycling at a time. The original circuit allowed the de-selected request to cycle by making its c-element transparent. The actual circuit is closer to a simple HS4 circuit with two requests multiplexed at the input. The table below maps the signal label names in the handshaking cell to the names used in the block diagram of the DSP.

hs_mux label	Corresponding Signal
reqiA	*reqi_C*1 (DVbsh)
reqiB	*reqi_C*2 (DVmult)
ackiAbar	*acki_C*1*bar*
ackiBbar	*acki_C*2*bar*
acki	*acki_alu*
T	*ALUSEL*
I	*Ialu*
acko	*acki_ACC*

For clarification, a diagram showing the combined handshaking circuitry of the shifter, multiplier and ALU is shown in Figure 7.20 with the global signal labels.

5. Global Placement and Routing

The construction of the DSP chip contains macrocells as described in Chapter 6 along with the handshaking circuits discussed above. Each handshake circuit was made as a separate cell and the global routing and placement was performed with interactive floorplanning tool of the LAGER system [72, 73, 74]. Chip size not being an large factor in the experimental DSP design, little attention was paid to doing efficient floorplanning. In fact, there was some interest in having a poor floorplan in order to exploit the self-timed behavior and elimination of clock signals that might be corrupted in a bad layout. While the handshake signal wires were controlled more carefully (to avoid extra loading that can occur when signal wires are routed completely automatically), no other attention was paid to wire lengths or cell placement other than to avoid a gross chip size.

The first trial layout indicated that a layout problem was present. In Chapter 4, an assumption was made that the *Acki* signal, which clocks the data input register to a self-timed stage, caused the data at the output of the register to be valid before the corresponding I signal of that stage went HIGH to allow the DCVSL logic to evaluate. The placement of the handshaking circuits was done in a way that they were in proximity to the datapath macrocells that they control, however the original floorplan had all of the control signal pipeline registers in one section that was far from the datapath. In the DSP design, the same acknowledgment signal that clocks the data registers between stages clocks the instruction pipeline registers. The assumption above was *not* adhered to in the original floorplan because the delay of the long wires between the handshake circuits and the instruction pipeline registers caused the control signals to arrive late. Again, the use

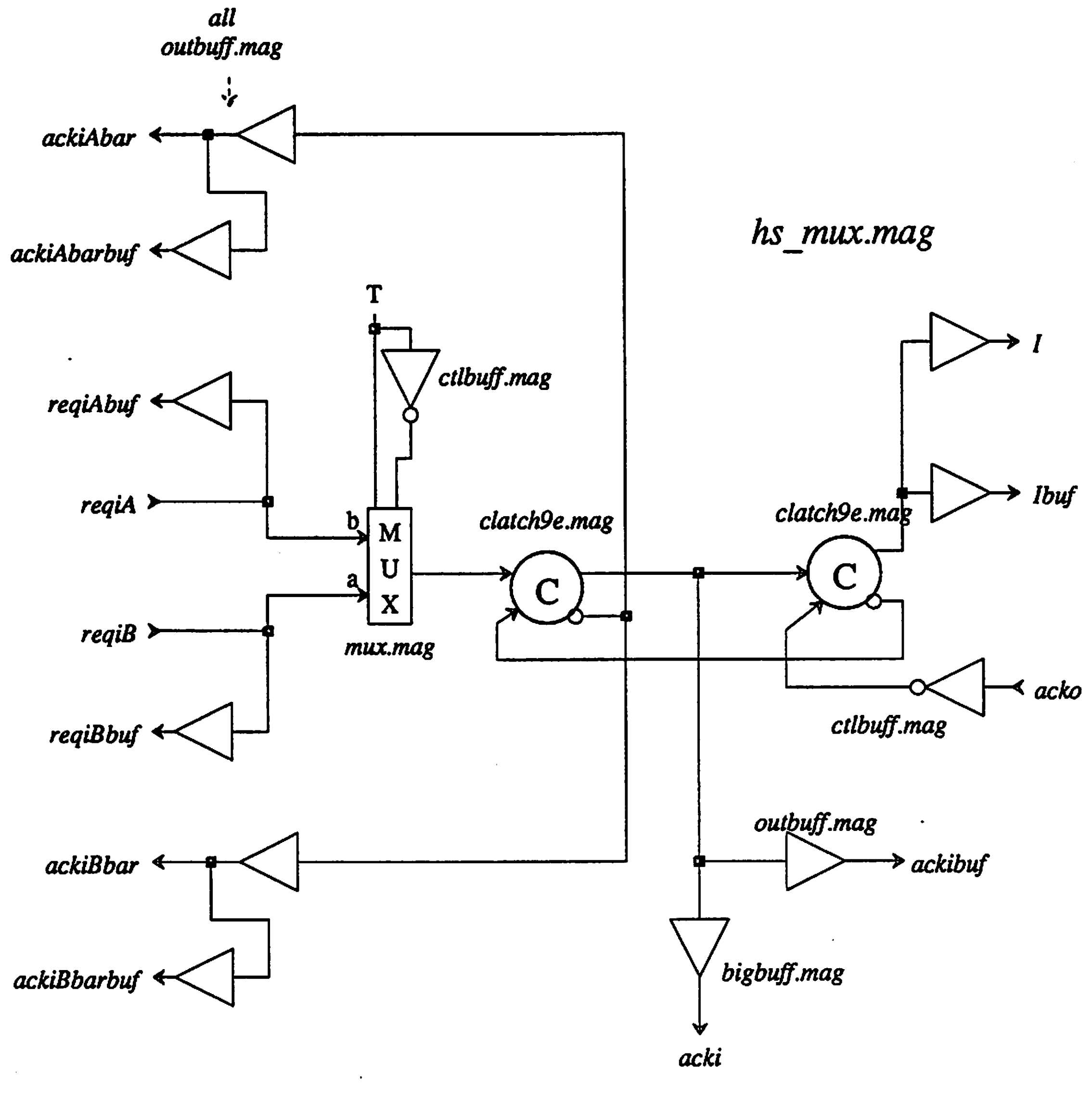


Figure 7.19: Detailed drawing of the handshake circuit used for the ALU.

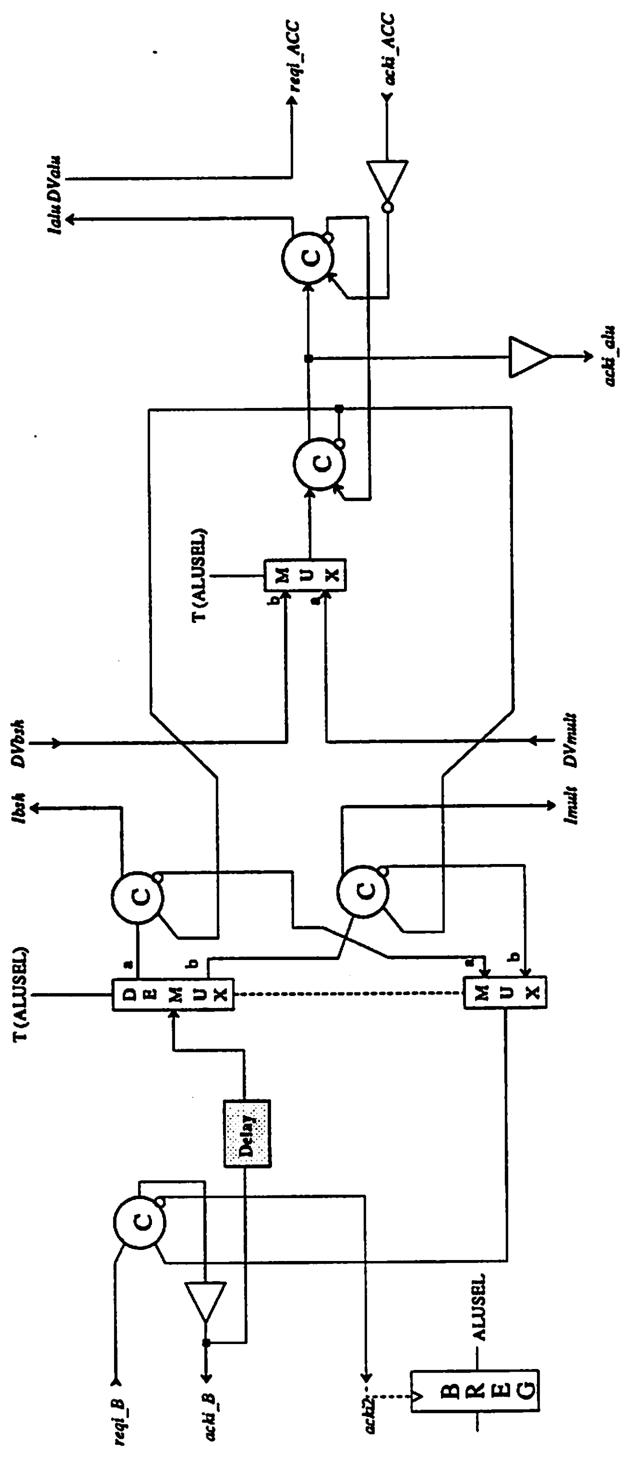


Figure 7.20: Combined handshake circuits for the second and third stages of the datapath pipeline.

of handshaking between the control registers and the datapath cells would have eliminated the problem although the long delays would degrade the overall efficiency of the circuit.

The interactive nature of the floorplanning tool along with the automatic routing of the system allowed a change to the original floorplan within hours of observing the problem. It did point out however that assumptions made about the operation of a self-timed circuit must be followed at all levels of the circuit and layout of the chip. In the second and final floorplan, the control registers were distributed around the chip so that they were in closer proximity to the actual datapath macrocells. Because the data input registers of each stage are built into the datapath macrocells, there was no problem meeting the assumption on their settling in time.

The RAM *write*-input MUXes are each a separate (tilable) macrocell as are the registers for the instruction pipeline and RAM feedback path. The overall chip area would have been smaller if these had been assembled as a datapath to cut down on the large amount of routing required between them. The floorplan of the DSP chip is shown in Figure 7.21. A micro-photograph of the actual chip is shown in Figure 7.22.

6. Chip Level Simulation

The system level simulations for the DSP chip took place in two parts. Before transistor level design began and during its early stages, the THOR behavioral simulator was used to perform timing analysis and verification on the self-timed DSP. A model was written for each of the datapath macrocells and for smaller elements making up the handshaking circuits - such as c-elements,MUXes,gates, etc. The THOR simulator models the latency of a cell as a bulk delay at the output of the cell. This was not accurate for the macrocells because the *pre-charge* time for a cell is typically much shorter than the *evaluate* time. A small modification to the program was made so that the *self-shed()* function could be used for any model. Typically used for a generator of some sort, the function allows the model to be placed into the event schedule so that it is called again after a certain number of time units. Using the function and several state variables in the model that served as flags, different pre-charge and evaluate delays could be implemented. The THOR simulations were able to verify the basic timing of the system however, the modeling was not accurately representing any of the wiring delays of an actual layout and hence it did not catch some of the problems encountered at a later stage such as the

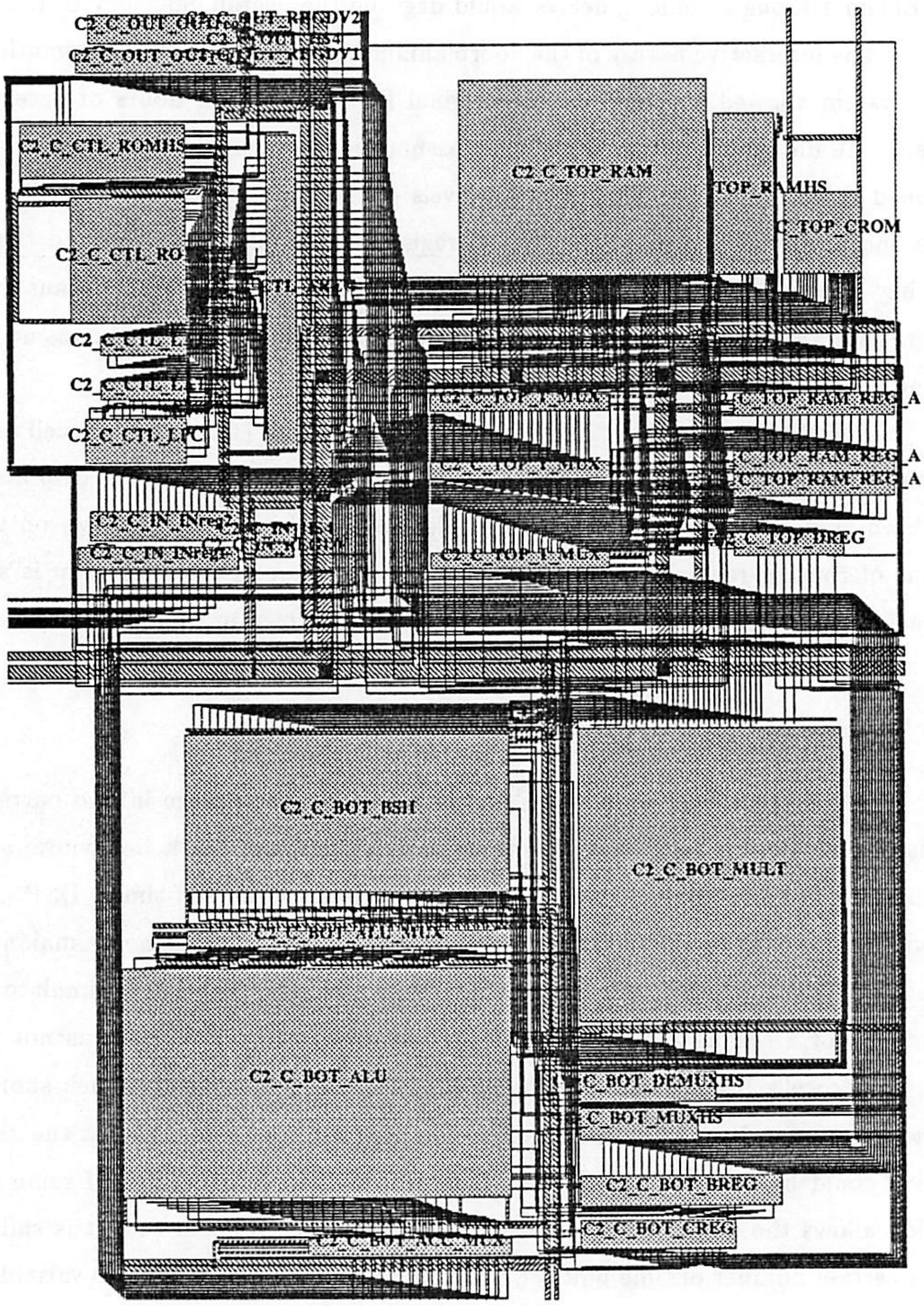


Figure 7.21: Plot of the DSP chip floorplan.

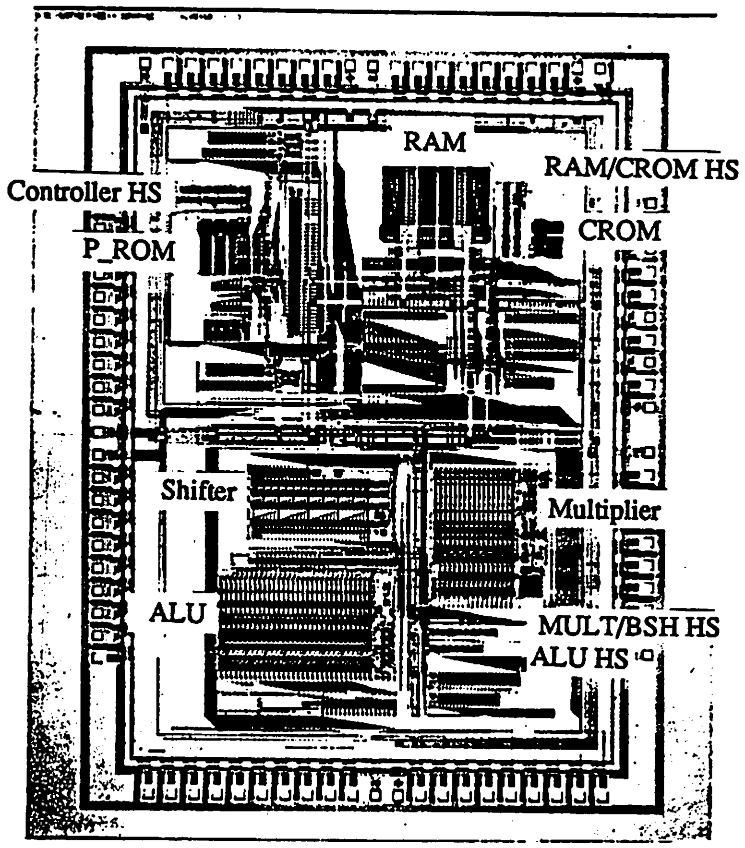


Figure 7.22: Micro-photograph of the DSP chip.

proximity problem of the instruction pipeline registers mentioned in the last section.

After layout, the IRSIM simulator was employed for chip level simulations. This simulator uses a simple RC model for each transistor and includes wiring delays. It should be stated that due to the existence of feedback in self-timed handshaking circuits, some switch level simulators are unusable. Without timing information, a simulator will iterate forever trying to reach a settled circuit state in this type of circuitry. Results from some of the simulations are compared to measured results in the next chapter.

7. Summary

Using the self-timed datapath macrocells described in Chapter 6, and handshaking realized with the methods presented in Chapter 4, a complete signal processor was constructed. The processor contains three levels of pipelining, each controlled by a separate handshake circuit. The Data Stationary architecture was employed because it more closely fits the self-timed paradigm where control signals move along with the data. The instruction pipeline was clocked by the same acknowledge signals as the data registers in the datapath, however there was no completion information generated by the instruction registers. The absence of handshaking between the control signal registers and the datapath cells caused several timing problems to appear, however they were eliminated by the use of several extra delay circuits and the matching characteristics of the IC. An interactive floorplanner was used to do the final assembly of the cells and perform routing. Chip level simulations were performed on the extracted chip with an event driven simulator which also is able to model the delay of each device using a simple RC model.

Chapter 8

Experimental Results

In the last chapter, the design of a complete asynchronous programmable Digital Signal Processor chip was presented. The chip can be programmed via the internal program ROM masks. By using the assembler described, several test programs were developed. In order to observe experimental results, a total of three versions of the DSP chip were fabricated, each with a different program and functionality. This chapter discusses the functionality of the three programs and presents experimental results from the actual chips.

1. DSP Test Chips

Each of the three test chips described below differ only in the program ROM contents. Two of the chips implement filtering functions, while the third chip runs through an "exerciser" program which tests all of the different datapath and control functions of the DSP. The coefficient CROM contents were specified so as to contain the constants necessary for all three chips.

1.1 CHIP1

The program contained in CHIP1 implements a simple 16-tap FIR filter as shown in the z-domain network in Figure 8.1. The filter was designed with an equal-ripple lowpass response where the passband ripple is $0.25dB$, the stopband rejection is nominally $-40dB$, and the transition band is between $0.175f_s$ and $0.300f_s$. The FILSYN program was utilized to calculate the coefficients of the filter as shown below. (A $20kHz$ sample rate was nominally chosen.)

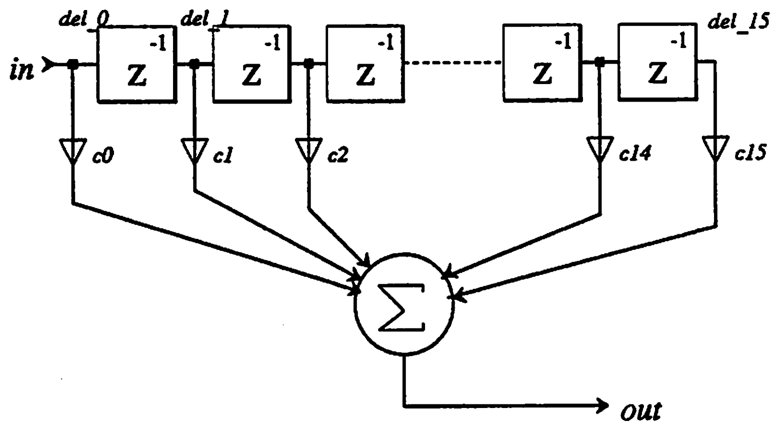


Figure 8.1: Network for 16-tap FIR filter implemented in CHIP1.

```
************************************************************************
        finite impulse response (fir)
        linear phase digital filter design

  LowPass Filter Design for CHIP1 testchip

        remez exchange algorithm
        bandpass filter
        filter length =  15
        filter length determined by approximation
        ***** impulse response *****

      sampling rate = 2.0000000d+04 hz

               function no. 0

          decimal                       octal

 h(  1) = -9.8465988d-03 =  h( 15) = -0.005025164100
 h(  2) =  1.5204853d-02 =  h( 14) =  0.007621670700
 h(  3) =  3.5358451d-02 =  h( 13) =  0.022065003000
 h(  4) = -1.8399691d-02 =  h( 12) = -0.011327274600
 h(  5) = -8.4882788d-02 =  h( 11) = -0.053353407000
 h(  6) =  2.5237037d-02 =  h( 10) =  0.014727571600
 h(  7) =  3.1088129d-01 =  h(  9) =  0.237127524000
 h(  8) =  4.7269809d-01 =  h(  8) =  0.362012760000

wish to see this plotted: y/n
> n
                     band  1       band  2          band
 lower band edge   0.              0.300000012
 upper band edge   0.174999997     0.500000000
 desired value     1.000000000     0.
 weighting         1.000000000     1.439011693
 deviation         0.019803245     0.013761698
 deviation in db   0.344062835   -37.226558685

************************************************************************
```

The filter synthesis produced a 15-tap filter and the impulse response is symmetrical to achieve linear phase. When the filter was coded in DSP assembly code, a mistake was made where the number of taps was actually written to be 16, with the 8th coefficient repeated. The coefficients were truncated to 16-bits so that they would fit into the CROM and they are shown below in 2's complement hexidecimal format:

```
Coefficients for fir filter as described in "fir_design2"
(scaled and truncated to 16bits)

 LowPass Filter Design for CHIP1 testchip

            c1 =  0xfec4  = c16
            c2 =  0x01e8  = c15
            c3 =  0x0470  = c14
            c4 =  0xfdb1  = c13
            c5 =  0xf559  = c12
            c6 =  0x032a  = c11
            c7 =  0x2705  = c10
            c8 =  0x3b54  = c9
```

The error in the frequency response caused by repeating the largest tap weight was large and the responses of the original filter (length 15) and the length 16 filter are shown in Figure 8.2. In both cases, the coefficient truncation effects from having a 16-bit wordlength in the CROM are included.

The program for the FIR filter written in DSP assembly code is shown next:

```
/*
 *   FILTER ASSEMBLY CODE FOR FIR Filter in "fir_design2"
 *      (using multiplier)
 *
 */


/*************************************************************************
Input is stored in RAM location 01.
All Coefficients are in CROM.

State Variables: Ram Location 0-15 are locations for delay line
Coefficients: CROM locations 0-7 are eight required coefficients.
Since the filter is linear phase, the impulse response is mirrored.

*************************************************************************/
DECL
        EQU     del_0   0
        EQU     del_1   1
        EQU     del_2   2
        EQU     del_3   3
        EQU     del_4   4
        EQU     del_5   5
        EQU     del_6   6
        EQU     del_7   7
        EQU     del_8   8
        EQU     del_9   9
        EQU     del_10  10
        EQU     del_11  11
```

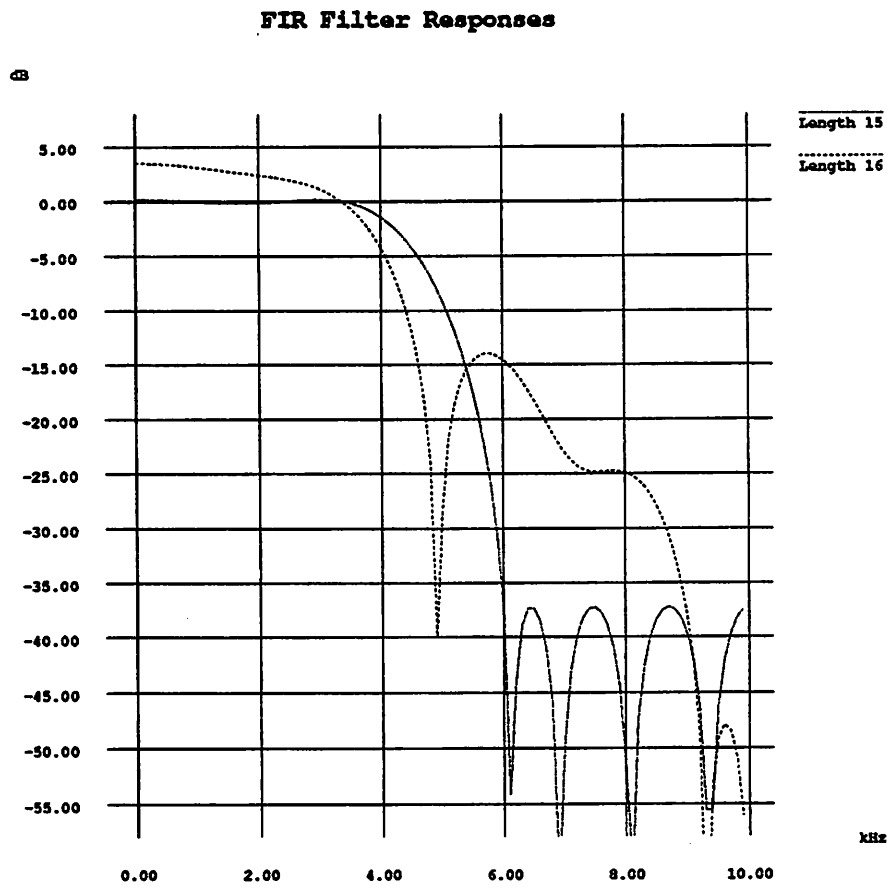


Figure 8.2: Frequency Responses of the original design 15-tap FIR filter and the 16-tap version after coefficient truncation to 16 bits.

```
          EQU     del_12  12
          EQU     del_13  13
          EQU     del_14  14
          EQU     del_15  15
          EQU     c1      0
          EQU     c2      1
          EQU     c3      2
          EQU     c4      3
          EQU     c5      4
          EQU     c6      5
          EQU     c7      6
          EQU     c8      7
START

init:     acc = 0;
          (del_0=acc) = 0;
          (del_1=acc) = 0;
          (del_2=acc) = 0;
          (del_3=acc) = 0;
          (del_4=acc) = 0;
          (del_5=acc) = 0;
          (del_6=acc) = 0;
          (del_7=acc) = 0;
          (del_8=acc) = 0;
          (del_9=acc) = 0;
          (del_10=acc) = 0;
          (del_11=acc) = 0;
          (del_12=acc) = 0;
          (del_13=acc) = 0;
          (del_14=acc) = 0;
          (del_15=acc) = 0;
          acc = acc;
          acc = acc;

sample:   acc = 0 | del_15*c1, (del_0 = in);
          acc = acc + (del_15=del_14)*c2;
          acc = acc + (del_14=del_13)*c3;
          acc = acc + (del_13=del_12)*c4;
          acc = acc + (del_12=del_11)*c5;
          acc = acc + (del_11=del_10)*c6;
          acc = acc + (del_10=del_9)*c7;
          acc = acc + (del_9=del_8)*c8;
          acc = acc + (del_8=del_7)*c8;
          acc = acc + (del_7=del_6)*c7;
          acc = acc + (del_6=del_5)*c6;
          acc = acc + (del_5=del_4)*c5;
          acc = acc + (del_4=del_3)*c4;
          acc = acc + (del_3=del_2)*c3;
          acc = acc + (del_2=del_1)*c2;
          acc = acc + (del_1=del_0)*c1;
          (out=acc) = acc;
          goto sample;
```

Each instruction in the core of the program implements a multiplication, addition, and data move in order to do a single tap of the filter. The code above the label "sample:" performs initialization by clearing the delay line. It is executed only a once. The last instruction of the loop is a NOP provided just to send the Accumulator value to the output port.

1.2 CHIP2

The program contained in CHIP2 implements a 8-pole bandpass IIR filter as shown in the z-domain network in Figure 8.3. The filter was designed with an equal-ripple bandpass response where the passband ripple is $0.25dB$, the stopband rejection is nominally $-30dB$, the upper transition band is between $0.15f_s$ and $0.225f_s$, and the lower transition band is from $0.0025f_s$ to $0.015f_s$. The FILSYN program was used to calculate the poles/zeros of the filter as shown below. (A $20kHz$ sample rate was nominally chosen)

```
 general filter synthesis program

 IIR Bandpass Filter Design for CHIP2  (0.25dB ripple)
    band-pass filter
       equal ripple pass band
          bandedge loss                           =  0.2500 db.
          preshifted lower passband edge frequency =  2.7766309d+02 hz.
          lower passband edge frequency            =  3.0000000d+02 hz.
          upper passband edge frequency            =  3.0000000d+03 hz.
          sampling frequency                       =  2.0000000d+04 hz.
       equal minima stop band with edge frequency  =  1.3875444d+02 hz.
       equal minima stop band with edge frequency  =  5.0286832d+03 hz.
          required stop band loss                  =   30.00 db.
          multiplicity of zero at zero             =  0
          multiplicity of zero at infinity         =  0
          number of finite transmission zero pairs =  4
          overall filter degree                    =  8
>
 command:
> scal
 12: 1 or 100: 2 scaling
> 1
 command:
> trun
 do you want truncation: 0 or rounding: 1
> 1
 enter # of significant bits
> 16
 command:
> pri

 digital filter transfer function
 IIR Bandpass Filter Design for CHIP2  (0.25dB ripple)
    (bilinear z transform used)
    filter type : bandpass
       sampling frequency       =  2.0000000d+04 hz.
       passband edge frequency  =  3.0000000d+02 hz.
       passband edge frequency  =  3.0000000d+03 hz.

 h(z) in factored form. coefficients of z**(-1) and z**(-2) printed

 **** numerator ****    multiplier =  5.6539917d-01

             -1.9030762d-01       1.0000000d+00
             -1.9994202d+00       1.0000000d+00
             -1.9972076d+00       1.0000000d+00
              1.2055817d+00       1.0000000d+00

 **** denominator ****

             -1.1237183d+00       4.5388794d-01
             -9.9107361d-01       7.9769897d-01
             -1.8367157d+00       8.5334778d-01
             -1.9683533d+00       9.7622681d-01
```

```
scale factors

  3.4770203d-01
  9.4448853d-01
  9.4606018d-01
  1.9398499d-01
```

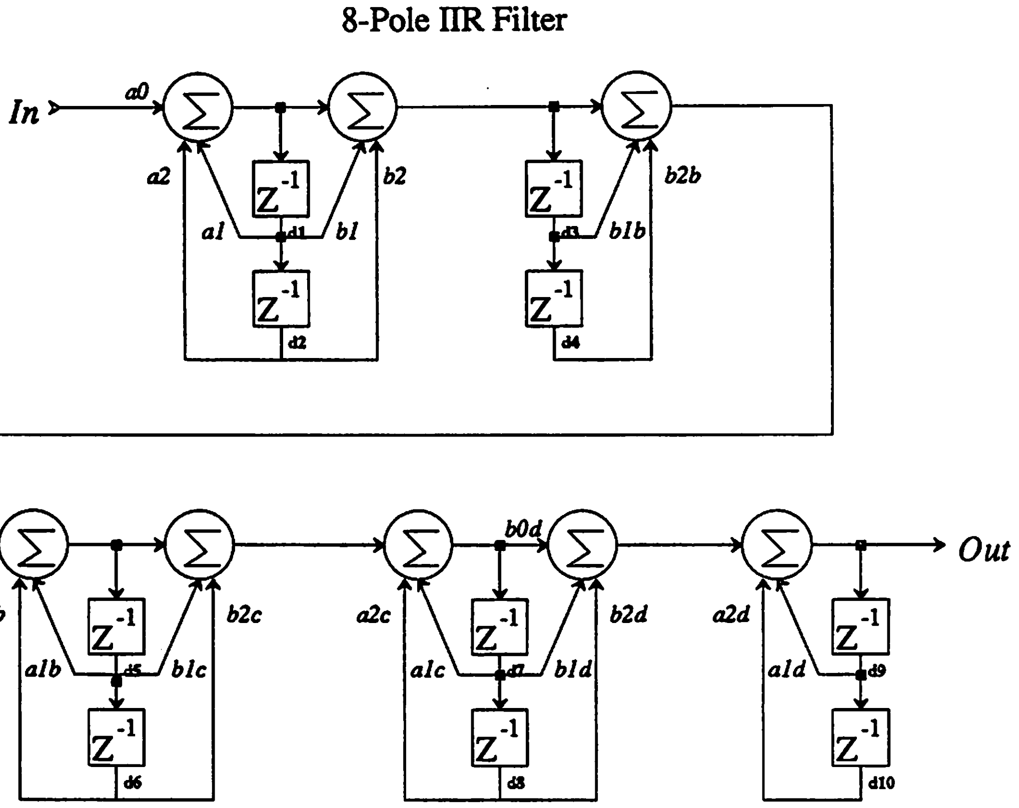


Figure 8.3: Network for 8-pole IIR filter implemented in CHIP1.

The exact coefficients used for the filter are shown in Table VI. Coefficients greater in magnitude than 1.0 are implemented as an addition plus the multiplication by the fractional part. Binary values of "n.a." mean that the coefficient can be implemented exactly with a single addition and shift (using the barrel shifter).

The frequency response of the IIR filter was calculated using a digital network analysis program and the result is shown in Figure 8.4. The passband of the filter is shown in further detail in Figure 8.5.

The assembly code for the IIR filter was given as an example in Chapter 7 in the section describing the assembler (Section 7.1.3). Some further scaling and re-arrangement

Table VI Coefficients for IIR Filter on CHIP2

Stage	Coefficient	Decimal	16-bit Binary
1	a0	0.50000	n.a.
1	a1	1.12371	1 + 0x0fd6
1	a2	-0.45389	0xc5e8
1	b1	-0.19031	0xe7a5
1	b2	1.00000	n.a.
2	b1b	-1.99942	-1 - 0x8013
2	b2b	1.00000	n.a.
2	a0b	0.50000	n.a.
2	a1b	0.99107	0x7edb
2	a2b	-0.79770	0x99e6
3	b1c	-1.99721	-1 - 0x805c
3	b2c	1.00000	n.a.
3	a1c	1.83672	1 + 0x6b19
3	a2c	-0.85335	0x92c6
4	b0d	0.25000	n.a.
4	b1d	0.30140	0x2694
4	b2d	0.25000	n.a.
4	a1d	1.96835	1 + 0x7bf3
4	a2d	-0.97623	0x8306

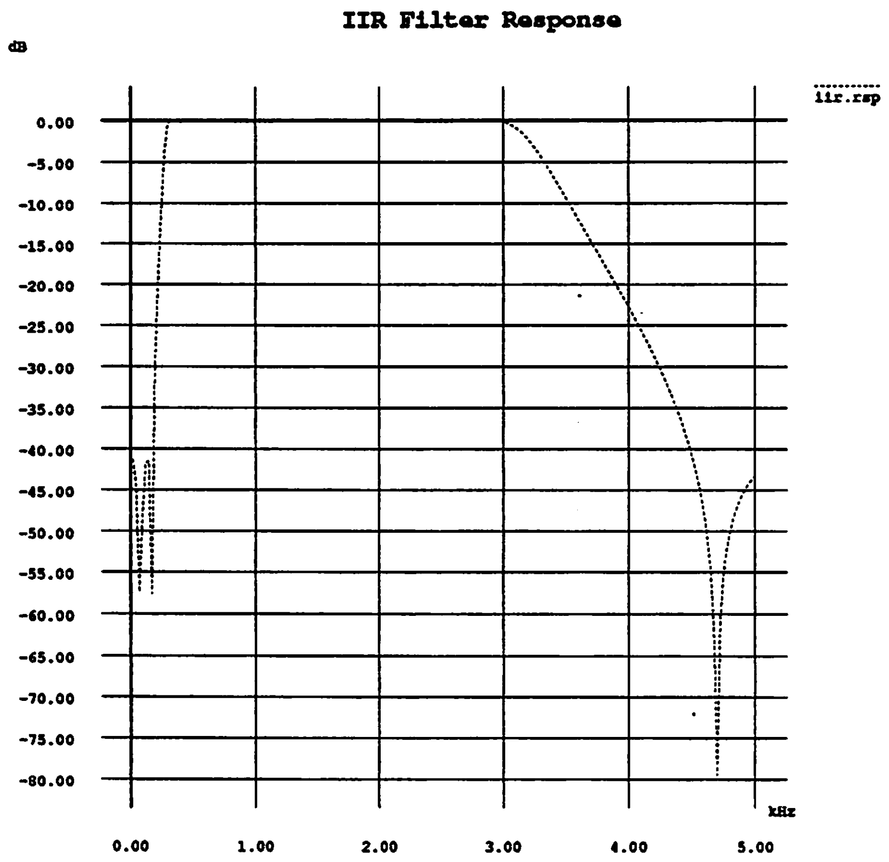


Figure 8.4: Frequency Response of 8-pole Bandpass IIR filter.

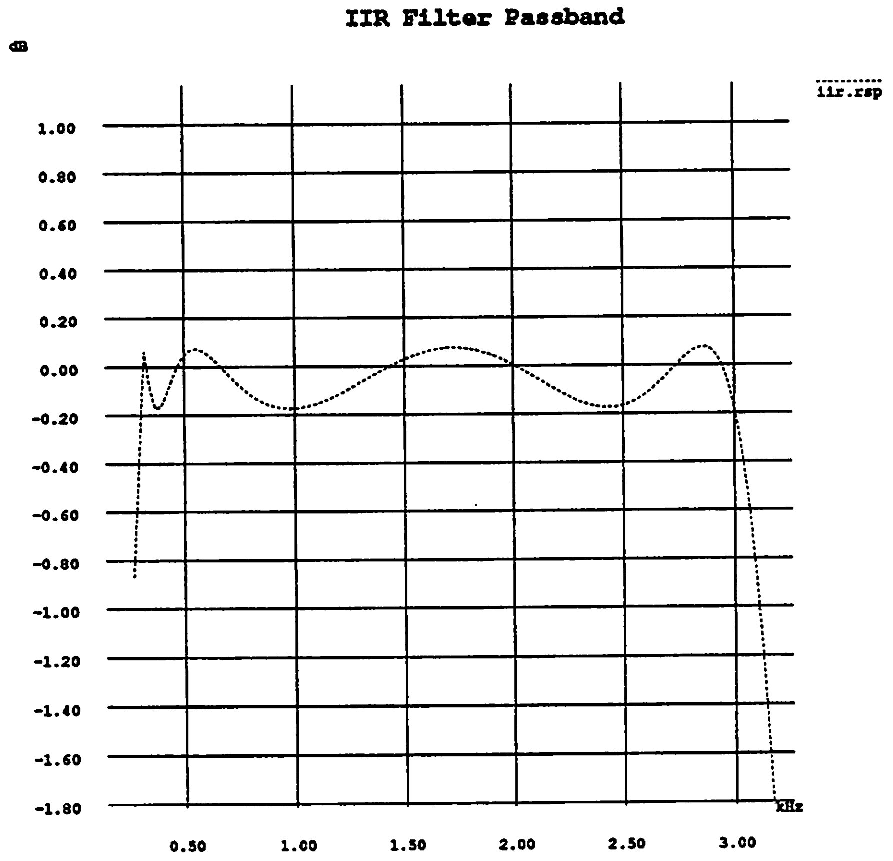


Figure 8.5: Passband of 8-pole Bandpass IIR filter.

of the stages was done to try to maximize the dynamic range of the filter. When scaling is done by a power of two, then simple shifting was used in the assembly code rather than multiplication. The program timing is interesting because, since the datapath operations switch back and forth between shifts and multiplies, the instruction cycle times switch between short and long periods since a multiplication requires more time.

1.3 CHIP3

The program in CHIP3 runs through a series of calculations that test the various functions available in the datapath and controller. Besides testing all of the mathematical functions, the program reads in several numbers from the input port and performs tests on them to determine when a branch condition is met. This simulates the control functionality of the DSP. The assembly code for the program is shown below. Numbers have been added to the instructions for easier referencing.

```
/* Exerciser program for datapath1 */

DECL
        /*  locations in ram */
        EQU     zero    0
        EQU     one     1
        EQU     mask    2
        EQU     ram1    3
        EQU     ram2    4
        EQU     ram3    5
        EQU     ram4    6
        EQU     ram5    7
        EQU     ram6    8
        EQU     ram7    9
        EQU     ram8    10
        EQU     onebar  16
        /*  locations in crom */
        EQU     crom0   20  /* 3/4 */
        EQU     crom1   21  /* -3/4 */
        EQU     crom2   22  /* 5/8 */
        EQU     crom3   23  /* -11/16 */
        /* condition codes */
        EQU     uncond  0
        EQU     BZ      1
        EQU     BZ_bar  2
        EQU     sign    3

START

 1    INIT:   ACC = 0;
            /* clear ram locations */
 2            (zero=acc) = 0;
 3            (ram1=acc) = 0;
 4            (ram2=acc) = 0;
 5            (ram3=acc) = 0;
 6            (ram4=acc) = 0;
 7            (ram5=acc) = 0;
 8            (ram6=acc) = 0;
 9            (ram7=acc) = 0;
10            (ram8=acc) = 0;
            /* load constants into ram from input port */
11     LOAD:   acc = acc,(one=in);
```

```
             acc = acc,(mask=in);
             acc = acc | ram1;
             (onebar=l_acc) = 0 | ~one;

    begin:   acc = 0;
         /* counting, adding */
             acc = acc + one;
             acc = acc + one,(ram2=in);
             acc = acc + one;
             (ram1=l_acc) = acc + one;

         /* masking and control */
             acc = 0 | ram2;
             acc = acc & one;
             acc = acc;
             if(BZ) goto print1;
             acc = 0 | (out=onebar);
             goto skip1;
    print1:  acc = 0 | (out=one);
    skip1:   acc = 0 | ram2;
             acc = acc & one<<1;
             acc = acc;
             acc = acc;
             if(BZ) goto print2;
             (out=l_acc) = 0 | onebar;
             goto skip2;
    print2:  (out=l_acc) = 0 | one;
    skip2:   acc = 0 | ram2;
             acc = acc & one<<2;
             acc = acc;
             acc = acc;
             if(BZ) goto print3;
             (out=l_acc) = 0 | onebar;
             goto skip3;
    print3:  (out=l_acc) = 0 | one;
    skip3:   acc = 0 | ram2;
             acc = acc & one<<3;
             acc = acc;
             acc = acc;
             if(BZ) goto print4;
             (out=l_acc) = 0 | onebar;
             goto skip4;
    print4:  (out=l_acc) = 0 | one;

         /* branch on zero */
    skip4:   acc = acc,(ram4=in);
             acc = acc;
             acc = acc;
             acc = 0 | ram4;
    again:   (out=l_acc) = acc - one;
             acc = acc;
             acc = acc;
             if(BZ) goto skip5;
             goto again;
    skip5:   (out=l_acc) = 0 | onebar;

         /* branch on sign */
             acc = acc,(ram4=in);
             acc = acc;
             acc = acc;
             acc = 0 | ram4<<15;
    again2:  (out=acc) = acc - one<<15;
             if(sign) goto skip6;
             goto again2;
    skip6:   acc = 0 | (out=onebar);

         /* logical functions */
             (out=l_acc) = 0 + one;
             (out=l_acc) = acc | ram1;
             (out=l_acc) = 0 + ~one;
```

```
72          (out=l_acc) = 0 | one;
73          (out=l_acc) = acc | onebar;
74          (out=l_acc) = acc ^ one;
75          (out=l_acc) = acc & one;

        /* multiply */
76          acc = 0;
77          acc = acc,(ram6=in);
78          acc = acc,(ram7=in);
79          acc = acc;
80          acc = 0 | (ram8=ram6)<<15;
81          (out=acc) = 0 + ram7 * T;
82          (out=acc) = 0 + ram7 * crom0;
83          (out=acc) = 0 + ram8<<15;
84          goto begin;
```

The functions that this test program performs will now be described in more detail. The first instructions following the initialization code (starting with Instruction 11) load in several constants to memory. The RAM location *one* receives the first constant which is supposed to be the value 0x0001 supplied by the user. The second value written to *mask* is not used in this version of the program so it is not relevant. After reading the constants, the 1's complement of the constant *one* is formed and stored in location *onebar*. The correct value is 0xfffe. The constant *one* is used for masking bits in the control section of the program. Both *one* and *onebar* are used as a crude signalling mechanism for when the program detects certain conditions. Thus, in the code that follows, often *one* is sent to the output port when a condition code is TRUE and *onebar* is sent to the output port when the same code is FALSE.

After the label "begin", some simple addition is performed. The accumulator is cleared and then *one* is added four times. The result is stored in RAM location *ram*4. During the counting, another input is read from the user into position *ram*2. This is meant to be an arbitrary number whose 4 LSB's are examined in the following test.

The masking and control section test uses the test value in *ram*2. The idea behind this section of code is the following: Examine the test number bit by bit and output *onebar* if the bit is a 1 and *one* if the bit is a 0. The constant *one* is used as a mask and the barrel shifter is employed to shift the single 1 in *one* to the desired bit location. For each bit test, the following instructions are executed:

1. Load Accumulator with test input.

2. AND Accumulator with mask. (*one* shifted left by n places)

3. Wait two instruction cycles for ACC to be valid.

4. if(ACC == 0) send out *one*, else send out *onebar*.

As an example, if the test input is 0x0004 (0100 in binary), then the output port values should consecutively be 0x0001, 0x0001, 0xfffe, 0x0001 from the four bit tests. The code actually contains a mistake however. Note that there is only a single NOP instruction in the section testing the first bit. Instruction 22 should be followed by another NOP before the Accumulator condition is checked. The two NOPs are necessary due to the pipeline delays in the datapath and the way the Data Stationary code is written. The effect of the missing NOP is that the masking is not performed. Instead, the test input value itself determines whether the BZ (Branch on Zero) condition is met. If the test input has *any* bit which is non-zero, then the condition is not met and the value 0xfffe is sent out.

The next section of code begins with Instruction 51. This section tests branching also. A test value is read into location *ram*4 and then it is decremented until the BZ (branch on zero) condition is met. Then the program branches out of the loop and sends *onebar* to the output port. After each decrement, the output port receives the value in the accumulator. Notice that for the test to be correct, two NOPs are inserted between the decrement instruction and the BZ test to compensate for the pipeline delays of the datapath. For an input number n read into *ram*4, the correct outputs will be $n-1, n-2, \cdots, 0, fffe$, where the last number is *onebar* which signals that the branch has occurred.

The next section of code begins with instruction 61. This tests the branch on sign condition. As with the last section, a test input number is read into *ram*4. The value is loaded into the accumulator (high order bits this time). The number is decremented until it becomes negative, satisfying the branch condition and causing *onebar* to be sent out. This section contains an intentional programming error and it is shown here again for convenience:

```
                /* branch on sign */
61                  acc = acc,(ram4=in);
62                  acc = acc;
63                  acc = acc;
64                  acc = 0 | ram4<<15;
65          again2: (out=acc) = acc - one<<15;
66                  if(sign) goto skip6;
67                  goto again2;
68          skip6:  acc = 0 | (out=onebar);
```

Notice that the test to branch on *sign* is performed in the instruction immediately following the decrement. This violates the rule that branch condition codes become valid two cycles following the instruction which reads data for the test. Enhancements to the assembler would include warnings of this sort of mistake. As it is, the code causes two extra decrements to occur before the fact that the Accumulator has become negative is realized. Therefore, the *correct* outputs for a test input number of n are $n-1, n-2, \cdots, 0, ffff, fffe, fffd, fffe$. The number decrements until it reaches the value $fffd$ where it is finally recognized by the branch on sign test and the number *onebar* is sent out as a signal that the condition has been met.

The section of code beginning with Instruction 69 tests the various logical functions available. First, the value *one* (0x0001) is added into the Accumulator. It is then OR 'd with the value in *ram*1 which is 0x0004 from the incrementing performed earlier in Instructions 16-19. The correct result is 0x0005. The next instruction is written to load the 1's complement of *one* into the Accumulator. However, rather than using the OR function to load the Accumulator, addition is used. This is generally not advisable because the adder portion of the ALU is slower than the logical section. Additionally, this instruction caught a small bug in the assembly language/ALU implementation. Whenever the A-input to the ALU is inverted, the C_{in} of the adder is set to a 1. The purpose is to correctly implement 2's complement subtraction. Since Instruction 71 uses the adder *and* the 1's complement function, the result is actually the same as subtraction. Therefore, the 2's complement of *one* or 0xffff is the result. The following instruction just loads the Accumulator again with *one*. The Accumulator is the OR 'd with *onebar* to give the result 0xffff. The Accumulator is next XOR 'd with *one* which has the effect of toggling the LSB to 0 resulting in 0xfffe. Finally, the Accumulator is AND 'd with *one* which has the result 0x0000.

The section of code beginning with Instruction 76 tests the multiplier and data move operations. First, two test inputs are read in from the input port. The are stored in *ram*6 and *ram*7 locations. Instruction 80, performs a data move between locations *ram*6 and *ram*8. The next instruction multiplies *ram*7 with the last location read from RAM, namely *ram*6. Thus, the output is the product of the two test inputs. The next output is the product of *ram*7 and a coefficient from the CROM. The last instruction loads *ram*8 to the Accumulator and sends it to the output port. If the data move instruction works properly, the output is identical to the first test input which was stored in *ram*6.

2. Laboratory Measurements

The three DSP chips were fabricated in MOSIS $2\mu m$ N-well CMOS technology. In the design of each of the handshake circuits, extra buffers were added on the handshake signals so that the signals could be brought out to pads without degrading the performance. Thus, all of the handshake signals between stages of the pipeline were available for observation. The names of the signals have a suffix "buf" added to them.

For testing the mathematical or logical operations of the chips, they were connected to a digital test system that is able to both supply arbitrary input vectors and acquire words from the output port. The acquisitions were triggered by the output port *Request*. The input port *Request* line was driven by an external clock that ran with a period greater than the time required for computation between samples. With this setup, the chips run at a slower *sample* rate than is possible with full handshaking at the I/O ports, but the internal computation rate is still full speed since it is completely self-timed. Between the time that a sample is computed and the next input request comes, the chip just waits. (See the section in the last chapter on the RAM handshaking for a more complete description.)

To observe timing signals and measure speed, an oscilloscope or digital signal analyzer was used.

2.1 CHIP1 Measurements

Logical

The logical outputs from CHIP1 are shown in Table VII for a full scale (0x7fff) input impulse. The ideal values are just the original impulse response of the FIR filter. The discrepancy between the ideal values and the measured outputs is predicted by the chip level simulations. In fact it is simply a consequence of the multiplier being single precision. Since it has only a 16-bit output (for a 16x16 input), there is a finite probability that the LSB will be in error. For the coefficients of this particular FIR filter, exactly 10 of the multiplications have the LSB error. Hence, each of the output values are low by an amount of 10 LSBs. The error causes a very minor shift in the frequency response. This is shown in Figure 8.6 where the ideal and measured frequency responses are plotted.

Table VII Impulse Response of CHIP1

Sample	Ideal	Measured
1	0xfec4	0xfeba
2	0x01e8	0x01de
3	0x0470	0x0466
4	0xfdb1	0xfda7
5	0xf559	0xf54f
6	0x032a	0x0320
7	0x2705	0x26fb
8	0x3b54	0x3b4a
9	0x3b54	0x3b4a
10	0x2705	0x26fb
11	0x032a	0x0320
12	0xf559	0xf54f
13	0xfdb1	0xfda7
14	0x0470	0x0466
15	0x01e8	0x01de
16	0xfec4	0xfeba
>16	0x0000	0xfff6

Timing

The timing signals of CHIP1 matched the simulations well. Since the program does a series of multiplications, the instruction rate follows the multiplication rate for all but the last NOP instruction. This is shown in the oscilloscope trace in Figure 8.7. The processor is shown waiting for the *Reqi_IN* signal to drop. The program itself is at the first instruction (which reads the input port for a new sample) during the wait. After *Reqi_IN* falls, the program begins and the various handshake signals shown pulse at the rate determined by the throughput of the iterative multiplier. Since the multiplier takes much longer than shift or add operation, it is will be referred to as a "slow" instruction.

At the end of the 16 multiplies that perform the FIR filtering, the last NOP instruction just sends the result to the output port. The NOP of course is a "fast" instruction and Figure 8.8 shows how the processor speeds up. As a comparison, the IRSIM chip simulation results in Figure 8.9 show the same signals that are shown in the scope traces of Figure 8.8.

The times for the fast (shift) and slow (multiplier) instructions were measured for the entire batch of chips and average times are shown in Table VIII. Times were measured

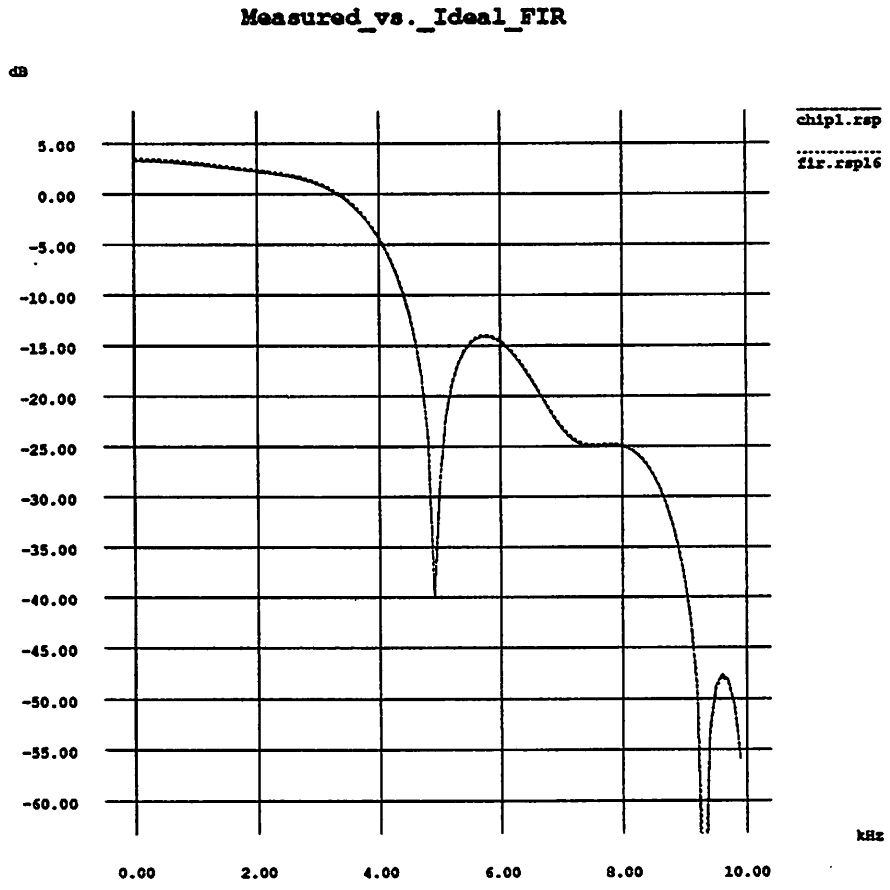


Figure 8.6: Passband of 8-pole Bandpass IIR filter.

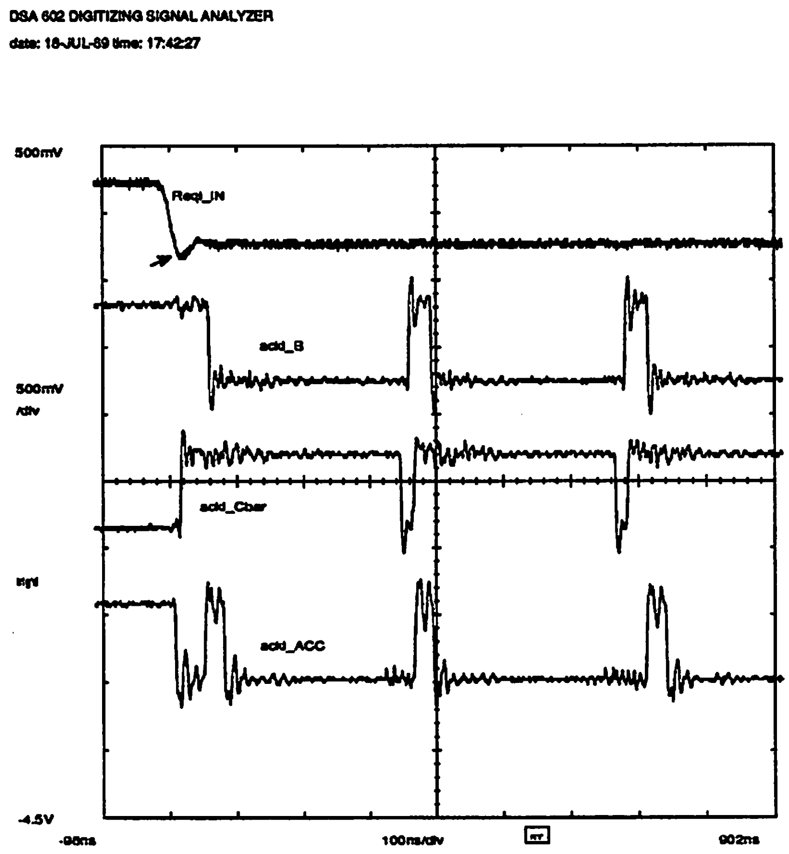


Figure 8.7: Oscilloscope trace showing handshake signals in CHIP1 just after a new sample arrives.

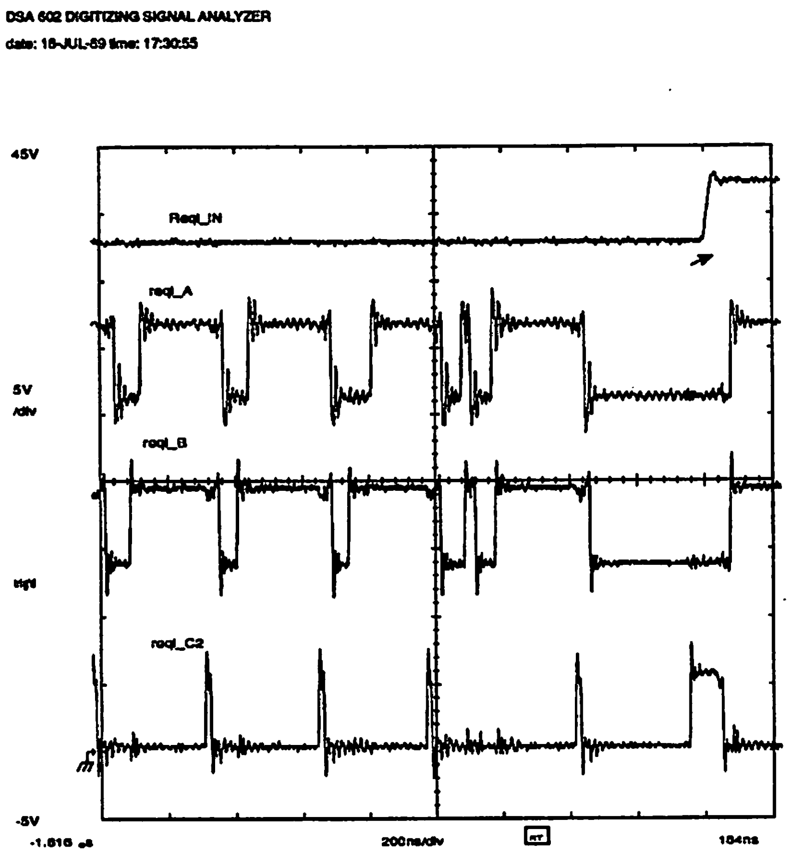


Figure 8.8: Oscilloscope trace showing handshake signals in CHIP1 at the end of the program loop. The fast instruction is a NOP.

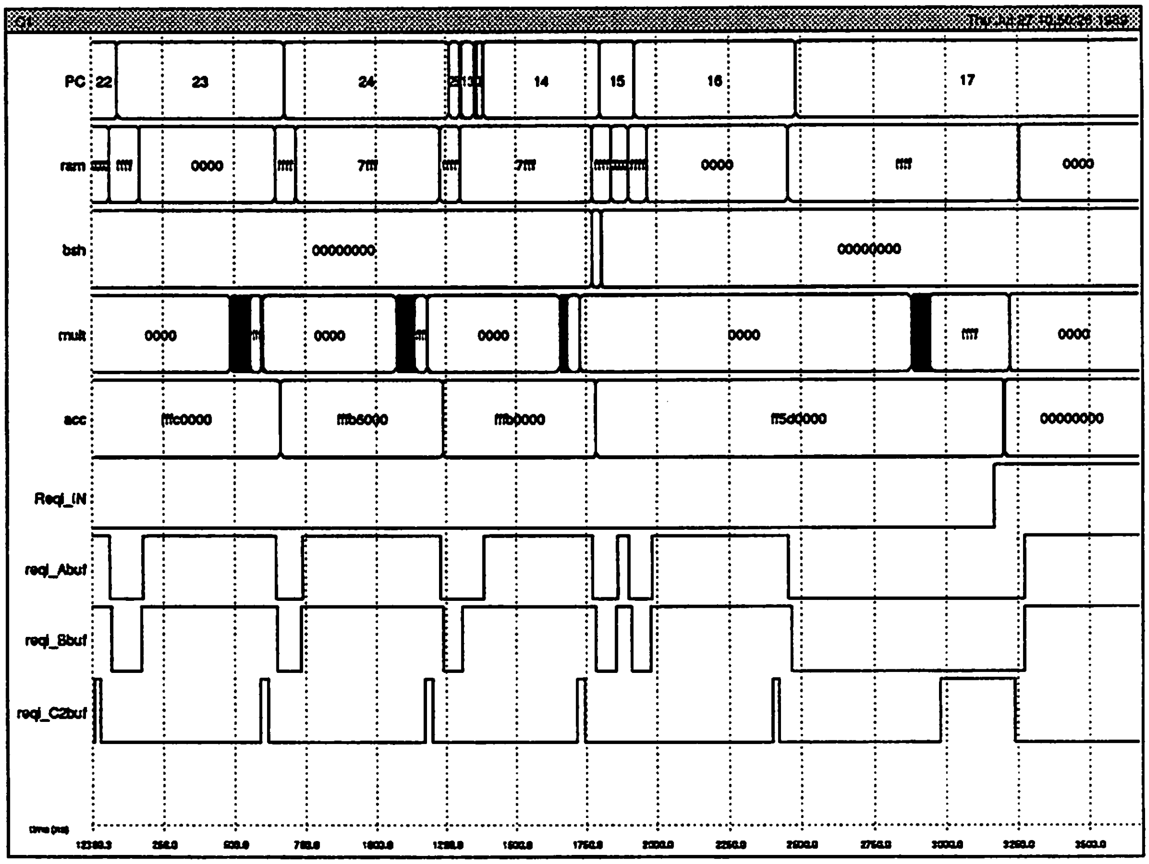


Figure 8.9: Simulation results showing the same handshake signals as measured in the previous figure.

at different supply voltages. Because of the compensating nature of self-timed circuitry, the processor continues to operate at different supply voltages, however the speed changes. The handshaking guarantees the correct sequence of events. Beyond that, the chip will run as fast as the circuits can compute given any set of data or supply voltage.

Table VIII Average Processor Instruction Periods
MOSIS $2\mu m$ N-well CMOS

V_{dd}	Shift	Multiply
3.6V	$105nsec$	$440nsec$
5.0V	$73nsec$	$337nsec$
7.0V	$55nsec$	$260nsec$

2.2 CHIP2 Measurements

Logical

As with CHIP1, the filter impulse response was measured for the IIR filter implemented on CHIP2. An input pulse of 0x4000 (decimal 0.5) was used. Due to the existence of feedback in an IIR filter, the impulse response length is infinite to within the precision of the processor. Table IX shows the measured results of the first sixteen samples. Again due to the multiplier LSB error, the measured values differ slightly from the ideal. Beyond the 7th sample, the measured response starts to deviate further from the ideal. The result was not predicted by simulation because of the limitations in the program for running such a long time simulation. It is suspected that a form of limit-cycle behavior occurs for the latter samples causing them to deviate from the ideal response. This is supported by data measured further out in time from the impulse. This behavior is likely due to the finite precision errors introduced by the multiplier.

Table IX Impulse Response of CHIP2

Sample	Ideal	Simulated	Measured
1	0x0800	0x07fd	0x07fd
2	0x1782	0x1778	0x1778
3	0x29f4	0x29dd	0x29dd
4	0x2ef7	0x2e52	0x2e52
5	0x1d23	0x1cd8	0x1cd8
6	0xf7e2		0xf76e
7	0xd238		0xd190
8	0xc25a		0xc171
9	0xceb9		0xcd7d
10	0xe9e6		0xe848
11	0xff1d		0xfd0a
12	0x02c6		0x002c
13	0xf924		0xf5e9
14	0xef72		0xeb7c
15	0xef99		0xeac9
16	0xf91b		0xf353

Timing

Figure 8.10 shows timing signals measured from CHIP2 just after a new input sample. The timing of CHIP2 is much more interesting than for the FIR filter because the program switches back and forth between using the multiplier and shifter. The beginning of the IIR filter program on CHIP2 is repeated below with program counter addresses for convenience. The scope trace shows the span between instructions 0x0d and 0x1f. Note that all input/output instructions are executed after the *last* pipeline stage of the datapath. The delay is necessary to output the correct Accumulator values when desired. For consistency, the input instructions are also performed after the last pipeline stage. Therefore, in the program below, while the assembly code shows an input instruction occurring first, when the PC = 0x0b, the actual *IN* control signal (see the RAM handshaking circuit) emerges from the last stage of the instruction pipeline causing the input to be read two instructions later. The PC value increments to 0x0d before the processor enters a wait state for the *Req_IN* signal. So, the first multiplication instruction is completed (data in ALU) and the second shift instruction is in progress (valid data sitting at shifter output). When the *Reqi_IN* signal arrives, the second instruction data enters the ALU

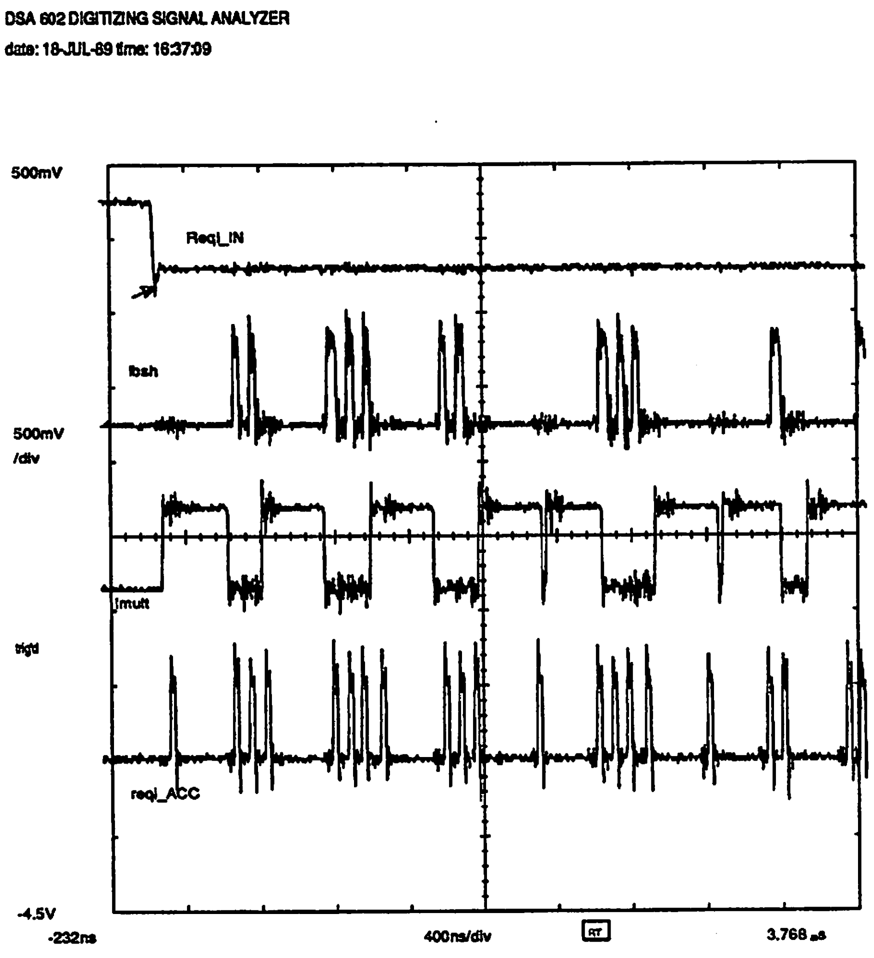


Figure 8.10: Oscilloscope trace showing the shifter, multiplier compute signals and *reqi_ACC* in CHIP2 just after a new sample arrives.

and the third multiply instruction will begin at the falling edge of *Reqi_IN*. The scope trace picks up the action at this point. The signal **Ibsh** or **Imult** is asserted when the instruction uses the shifter or multiplier respectively. The *reqi_ACC* of course cycles once per instruction.

```
/****** beginning of IIR filter program on CHIP2 ******/
PC   START

00   INIT:   acc = 0;

             initialization code...

0b   sample: acc = 0 | del2 * a2,(input=in);
0c           acc = acc + (tmp=del1)<<15;           coeff a1 > 1
0d           acc = acc + del1 * a1;  fractional part of a1
0e           (del1=acc) = acc + input<<15;
0f           acc = acc + del2<<15;
10           acc = acc + (del2=tmp) * b1;
11           (tmp=acc) = acc;
12           acc = 0 | del4<<15;
13           acc = acc - del3<<15;
14           acc = acc + (del4=del3) * b1b;
15           acc = acc + (del3=tmp)<<15;
16           (tmp2=acc) = acc;
17           acc = 0 | del6 * a2b;
18           acc = acc + (tmp=del5) * a1b;
19           (del5=acc) = acc + tmp2<<14;
1a           acc = acc + del6<<15;
1b           acc = acc - (del6=tmp)<<15;
1c           acc = acc + tmp * b1c;
1d           acc = acc + del8 * a2c;
1e           acc = acc + (tmp=del7)<<15;
1f           (del7=acc) = acc + del7 * a1c;
             .
             .
             .
```

Figure 8.11 shows the IRSIM output from the same time span as in Figure 8.10. Note that the program counter value leads the current instruction values of **Ibsh** and **Imult** by two. This is because 1) in the controller, the PC value is incremented when the current instruction ROM code becomes valid and 2) there is one pipeline delay between the controller and shifter/multiplier. Some of the other pipeline handshaking signals are shown in the oscilloscope trace of Figure 8.12 for the same section of code at the beginning of a new sample period.

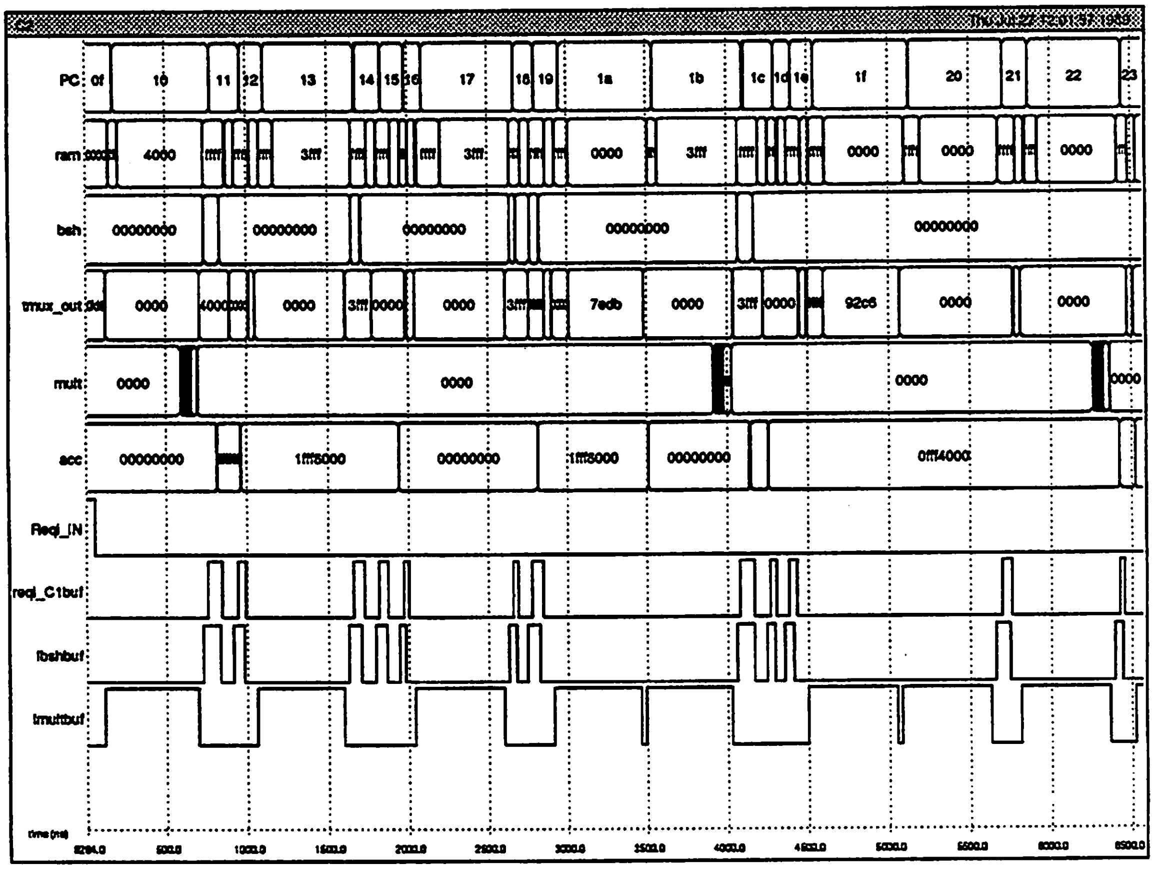


Figure 8.11: Simulation results showing the same handshake signals as measured in the previous figure.

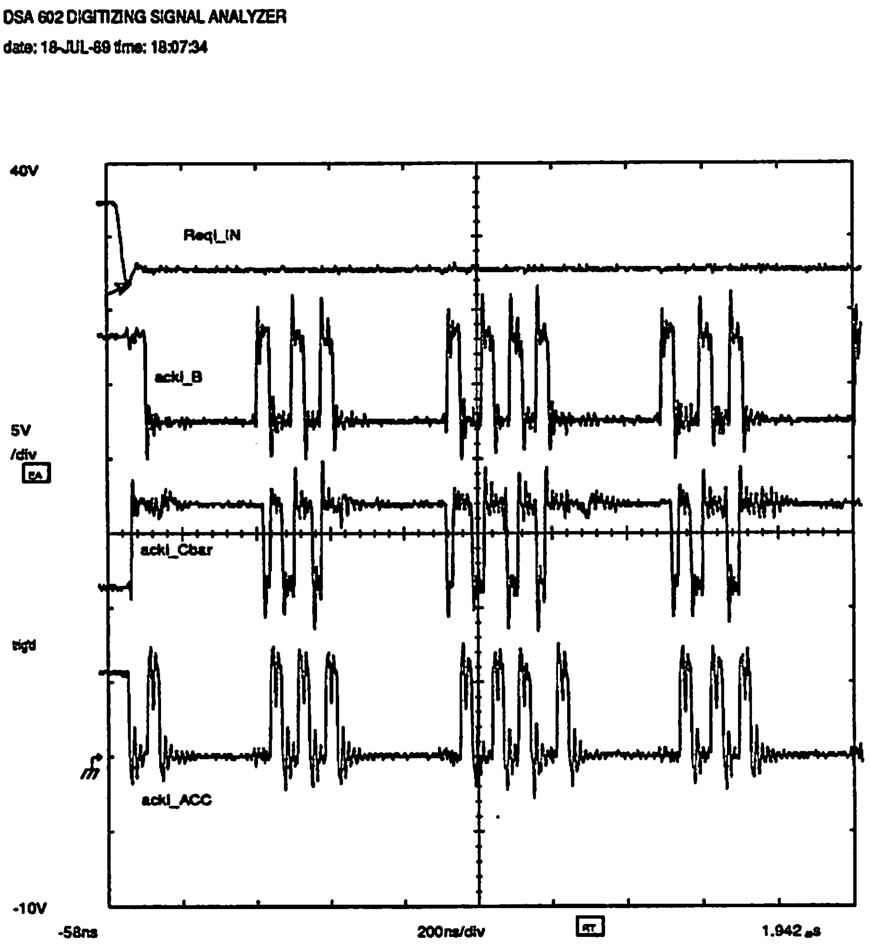


Figure 8.12: Oscilloscope trace showing other handshake signals in CHIP2 just after a new sample arrives.

2.3 CHIP3 Measurements

Logical

The digital test system was programmed to supply the required test input numbers for the program in CHIP3 and the outputs were acquired. In the test setup, the output port *Acknowledge* signal is tied to the output *Request*. The output data is strobed into the tester on the rising edge of the two signals. The input data was sent in at a constant rate determined by a setting on the tester. The time between input samples was just chosen to exceed the longest time required for the processor to do its computation between input instructions.

All of the predicted results of the tests were observed at the output port. The mistakes in the assembly code mentioned above caused branch conditions to occur late due to pipelining effects as predicted (This showed up in the chip simulations also). The multiplication tests showed a single LSB error as predicted by the simulation also.

3. Interface to the Outside World

Clearly, interfacing the DSP input or output port to another self-timed device is very simple. The handshake signals are just connected. For external clocked devices however, a different strategy is needed. Figure 8.13 shows one way of making the connections. At the input side, for a signal processing application, there is typically samples that arrive synchronized to some sample clock. The clock is fixed for real time processors. Connecting this clock to the *Reqi_IN* signal causes the transfers to occur in sync with the sample clock, and there is no checking done to ensure that the processor is ready since the *Acki_IN* signal is ignored. The user must ensure that the program execution time between input instructions is shorter than the period of the sample clock. The output port could be hooked to a host processor which accepts interrupt signals. The *Reqo_OUT* signal is tied to the interrupt pin and the acknowledge signal from the host clocks in data and is connected to *Acko_OUT*. Alternately, the same sample clock could be used to clock the register in the external device because having the FIFOs present on the DSP chip makes the exact time of the transfer of data after an I/O instruction unimportant as long as it occurs before the end of a sample period. As the figure shows, there is no timing clock required for the DSP.

The question often arises about how to evaluate the execution time of a DSP program when the exact instruction cycle times are not known in advance. This can be problem in *both* clocked and self-timed systems when the program is data-dependent. For example, in a signal processing program, the convergence of an adaptive filter might trigger some other event. Since the convergence time is not known *a priori*, the execution time is also unknown. For a self-timed processor, there is added uncertainty due to the different times of instructions and their data dependency. The data dependency helps in achieving an average execution time but it is by no means a requirement of the design. A self-timed adder cell can easily be made to generate a data valid signal always after the worst-case carry propagation time (the time required for using the adder in a clocked situation). As far as the different instruction times, the user can always set an upper bound by using worst case times for the various instructions. So, the conservative estimate for a self-timed DSP execution time essentially matches that of a clocked processor. Beyond that, the user may take advantage of the average times in a manner suitable to the application.

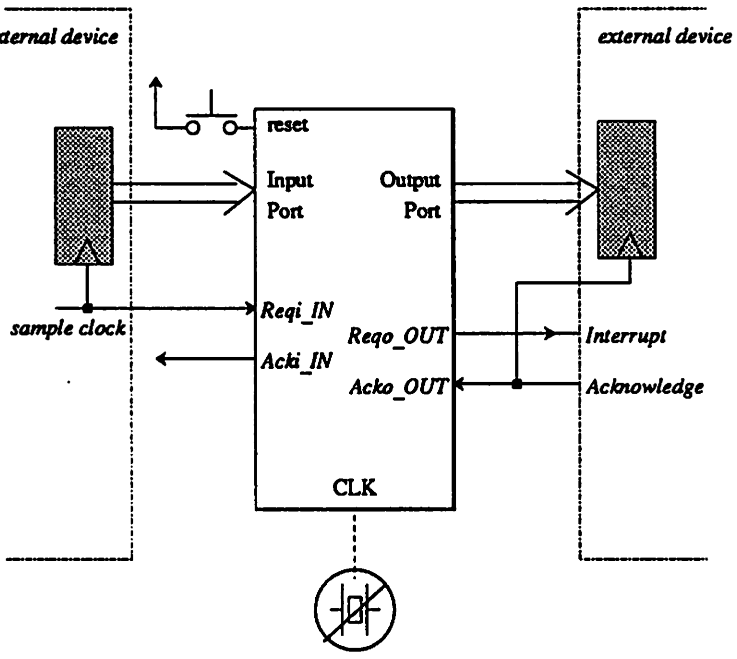


Figure 8.13: Interfacing the self-timed DSP to external devices.

4. Summary

Three programs were written to test the operation of the self-timed programmable signal processor chip. The first was a simple FIR filter which makes extensive use of the multiplier. The second program implements an IIR filter and it combines shift and multiply operations. The third program tests various other functions of the DSP including branch operations. The single precision output of the multiplier causes minor errors in the response of the filter chips, however these were predicted by simulations. The IIR filter exhibits some limit-cycle behavior which was not originally known due to limitations of running the chip level simulator for a large number of cycles. The third chip functions agreed with the simulations exactly.

The timing signal outputs of the self-timed DSP illustrate the differences between this type of circuit and normal clocked systems. Depending on whether a shift or multiply operation is performed, the cycle time adjusts to the speed of the hardware being utilized. The DSP also functions properly over a wide range of power supply voltages, the instruction speed varying with the supply, revealing the self-compensating nature of the circuit operation.

Chapter 9

Conclusion

1. DSP Design

This work has demonstrated the design of a general purpose programmable Digital Signal Processing integrated circuit that has asynchronous operation via the use of self-timed circuitry. After examining the motivation for pursuing a design of this type, the DCVSL logic family was presented as a way of generating completion information in each circuit. Using Signal Transition Graphs to describe the timing of each stage, handshaking circuitry can be synthesized to ensure that timing sequence. DCVSL gates were assembled to make standard datapath macrocells while handshake circuits were synthesized to interconnect the cells for a particular DSP architecture. Finally, three versions of the programmable DSP were fabricated and measured in the laboratory.

When this project was begun, there was virtually no previous work on self-timed programmable circuits. While several self-timed datapath cells such as a multiplier or divider[83] were presented, there was no circuitry that contained feedback. Additionally, many of the procedures for synthesizing handshake circuitry that were published contained errors or incorrect assumptions. Therefore, a great effort went into just understanding the problems of designing a self-timed DSP. The choice for partitioning the circuitry at the macrocell level was made on the basis that the board-level timing problems of today will be the chip-level timing problems of tomorrow. While the DSP chip discussed eventually worked correctly in the laboratory, it is instructive at this point to look back and examine the parts of the design that presented the greatest challenge and perhaps make some decisions about certain aspects of the design itself.

1.1 Handshaking Logic Design

In the early versions of the DSP, extremely unreliable operation was observed. Certain parts would function correctly for several seconds and then just stop, being locked in a certain state. Any fluctuation in the power supply voltage also caused the lock-up to occur. This was disappointing behavior for a circuit that is supposed to be *more* tolerable of processing and supply variations. By analyzing the handshaking signals during lock-up, the state was defined and simulations were performed to try and re-create the events leading to that state. The lock-up did not appear in any of the system simulations prior to fabrication.

The source of the lock-up condition was isolated to a single latch design in the controller handshaking circuit (ROMHS). It is part of the sequential handshake circuit shown in Figure 7.10. The latch schematic is given in Appendix C in Figure C.87. The pseudo-NMOS style SR-latch (with built in AND gate on the S input, S = A.B) has the characteristic that if A, B and R inputs are all HIGH , then *both* Q and $\bar{Q}$ will go LOW . Normally, the situation where all inputs are HIGH should not occur. However, on the actual chip, the situation did occur due to some transient and caused the problem. In later versions, only the $\bar{Q}$ output of the latch was used and a separate inverter was inserted to derive Q from $\bar{Q}$. The lock-up state disappeared and the operation of the DSP became extremely reliable. In fact, the supply voltage could be continuously varied between 3.5V and 7V without any interruption in the operation of the DSP.

The problem described above makes an important point about the design of asynchronous circuits. The logical equations that are obtained from the handshake synthesis must be *exactly* implemented for all conditions of the inputs of the circuit. The complementary outputs of the SR-latch in the controller handshaking became non-complementary under a single condition which violated their desired relationship ($\bar{Q}$ = NOT Q). However unlikely a certain input state may seem, the logic must be designed to function properly for that state. The designer may get caught is a trap trying to meet several goals. The desirability of having efficient, high-speed handshake circuits makes it tempting to use shortcuts or circuits which are less rigorous in following the logic equations under all conditions. However, since the handshake circuit elements are used in many places of an asynchronous system and their operation is critical to the correct functioning of the system, they should be designed to be "bullet-proof".

1.2 Delay Matching

The use of circuit delay matching in place of true self-timed operation had mixed results and several conclusions can be drawn:

Macrocell Design

As explained in Chapter 6, there were places in the macrocell design where the matching of circuit delays on an IC was exploited to reduce the circuitry required for generating completion signals. This approach was observed to be safe since the delay matching is done on a local level and the loading on the circuits being matched is well defined within the macrocell. No problems were encountered in macrocells that used the approach such as the barrel shifter.

Between Macrocells

At locations outside of a macrocell, some delay matching was done to avoid the use of handshaking between the instruction pipeline and the datapath handshake circuits as explained in Chapter 7. The use of delay matching between macrocells is seen as a risky alternative to implementing correct handshaking. While it can be made to work, the required delay time depends on the loading seen by the signals, which in turn depends on the global routing of the chip. One of the assumptions made in the model of a self-timed stage was that the registers which store information (data or control) be in proximity to the computational block so that their outputs are stable before the computation is initiated. The registers of the datapath macrocells were designed directly inside the cell boundary resulting in a well defined loading and delay. The instruction pipeline registers however were kept separate from the datapath. In an early layout floorplan for the DSP, all of the instruction pipeline registers were placed in one portion of the chip layout. Wires from the pipeline to the different datapath stages varied greatly in length. This caused the DSP to operate incorrectly because of the long delays between the instruction pipeline and the datapath cells using the control signals (The delays were gross and the error was caught in chip simulations). While the layout was modified to match the original assumption by placing each instruction pipeline register near the cells which it feeds, the loading is still not easily known in advance. Therefore, delay matching techniques that compensate for these wiring delays have to be quite conservative to work. A long chain of inverters might be necessary to ensure that the delay is long enough, and of course the efficiency of operation is reduced by the conservative approach.

The conclusion for avoiding delay matching between cells that are connected by the global routing is related to the next section. In fact, by making more rigorous use of handshaking, the problems mentioned above can be completely avoided. A register giving completion information may be required in some cases.

1.3 Register Completion

In the initial phase of the DSP design, the lack of a self-timed circuit for a register was not seen as a handicap. A plan was adopted where the handshake circuits for the datapath would also control the instruction pipeline. Handshaking between the registers in the instruction pipeline and datapath was not used. This in fact did cause several problems to appear in the early versions of the chip. While the problems could be overcome by adding some delay circuits, the solution is not as reliable and it is more sensitive to the global routing as explained above.

Instruction Pipeline

The instruction pipeline registers are clocked by the *Acki* signals from each stage in the datapath pipeline. In the model followed by the chip design, a register clocked by *Acki* is assumed to have valid outputs before the next operation (usually raising the I signal to initiate computation in the logic block) begins. In the case of a control signal affecting the handshake circuit configuration itself, a potential race condition exists as was shown in Figure 7.12. The solution used on the DSP design was shown in Figure 7.13, where a delay was added between the *Acki* signal and the part of the handshaking circuit that is affected by the control signal emerging from the instruction pipeline register clocked by *Acki*. This was made to work, but the risk in such a design procedure is greater due to the uncertainty in the loading on the control signal from routing wires. The design of the *Reg Delay* block in the figure can include the clock buffer for the register being matched along with an actual register cell. However, in the actual instruction register being matched, there are a number of register cells placing a load on the clock lines along with the load from the routing wires between the outputs and their destinations. The routing wires are created at the end of the design process. Thus, the risk involves correctly predicting the loading and delay seen by the control signals.

The prediction of loading and delay from routing wires is precisely the kind of activity that self-timing is supposed to eliminate. Therefore, the more rigorous way to approach the problem with the control signals is simply to use handshaking between the datapath and the control pipeline. A slight increase in the amount of circuitry required for handshaking is necessary to handle a completion signal from the instruction pipeline register. This would most likely be smaller than the *Reg delay* cell. The method however demands a completion signal from the instruction register. A completion signal can be generated at the register itself by using a "dummy" register cell, but of course it still suffers from the unknown delay between the register and the rest of the circuitry. Delay matching of this sort is a safer approach if the completion signal generated by the dummy cell is routed to the same location as the control signal, equalizing the wiring delays for the pair.

Another way of generating a completion signal for a register is to compare the input and output of the register. When they are the same while the clock is high, then the output is assumed to be valid[66]. However, the comparison might be necessary for each bit of the register, depending on the application. For a single control signal affecting the state of a MUX such as in the DSP datapath, the method is appropriate. There is currently research being done to examine the effects of combining the handshaking circuitry and the register into a single circuit, thus eliminating the need for a register with completion[69].

I/O FIFO problem

In both the Input Port FIFO (Figure 7.15) and Output Port FIFO (Figure 7.16), two registers are cascaded, the timing of which are controlled by a simple HS4 circuit. The registers form a self-timed pipeline. In the normal connection for a self-timed stage, the *Reqo* signal is fed to the DCVSL **I** input and the **DV** is then used as the new *Reqo* signal. For a FIFO, there is no computation circuitry, just the registers themselves. For this connection, once again a form of delay matching was employed. The first FIFO register outputs must be settled before the second FIFO register is clocked (plus the set-up time of the second register). On some versions of the DSP, errors in the input port values were observed because some the bits in the FIFO register were not settling in time. The use of fast c-elements, which normally are chosen to increase the efficiency of the handshaking circuitry, actually caused a problem in the FIFOs because they were so much faster than the registers themselves.

To alleviate the problem, dummy register delays were placed in series with the *Reqo* signal from the HS4 circuit to match the "computation" delay of the FIFO registers as shown in Figure 7.17. The loading between the two registers in the FIFO is small since they are in a common layout cell. The loading on the output of the second register however, is subject to global routing constraints which makes it more uncertain.

1.4 Simulation Environment

The design of an asynchronous circuit typically places more demands on the simulation environment than a synchronous circuit. The circuit operation is made to follow a desired *sequence* but the actual speed of that sequence is not controlled. Unlike a synchronous circuit where data are transferred only at clock edges, the operation of an asynchronous circuit more like the carry chain of an adder. Within the constraints of the handshaking protocol, the data ripples through the circuit at full speed. This gives the operation more of an analog flavor. An absolute requirement for system simulation is an event-driven simulator. The existence of feedback in the handshaking circuits causes infinite loops to occur in a simulator where the node voltages are calculated until no further changes are observed. The IRSIM simulator was useful because it combined event-driven behavior with a pseudo-analog simulation of the actual devices and wire capacitances. An RC model for each device is constructed from its sizing information and the layout extracted capacitances. While this allowed system level simulations, several shortcomings of the program were still apparent (when non-working chips arrived!). First, it did not handle set-up time violations well on registers. Second, the accuracy of the timing itself is not high due to the simple model used for each device.

Using a simulator such as SPICE for an entire chip design is not possible currently due to the computation requirements. However, since it is really only the handshaking circuits, whose operations are critical to the chip performance, a good simulation environment for asynchronous chip design would have a "mixed-mode" capability. The large macrocells in the datapath of the DSP only introduce a delay in the handshaking signals. It would be useful to be able to describe them behaviorally and do a detailed simulation of only the handshaking circuitry.

2. Future Work

This work has shown promising results from the application of self-timed circuits in digital integrated circuits. Much future work is necessary however for asynchronous chips to take hold in the design community. Given the resources available, the fabrication of the DSP presented here was done on a fairly standard $2\mu m$ process. Since the problems associated with global synchronization are really expected to occur at feature sizes below $1\mu m$, more data is necessary from asynchronous designs fabricated in the very latest technologies. A good sign was that the speed of the processor built in $2\mu m$ CMOS was comparable to clocked designs in the same process. So, while there may be no great benefit for using self-timed circuits with current chip technology, there was no apparent disadvantage in terms of processing power. This helps support the theory that when the clock skews begin to dominate a clocked design, a superior asynchronous version can be made which takes full advantage of the raw device speed.

In terms of the design methodology, a great deal of progress has been made in defining a reliable path from system specifications to asynchronous circuits. More work is needed however in the means for describing a desired series of operations for that appropriate handshake circuit synthesis. It can still be a confusing process to describe what one wants the circuit to do in terms of either a signal transition graph or guarded command. Further, taking the generated logic and actually connecting in the rest of the circuit with appropriate buffers, etc. can be error prone depending on the assumptions that are made beforehand. A more streamlined method for this part of the design methodology would be welcomed.

As with any new discipline, it takes time to become used to the idea of doing things in the new way and experience to make the tasks easier. It is felt that, with the great deal of research activity in the area of asynchronous circuit design currently taking place, this design style will become very natural to future chip designers.

3. Speed Comparisons

A natural question that arises in the study of asynchronous circuits is, "What is the speed advantage?" By avoiding the need for distributing a global clock, the design methodology is simplified at the chip level and the time delays incurred by a global clock can be used for computation. Additionally, an average versus worst-case instruction cycle is achievable. Obviously, quantifying the actual improvement is very much circuit design and application dependent. As an exercise though, some assumptions will be made in this section in order to try to get an idea of the improvement when using self-timed circuits.

Our hypothetical processor is designed in a 3μ process and it is roughly $1cm$ on a side. The instruction cycle rate is nominally $100MHz$, so the instruction period is $10nsec$. For a self-timed macrocell connected to a handshaking circuit as explained in Chapter 5, the overhead time for the each evaluation cycle is equal to 4 c-element delays and 2 buffer delays (Equation 5.8), where the buffers are used to increase the drive between the handshake circuit output and the DCVSL I signal inputs. A $1pF$ load is assumed for the buffer which means that $\tau_{1pF} = 0.5nsec$ for a buffer that uses a cascade of gates, each sized e larger than the previous (optimum). Using data from Table II in Chapter 2, one gate delay is $85psec$. Assuming a c-element can switch in a single gate delay, the total overhead time for handshaking is $0.5nsec$ for the buffer time plus 4 gate delays of $85psec$, or $0.84nsec$. The efficiency of the system is 91.6% when the overhead is subtracted from the clock period.

For a synchronous system, the clock non-overlap time is what subtracts from the time to do computation. From Table II, it is estimated that $1.4nsec$ will be required for non-overlap time for each clock phase. The efficiency ranges from 44% to 72% depending on whether a two or four phase clock is used.

Therefore, an initial estimate based on comparing these efficiencies is that the asynchronous system will be 30 to 100% faster. Taking into account the ability to exploit data dependencies on circuit delay times in self-timed circuits implies that for some applications the advantage over synchronous circuits will be even greater. For real comparisons, time will tell as designs of both styles emerge from the manufacturing community.

Bibliography

[1] D.A. Huffman, "The Synthesis of Sequential Switching Circuits", *J.Franklin Institutes*, March and April 1954, pp.161-190,275-303.

[2] D.B. Armstrong, A.D. Friedman, and P.R. Menon, "Design of Asynchronous Circuits Assuming Unbounded Gate Delays", *IEEE Trans. on Computers*, vol. C-18, no. 12, December 1969.

[3] A.S. Wojcik and K.-Y. Fang, "On the Design of Three-Valued Asynchronous Modules", *IEEE Trans. on Computers*, vol. C-29, no. 10, October 1980.

[4] R.E. Miller, *Switching Theory*, John Wiley and Sons, Inc., New York (1965).

[5] Stephen H. Unger, *Asynchronous Sequential Switching Circuits*, John Wiley and Sons, New York (1969).

[6] Gyula Mago, "Realization Methods for Asynchronous Sequential Circuits", *IEEE Trans. on Computers*, vol. C-20, no. 12, March 1971.

[7] Jack B. Dennis, "Data Flow Supercomputers", *Computer*, November 1980, pp. 48-56.

[8] J.B. Dennis, G.A. Boughton and C.K.C. Leung, "Building Blocks for Data Flow Prototypes", *Proc. 7th Annual Symposium on Computer Architecture*, May 1980, pp. 1-8.

[9] Robert G. Babb II, "Parallel Processing with Large-Grain Data Flow Techniques", *Computer*, July 1984, pp. 55-61.

[10] I. Hartimo, K. Kronlof, O. Simula, and J. Skytta, "DFSP: A Data Flow Signal Processor", *IEEE Trans. on Computers*, Vol. C-35, No. 1, January 1986, pp. 23-32.

[11] Edward A. Lee, "A Coupled Hardware and Software Architecture for Programmable Digital Signal Processors", *Ph.D. Thesis, University of California, Berkeley*, UCB-ERL Memorandum No. M86/54, June 18, 1986.

[12] Daniel M. Chapiro, "Globally-Asynchronous Locally-Synchronous Systems", *Ph.D. Thesis*, Stanford University, October 1984.

[13] S.Y. Kung, "On Supercomputing with Systolic/Wavefront Array Processors", *Proceedings of the IEEE*, Vol. 72, No. 7, July 1984, pp. 867-884.

[14] Kostas N. Oikonomou, "Ideal Arbiters: Analysis and Design", *AT&T Technical Journal*, Vol. 66, Issue 2, March/April 1987, pp. 78-96.

[15] J. Hohl, W. Larsen, and L. Schooley, "Prediction of Error Porobability for Integrated Digital Synchronizers", *IEEE J. of Solid State Circuits*, Vol. SC-19, no. 2, February 1982.

TECHNOLOGY SCALING:

[16] A. Reisman, "Device, Circuit, and Technology Scaling to Micron and Submicron Dimensions", *Proceedings of the IEEE*, Vol. 71, No. 5, May 1983.

[17] C.K. Wang, "Switched Capacitor Signal Processing Circuits in Scaled Technologies", *ERL Memorandum*, University of California, Berkeley, No. UCB/ERL M86/84, November 3, 1986.

[18] Ping Ko, *Private Communication*, University of California, Berkeley, September 1987.

SYSTEM TIMING

[19] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley Publishers (1980), Chapter 7.

[20] *Motorola 68000 series Data books*, Motorola, Inc. 1986.

[21] Peter Ruetz, *Private Communication*, November 1988.

INTERCONNECT AND TECHNOLOGY LIMITATIONS:

[22] J.D. Meindl and H.B. Bakoglu, "Optimal Interconnect Circuits for VLSI", *Proceedings of IEEE Intl. Solid State Circuits Conference*, San Francisco, CA, February 1984, p. 164.

[23] W.S. Song and L.A. Glasser, "Power Distribution Techniques for VLSI Circuits", as submitted to *IEEE J. of Solid State Circuits*, May 1985.

[24] W.R. Heller, W.F. Mikhail, and W.E. Donath, "Prediction of Wiring Space Requirements for LSI", *Journal of Design Automation & Fault Tolerant Computing*, Vol. 2, No. 2, May 1978, pp. 117-143.

[25] W. E. Donath, "Placement and Average Interconnection Lengths of Computer Logic", *IEEE Transactions on Circuits and Systems*, Vol. CAS-26, No. 4, April 1979, pp. 272-277.

[26] Y. Pauleau, "Interconnect Materials for VLSI Circuits - Part I", *Solid State Technology*, February 1987, pp. 61-67.

[27] Y. Pauleau, "Interconnect Materials for VLSI Circuits - Part II", *Solid State Technology*, April 1987, pp. 155-162.

[28] Y. Pauleau, "Interconnect Materials for VLSI Circuits - Part III", *Solid State Technology*, June 1987, pp. 101-105.

[29] H.B. Bakoglu, "Circuit and System Performance Limits on ULSI: Interconnections and Packaging", Ph.D. Thesis, Technical Report No. G541-4, Stanford Electronics Laboratories, Stanford University, October 1986.

[30] K.C. Saraswat and F. Mohammadi, "Effect of Scaling of Interconnections on the Time Delay of VLSI Circuits", *IEEE J. Solid State Circuits*, Vol. SC-17, No. 2, April 1982, pp. 275-280.

[31] Takayasu Sakurai, *Approximation of Wiring Delay in MOSFET LSI*, *IEEE J. Solid State Circuits*, Vol. SC-18, No. 4, August 1983, pp. 418-426.

[32] P.W. Cook, S.E. Schuster, J.T. Parrish, V. DiLonardo, and D.R. Freedman, "1-um MOSFET VLSI Technology: Part III - Logic Circuit Design Methodology and Applications", *IEEE J. Solid State Circuits*, Vol. SC-14, No. 2, April 1979, pp. 255-268.

[33] B.L. Crowder and S. Zirinsky, "1-um MOSFET VLSI Technology: Part IV - Metal Silicide Interconnection Technology - A Future Perspective", *IEEE J. Solid State Circuits*, Vol. SC-14, No. 2, April 1979, pp. 291-293.

DCVSL REFERENCES:

[34] W.R. Griffin, L.G. Heller, and J.A. Hiltebeitel, "True/Complement NMOS-Rich CMOS Logic Circuit", *IBM Technical Disclosure Bulletin*, Vol. 25, No. 11B, April 1983, pp. 6060.

[35] L.G. Heller, W.R. Griffin, "Cascode Voltage Switch Logic: A Differential CMOS Logic Family", *ISSCC Digest of Technical Papers*, International Solid State Circuits Conference, New York, February 1984.

[36] W.J. Craig, W.R. Griffin, and L.G. Heller, "CVS Logic Circuit with Decoupled Outputs", *IBM Technical Disclosure Bulletin*, Vol. 27, No. 1B, June 1984, pp. 657-658.

[37] W.J. Craig, J. A. Hiltebeitel, "High Speed Cascode Voltage Switch Logic Circuit", *IBM Technical Disclosure Bulletin*, Vol. 27, No. 1B, June 1984, pp. 667-668.

[38] C.K. Erdelyi, W.R. Griffin, and Ralph D. Kilmoyer, "Cascode Voltage Switch Logic Design", *VLSI Design*, Volume V., No 10, October 1984, pp. 78-86.

[39] C.K. Erdelyi, "Random Logic Design Utilizing Single-ended Cascode Voltage Switch Circuit in NMOS", *Proc. IEEE Custom Integrated Circuits Conf.*, Rochester, New York, 1984, pp. 145-149.

[40] "Cascode Voltage Switch Logic", *IBM Technical Disclosure Bulletin*, Vol. 27, No. 10B, March 1985, pp. 6007.

[41] "Stabilizing Cascode Voltage Switch Logic", *IBM Technical Disclosure Bulletin*, Vol. 27, No. 10B, March 1985, pp. 6015.

[42] "Clocked Differential Cascode Voltage Switch Logic Circuit", *IBM Technical Disclosure Bulletin*, Vol. 27, No. 11, April 1985, pp. 6779.

[43] "Single Ended Cascode Voltage Switch Logic Circuit", *IBM Technical Disclosure Bulletin*, Vol. 27, No. 11, April 1985, pp. 6789.

[44] "Pseudo-Clocked Cascode Voltage Switch Logic System", *IBM Technical Disclosure Bulletin*, Vol. 28, No. 6, November 1985, pp. 2536.

[45] William R. Griffin, *Private Communication*, October 1988.

OTHER LOGIC FAMILIES

[46] J.A. Pretorious, A.S. Shubat, and C.A.T.Salama, "Latched Domino CMOS Logic", *IEEE J. Solid State Circuits*, Vol. SC-21, No. 4, August 1986, pp. 514-522.

[47] Shih-Lien Lu, "Implementation of Iterative Networks with CMOS Differential Logic", *IEEE J. Solid State Circuits*, Vol. 23, No. 4, August 1988, pp. 1013-1017.

[48] Timothy A. Grotjohn and Bernd Hoefflinger, "Sample-Set Differential Logic (SSDL) for Complex High-Speed VLSI", *IEEE J. Solid State Circuits*, Vol. SC-21, No. 2, April 1986, pp. 367-369.

[49] R.H. Krambeck, C.M. Lee, and H.S. Law, "High-Speed Compact Circuits with CMOS", *IEEE J. Solid State Circuits*, Vol. SC-17, No. 3, June 1982, pp. 614-619.

[50] N.F. Goncalves, and H.J. De Man, "NORA: A racefree Dynamic CMOS Technique for Pipelined Logic Structures", *IEEE J. Solid State Circuits*, Vol. SC-18, No. 3, June 1983, pp. 261-266.

DCVSL SYNTHESIS PROGRAM ("ntree" reference)

[51] Randal E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Transactions on Computers*, Vol. C-35, No. 8, August 1986, pp. 677-691.

[52] C. Y. Lee, "Representation of switching circuits by binary-decision programs", *Bell. Syst. Tech Journal*, Vol. 38, July 1959, pp. 985-999.

DESIGN AND LAYOUT OF DCVSL:

[53] Kan M. Chu and David I. Pulfrey, "Design Procedures for Differential Cascode Voltage Switch Logic", *IEEE J. Solid State Circuits*, Vol. SC-21, No. 6, December 1986, pp. 1082-1087.

[54] P. S. Hauge, M. Schlag, C. K. Wong, and E. J. Yoffa, "Method for Improving Cascode-Switch Macro Wirability", *IBM Technical Disclosure Bulletin*, Vol. 27, No. 9, February 1985, pp. 5235-5239.

[55] M. D. F. Schlag, E. J. Yoffa, P. S. Hauge, and C. K. Wong, "A Method for Improving Cascode-Switch Macro Wirability", *IEEE Trans. on Computer-Aided Design*, Vol. CAD-4, No. 2, April 1985, pp. 150-155.

[56] Peter S. Hauge and Ellen J. Yoffa, "ACORN: A system for CVS macro design by tree placement and tree customization", *IBM J. Research and Development*, Vol. 28, No. 5, September 1984, pp. 596-602.

ASYNCHRONOUS DESIGN

[57] David Misunas, "Petri Nets and Speed Independent Design", *Communications of the ACM*, Volume 16, Number 8, August 1973, pp. 474-481.

[58] S.Y. Kung, R.J.Gal-Ezer, "Synchronous versus Asynchronous Computation in Very Large Scale Integrated (VLSI) Array Processors", *SPIE, Real Time Signal Processing*,Vol 341, 1982, pp. 53-64.

[59] Tam-Anh Chu, "Synthesis of Self-Timed VLSI Circuits from Graph-theoretic Specifications", *Proceedings of ICCD*, 1987.

[60] Tam-Anh Chu, "Synthesis of Self-Timed VLSI Circuits from Graph-theoretic Specifications", *Ph.D. Thesis*, Dept. of EECS, Massachusetts Institute of Technology, MIT/LCS/TR-393,June 1987.

[61] Tam-Anh Chu, "Design of VLSI Asynchronous FIFO Queues for Packet Communication Networks", *Proceedings of the International Conference on Parallel Processing*, 1986, pp. 397-400.

[62] Tam-Anh Chu, Lance A. Glasser, "Synthesis of Self-timed Control Circuits from Graphs: An Example", *Proc. IEEE 1986 ICCD*, October 1986, pp. 565-571.

[63] E.W. Dijkstra, "Guarded Commands, Nondeterminancy and Formal Derivation of Programs", *Communications of the ACM* **18***(8)*, August 1975, pp. 453-457.

[64] T. Meng, R.W. Brodersen, and D.G. Messerschmitt, "Asynchronous Logic Synthesis for Signal Processing from High Level Specifications", *IEEE ICCAD 87 Digest of Technical Papers*, November 1987.

[65] T. Meng, G.M. Jacobs, R.W. Brodersen, and D.G. Messerschmitt, "Asynchronous Processor Design for Digital Signal Processing", presented at *IEEE ICASSP 1988*, New York, New York, April 1988.

[66] T. Meng, "Asynchronous Processor Design for Digital Signal Processing", *Ph.D. Thesis*, University of California, Berkeley, December 1988.

[67] M. R. Greenstreet, J. Straunstrup, and T. E. Williams, "Designing Iterative Self-Timed Circuits", *IEEE Transactions on Computers*, 1988.

[68] G.M. Jacobs, "Self-Timed Integrated Circuits for Digital Signal Processing Applications", presented at *IEEE VLSI Workshop 1988*, Monterey, California, November 1988. (also published in *VLSI Signal Processing, Vol III*, IEEE Press, New York (1988))

[69] Anton Stoelzle, *Private Communication*, University of California, Berkeley, September 1989.

STATE GRAPH TO LOGIC CIRCUITS

[70] R. K. Brayton, G. Hachtel, C. T. McMullen and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Acedemic Publishers, 1984.

[71] R. K. Brayton, R. L. Rudell, A. L. Sangiovanni-Vincentelli and A. Wang, "MIS : A Multi-Level Logic Synthesis System", *IEEE Trans. on Computer Automated Design*, 1987.

LAGER, MACROCELL BASED DESIGN TOOLS

[72] Stephen P. Pope, "Automatic Generation of Signal Processing Integrated Circuits", *Ph.D. Thesis, University of California, Berkeley*, UCB/ERL Memorandum No. M85/11, February 22, 1985.

[73] P.A. Ruetz, R. Jain, C.-S. Shung, J.M. Rabaey, G.M. Jacobs, and R.W. Brodersen, "Automatic Layout Generation of Real-Time Digital Image Processing Circuits", presented at the *IEEE Custom Integrated Circuits Conference*, Rochester, NY, May 1986.

[74] R.W. Brodersen et. al., "LagerIV Cell Library Documentation", Electronics Research Laboratory, University of California, Berkeley, June 23, 1988.

CLOCK SKEW AND DISTRIBUTION

[75] Mehdi Hatamian and Glenn L. Cash, "High Speed Signal Processing, Pipelineing, and VLSI", *Proceedings of IEEE Int. Conference on Acoustics, Speech, and Signal Processing*, Tokyo, Japan, April 1986, pp. 1173-1176.

[76] Mehdi Hatamian and Glenn L. Cash, " A 70MHz 8-bit X 8-bit Parallel Pipelined Multiplier in 2.5-um CMOS", *IEEE J. Solid State Circuits*, Vol. SC-24, No. 4, August 1986, pp 505-513.

[77] D.K. Jeong, G. Borriello, D.A. Hodges, and R.H.Katz, "Design of PLL-Based Clock Generation Circuits", *IEEE J. Solid State Circuits*, Vol. SC-22, No. 2, April 1987, pp 255-261.

MULTIPLIERS

[78] M.D. Ercegovac and J.G. Nash, "An Area-Time Efficient VLSI Design of a Radix-4 Multiplier", *Proceedings of IEEE International Conference on Computer Design*, 1983, pp 684-687.

[79] J.Y. Lee, H.L. Garvin, and C.W.Slayman, "A High-Speed High-Density Silicon 8x8-bit Parallel Multiplier", *IEEE J. Solid State Circuits*, Vol. SC-22, No. 1, February 1987, pp. 35-40.

[80] T.G. Noll, D. Schmitt-Landsiedel, H. Klar, and G. Enders, "A Pipelined 330-MHz Multiplier", *IEEE J. Solid State Circuits*, Vol. SC-21, No. 3, June 1986, pp. 411-417.

[81] D.A. Henlin, M.T.Fertsch, M. Mazin, and E.T. Lewis, "A 16-bit X 16-bit Pipelined Multiplier Macrocell", *IEEE J. Solid State Circuits*, Vol. SC-20, No. 2, April 1985, pp. 542-547.

[82] Luc De Vos, "Architectuur en Circuit-Ontwerp Van Arithmetische Bouwblokken voor Hoge-Snelheid Digitale Signaalverwerking", *Masters Thesis*, Katholikek Universiteit Leuven, Belgium, 1984-85.

SELF-TIMED MACROCELLS

[83] T.E. Williams, M.Horowitz, R.L. Alverson, and T.S. Yang, "A Self-Timed Chip for Division", *Advanced Research in VLSI, Proc. of 1987 Stanford Conference*, March 1987, pp. 75-96.

Appendix A

ntree Program Documents

1. Manual Page

The manual page for the *ntree* program is provided below for reference.

NTREE(1)

ntree - is a program that aids in the design of logic gates that are implemented using the Differential Cascode Voltage Switch Logic topology. This topology generates both the logic function and its complement. It expects complementary inputs, i.e. inputs coming from another dcvsl stage. The two output nodes are precharged using P-channel devices and then a tree of N-channel devices performs the actual logical operation. *Ntree* generates the tree of N-channel transistors.

SYNOPSIS

ntree inputfile

DESCRIPTION

Ntree reads a logical function expression from the input file and generates a spice file output that contains the necessary transistors to implement that expression in DCVSL technology. The format of the input file is discussed below. **Ntree** forms a directed graph representing the logic function and then reduces the graph so that it contains the minimum number of vertices (and therefore transistors). Since the *ordering* of the inputs in the logic expression affects the number of vertices in the graph for certain logic functions, **ntree** tries *all* of the different orderings and chooses the graph with the least number of vertices. This exhaustive search technique is not adequate for a large number of inputs, so the number of independent input variables allowed in the logic expression is limited to *8*. This is consistent with CMOS design, where a large number of transistors in series can cause speed degradation. The output file contains the transistors of the NMOS tree for a DCVSL gate. The pre-charge devices, output inverters, and optional feedback devices for static gates must be added by the user after **ntree** is run.

INPUT FILE FORMAT

For lack of a sophisticated parser, the input format does not match that of programs such as *eqntott*. Instead, the logic operations are specified in a lisp-like format. Each operation must be surrounded by parentheses. The number of inputs for each operation is unrestricted and all inputs must be numerical and greater than zero. Empty lines are ignored in the input file and comments are any lines that begin with the character '#'. An example of an input file for **ntree** is shown below:

```
# This is a comment
(example gate 1 (or (and 1 2 3) (nor 3 4) (xor 1 3) (not 2)))
```

The allowed logical operations are: *and*, *or*, *nand*, *nor*, *xor*, *xnor*, and *not*. Only the *not* operation may have less than two operands. The information between the first and second left parenthesis in the input can be whatever the user desires to describe the gate.

OUTPUT FILES

The SPICE output file is written to filename.*spi*. It contains the logic expression being implemented, a mapping of the input numbers to circuit node numbers, and finally a series of MOSFET lines with the actual transistors in the tree. **Ntree** writes information to the standard output also. (Use redirected I/O to save this information to a file.) The logical function expression is echoed to the output along with the number of unique inputs (depth). The program prints out the number of vertices in the graph along with the number of series connected devices in the tree for the best input orderings. Program heuristics will choose trees with slightly more vertices in preference to those that contain less vertices, but more transistors in series, as a speed consideration. The program also prints to the standard output a textual representation of the final function graph using tabbing to indicate branches.

REFERENCES

"Graph Based Algorithms for Boolean Function Manipulation", by Randal E. Bryant, *IEEE Transactions on Computers*, Vol. C-35, No. 8, August 1986.

AUTHOR

Gordon Jacobs.

2. Example

Below is an example of running *ntree*.

Input file

```
# ALU logic circuits
#  Complementation is accomplished by muxes on input
#  choosing between 1, 1*, 2, 2*
#  Inputs: 1,2
#  Control: 3,4 -> binary coded:
#               0  not-used
#               1  or
#               2  and
#               3  xor
#
#
(logic outputs
        (or
            (and 3 (not 4) (or 1 2))
            (and (not 3) 4 (and 1 2))
            (and 3 4 (xor 1  2))
        )
)
#
```

SPICE Output File created

```
         NMOS Tree for (logic outputs    )
*
* Logic Expression: (or       (and 3 (not 4) (or 1 2))
(and (not 3) 4 (and 1 2))       (and 3 4 (xor 1  2))  )
*
* This file generated by ntree on Tue Jul 14 14:34:57 1987
*
*  NODE ASSIGNMENTS:
*  GND  =  0     Vdd  =  100
*  Pbulk = 102   Nbulk = 101
*  (Complement of )Input Number 1  is node (11) 1
*  (Complement of )Input Number 2  is node (12) 2
*  (Complement of )Input Number 3  is node (13) 3
*  (Complement of )Input Number 4  is node (14) 4
*  F_OUT = 21   F_BAR_OUT = 20
*
*     D   G   S   B
m1   24  11   0  101  NMOS
m2   20  12  24  101  NMOS
m3   26   2  24  101  NMOS
m4   20  13  26  101  NMOS
m5   21   3  26  101  NMOS
m6   28   1   0  101  NMOS
m7   26  14  28  101  NMOS
m8   29   4  28  101  NMOS
m9   26  12  29  101  NMOS
m10  30   2  29  101  NMOS
m11  21  13  30  101  NMOS
m12  20   3  30  101  NMOS
***end***
```

Information sent to stdout

```
CVSL Logic Minimization Program by Gordon Jacobs
Rev 1.0

Date:  Tue Jul 14 14:34:57 1987

********* INPUT LISTING ************  Input File:  alu5 ****

(logic outputs
(or
    (and 3 (not 4) (or 1 2))
    (and (not 3) 4 (and 1 2))
    (and 3 4 (xor 1  2))
)
)

Gate Name: logic outputs
Number of unique inputs (depth) = 4
***********************************************************
       INPUTS: 3 4 1 2
For indices:   1 2 3 4   G = 10   L = 4
For indices:   1 3 2 4   G = 9   L = 4
For indices:   2 3 1 4   G = 9   L = 3
For indices:   4 2 1 3   G = 8   L = 3

G => number of vertices.   L => number of series devices.
Gmin = 8/L = 3.  Lmin = 3/G = 8

--------------- FUNCTION GRAPH: --------------------

-> INPUT ORDERING: 1 4 2 3

Vertex: INPUT 1  index = 1  id = 1
`with low(0) side --
|   Vertex: INPUT 2  index = 3  id = 2
|   `with low(0) side --
|   |   ***** ZERO *****
|   `with high(1) side --
|   |   Vertex: INPUT 3  index = 4  id = 4
|   |   `with low(0) side --
|   |   |   ***** ZERO *****
|   |   `with high(1) side --
|   |   |   ***** ONE ******
`with high(1) side --
|   Vertex: INPUT 4  index = 2  id = 6
|   `with low(0) side --
|   |   Vertex: INPUT 3  index = 4  id = 4
|   `with high(1) side --
|   |   Vertex: INPUT 2  index = 3  id = 7
|   |   `with low(0) side --
|   |   |   Vertex: INPUT 3  index = 4  id = 4
|   |   `with high(1) side --
|   |   |   Vertex: INPUT 3  index = 4  id = 8
|   |   |   `with low(0) side --
|   |   |   |   ***** ONE ******
|   |   |   `with high(1) side --
|   |   |   |   ***** ZERO *****

Memory used:
 Vertices used = 18
 Lists used = 8
 Trees used = 32

*** end ***
```

3. Program Source Code

The *ntree* program source code is too lengthy for inclusion in this document. The code can however be obtained by contacting:

Professor Robert W. Brodersen
University of California, Berkeley
Department of EECS, Cory Hall
Berkeley, California 94720

Appendix B

Assembler Program Documents

1. Manual Page for asm

The manual page for the *asm* program is provided below for reference.

ASM(1)

NAME

asm - convert asynchronous processor assembly code into program ROM data for THOR simulations.

SYNOPSIS

asm [-dsh] *file*

DESCRIPTION

asm is a program that converts an *asm* file (*file.asm*) written for the asynchronous digital signal processor datapath into program ROM code that is used by the THOR simulator for simulating the signal processing program on a model of the actual hardware. A total of three output files are created. The file *file.out* contains general information about the assembler actions. For each instruction, the assigned program counter address is shown along with a table of the various control signals that the horizontal micro-code for the instruction contains. The RAM Read and Write addresses are given first with an empty Write address meaning that the write is disabled for that instruction cycle. Next, the number of bits that the barrel shifter shifts left is shown. If the number is followed by a '.' (period), then sign extension is indicated. The next entry of the table is the OP code which shows which logical or arithmetic operation the ALU performs along with the state of the two signals which zero the A-input (zA) and the B-input (zB) of the ALU. The B-input is from the Accumulator. The IN and OUT table entries indicate whether that instruction will read from/write to the input/output ports respectively. The WRSEL field tells whether the RAM is written from the low order Accumulator bits(2), the high order Accumulator bits(1) or the local feedback path(0). The MULT? field tells when the multiplier is selected in place of the barrel shifter for the second stage of the datapath. Finally, the CROM entry gives the address of the coefficient ROM that is used for a multiplication if it exists.

At the bottom of the *file.out* file, is a listing of the hash table entries for the assembler. These are filled in by the assembler for all EQUate statements and labels. They give the final address of the labels and symbolic names.

The two output files *file.ROM_L* and *file.ROM_H* are each intended to be read by the THOR simulator for the asynchronous processor. Because the Program ROM is 40-bits wide and THOR can only work with 32-bit numbers, two files are created. *file.ROM_L* contains the lower 32-bits of the program ROM entries in Hexidecimal format. *file.ROM_H* contains the upper 8-bits of the program ROM entries. Each file begins with a line of the form:

```
%num
```

where *num* is the number of instructions to follow.

OPTIONS

The command line switches are:

-d
Generate debug information. Usually should not be used by normal users.

-s
Set the warnings flag. Prints out extra warnings about the assembly code.

-h
Prints out usage.

SEE ALSO

ROMconvert(1)

BUGS

Warnings not implemented yet.

AUTHOR

Gordon Jacobs, University of California, Berkeley (jac@zion.berkeley.edu)

2. Manual Page for ROMconvert

ROMconvert(1)

NAME

ROMconvert - convert THOR simulator files of Program ROM code for the asynchronous processor into a Lager parameter file that can be used to generate the physical ROM.

SYNOPSIS

ROMconvert *file [.asm]*

DESCRIPTION

ROMconvert is a program that converts the THOR simulation files containing program ROM code for the asynchronous processor into parameter files that the Lager system can use to generate the actual physical program ROM. The input file name *file [.asm]* is used as a root for the files that are actually read, which are produced with **asm**. The files *file.ROM_L* and *file.ROM_H* are read and the parameter file for Lager is written to the standard output.

OPTIONS

No command line options.

SEE ALSO

asm(1), LagerIV documentation Manuals.

BUGS

None to my knowledge.

AUTHOR

Gordon Jacobs, University of California, Berkeley (jac@zion.berkeley.edu)

3. Example

Below is an example of running the assembler:

Input file

```
/*
 *   FILTER ASSEMBLY CODE FOR FIR Filter in "fir_design2"
 *      (using multiplier)
 *
 */

/*************************************************************************
Input is stored in RAM location 01.
All Coefficients are in CROM.

State Variables: Ram Location 0-15 are locations for delay line
Coefficients: CROM locations 0-7 are eight required coefficients.
Since the filter is linear phase, the impulse response is mirrored.

**************************************************************************/
DECL
EQU del_0 0
EQU del_1 1
EQU del_2 2
EQU del_3 3
EQU del_4 4
EQU del_5 5
EQU del_6 6
EQU del_7 7
EQU del_8 8
EQU del_9 9
EQU del_10 10
EQU del_11 11
EQU del_12 12
EQU del_13 13
EQU del_14 14
EQU del_15 15
EQU c1 0
EQU c2 1
EQU c3 2
EQU c4 3
EQU c5 4
EQU c6 5
EQU c7 6
EQU c8 7
START

init: acc = 0;
(del_0=acc) = 0;
(del_1=acc) = 0;
(del_2=acc) = 0;
(del_3=acc) = 0;
(del_4=acc) = 0;
(del_5=acc) = 0;
(del_6=acc) = 0;
(del_7=acc) = 0;
(del_8=acc) = 0;
(del_9=acc) = 0;
(del_10=acc) = 0;
(del_11=acc) = 0;
(del_12=acc) = 0;
(del_13=acc) = 0;
(del_14=acc) = 0;
(del_15=acc) = 0;
acc = acc;
acc = acc;

sample: acc = 0 | del_15*c1, (del_0 = in);
```

```
acc = acc + (del_15=del_14)*c2;
acc = acc + (del_14=del_13)*c3;
acc = acc + (del_13=del_12)*c4;
acc = acc + (del_12=del_11)*c5;
acc = acc + (del_11=del_10)*c6;
acc = acc + (del_10=del_9)*c7;
acc = acc + (del_9=del_8)*c8;
acc = acc + (del_8=del_7)*c8;
acc = acc + (del_7=del_6)*c7;
acc = acc + (del_6=del_5)*c6;
acc = acc + (del_5=del_4)*c5;
acc = acc + (del_4=del_3)*c4;
acc = acc + (del_3=del_2)*c3;
acc = acc + (del_2=del_1)*c2;
acc = acc + (del_1=del_0)*c1;
(out=acc) = acc;
goto sample;
```

Output File created (*file.out*)

```
ASSEMBLER OUTPUT


  PC Raddr Waddr shift OP IN OUT WRSEL MULT? CROM

   0:   0   0   + zAzB  0  0   0   N
   1:   0   0   0    + zAzB  0  0   1   N
   2:   0   1   0    + zAzB  0  0   1   N
   3:   0   2   0    + zAzB  0  0   1   N
   4:   0   3   0    + zAzB  0  0   1   N
   5:   0   4   0    + zAzB  0  0   1   N
   6:   0   5   0    + zAzB  0  0   1   N
   7:   0   6   0    + zAzB  0  0   1   N
   8:   0   7   0    + zAzB  0  0   1   N
   9:   0   8   0    + zAzB  0  0   1   N
  10:   0   9   0    + zAzB  0  0   1   N
  11:   0  10   0    + zAzB  0  0   1   N
  12:   0  11   0    + zAzB  0  0   1   N
  13:   0  12   0    + zAzB  0  0   1   N
  14:   0  13   0    + zAzB  0  0   1   N
  15:   0  14   0    + zAzB  0  0   1   N
  16:   0  15   0    + zAzB  0  0   1   N
  17:   0   0   + zA     0  0   0   N
  18:   0   0   + zA     0  0   0   N
  19:  15   0   0    |  zB   1  0   0   Y   0
  20:  14  15   0    +       0  0   0   Y   1
  21:  13  14   0    +       0  0   0   Y   2
  22:  12  13   0    +       0  0   0   Y   3
  23:  11  12   0    +       0  0   0   Y   4
  24:  10  11   0    +       0  0   0   Y   5

  PC Raddr Waddr shift OP IN OUT WRSEL MULT? CROM

  25:   9  10   0    +       0  0   0   Y   6
  26:   8   9   0    +       0  0   0   Y   7
  27:   7   8   0    +       0  0   0   Y   7
  28:   6   7   0    +       0  0   0   Y   6
  29:   5   6   0    +       0  0   0   Y   5
  30:   4   5   0    +       0  0   0   Y   4
  31:   3   4   0    +       0  0   0   Y   3
  32:   2   3   0    +       0  0   0   Y   2
  33:   1   2   0    +       0  0   0   Y   1
  34:   0   1   0    +       0  0   0   Y   0
  35:   0   0   + zA     0  1   1   N
  36: Branch to (19)

  Notes: Blank Waddr means Write Enable OFF
         "." => shifter sign extension ON
         "~" => one's complement on ALU A-input
         "zB" => zero B (accumulator) input to ALU
         "zA" => zero A (RAM) input to ALU
         WRSEL=2  => Write LOW order bits of accumulator

------- HASH TABLE ENTRIES --------
  Location 80:    Name = init  Value = 0
  Location 96:    Name = del_0  Value = 0
  Location 97:    Name = del_1  Value = 1
  Location 98:    Name = del_2  Value = 2
  Location 99:    Name = del_3  Value = 3
  Location 100:    Name = del_4  Value = 4
  Location 101:    Name = del_5  Value = 5
  Location 102:    Name = del_6  Value = 6
  Location 103:    Name = del_7  Value = 7
  Location 104:    Name = del_8  Value = 8
  Location 105:    Name = del_9  Value = 9
  Location 145:    Name = del_10  Value = 10
  Location 146:    Name = del_11  Value = 11
Location 147:    Name = del_12  Value = 12
Location 148:    Name = del_13  Value = 13
Location 149:    Name = del_14  Value = 14
Location 150:    Name = del_15  Value = 15
Location 151:    Name = c1  Value = 0
Location 152:    Name = c2  Value = 1
Location 153:    Name = c3  Value = 2
Location 154:    Name = c4  Value = 3
Location 155:    Name = c5  Value = 4
Location 156:    Name = c6  Value = 5
Location 157:    Name = c7  Value = 6
Location 158:    Name = c8  Value = 7
Location 286:    Name = sample  Value = 19
```

ROM Code File created (*file.ROM_L*)

```
ROM CODE FROM FILE: fir2.asm
%37
00300000
10304000
10304080
10304100
10304180
10304200
10304280
10304300
10304380
10304400
10304480
10304500
10304580
10304600
10304680
10304700
10304780
00100000
00100000
1e60000f
1600078e
1600070d
1600068c
1600060b
1600058a
16000509
16000488
16000407
16000386
16000305
16000284
16000203
16000182
16000101
16000080
20104000
50000013
```

Result of running ROMconvert:

```
;;;
;;; Parameter File for Program ROM of fir2.asm
;;;

(minterms 37)
(outwidth 40)
(inwidth 8)
(in-plane
(
 "00000000"
 "00000001"
 "00000010"
 "00000011"
 "00000100"
 "00000101"
 "00000110"
 "00000111"
 "00001000"
 "00001001"
 "00001010"
 "00001011"
 "00001100"
 "00001101"
 "00001110"
 "00001111"
 "00010000"
 "00010001"
 "00010010"
 "00010011"
 "00010100"
 "00010101"
 "00010110"
 "00010111"
 "00011000"
 "00011001"
 "00011010"
 "00011011"
 "00011100"
 "00011101"
 "00011110"
 "00011111"
 "00100000"
 "00100001"
 "00100010"
 "00100011"
 "00100100"
)
)

(out-plane
(
 "0000000000000000001100000000000000000000"
 "0000000000010000001100000100000000000000"
 "0000000000010000001100000100000010000000"
 "0000000000010000001100000100000100000000"
 "0000000000010000001100000100000110000000"
 "0000000000010000001100000100001000000000"
 "0000000000010000001100000100001010000000"
 "0000000000010000001100000100001100000000"
 "0000000000010000001100000100001110000000"
 "0000000000010000001100000100010000000000"
 "0000000000010000001100000100010010000000"
 "0000000000010000001100000100010100000000"
 "0000000000010000001100000100010110000000"
 "0000000000010000001100000100011000000000"
 "0000000000010000001100000100011010000000"
 "0000000000010000001100000100011100000000"
 "0000000000010000001100000100011110000000"
 "0000000000000000000100000000000000000000"
```

```
    "0000000000000000000100000000000000000000"
    "0000000000011110011000000000000000001111"
    "0000000100010110000000000000011110001110"
    "0000001000010110000000000000011100001101"
    "0000001100010110000000000000011010001100"
    "0000010000010110000000000000011000001011"
    "0000010100010110000000000000010110001010"
    "0000011000010110000000000000010100001001"
    "0000011100010110000000000000010010001000"
    "0000011100010110000000000000010000000111"
    "0000011000010110000000000000001110000110"
    "0000010100010110000000000000001100000101"
    "0000010000010110000000000000001010000100"
    "0000001100010110000000000000001000000011"
    "0000001000010110000000000000000110000010"
    "0000000100010110000000000000000100000001"
    "0000000000010110000000000000000010000000"
    "0000000000100000001000001000000000000000"
    "0000000001010000000000000000000000010011"
)
)
```

4. Program Source Code

The *asm* and *ROMconvert* program source code is too lengthy for inclusion in this document. The code can however be obtained by contacting:

Professor Robert W. Brodersen
University of California, Berkeley
Department of EECS, Cory Hall
Berkeley, California 94720

Appendix C

Circuit Detail

1. Barrel Shifter

1.1 Floorplan

The floorplan for the barrel shifter is described below. The tiling starts from the bottom-left side of the shifter as shown in Figure 6.1 and proceeds to the right for each new cell and to the top for each new row.

1. Place *bshifter.left.mag*
2. Add-right *DVc.mag*, (26) *DVa.mag*, (3) *DVb.mag*
3. Add-right *bshift.ctldp1.mag*
4. New Row: (32) *buffer.bsh.mag*
5. New Row: (32) *2inmux2.mag*
6. New Row: (4) *bch0 – 8.mag*
7. New Row: (32) *2inmux2.mag*
8. New Row: (8) *bch0 – 4.mag*
9. New Row: (32) *2inmux2.mag*
10. New Row: (16) *bch0 – 2.mag*
11. New Row: (32) *2inmux2.mag*
12. New Row: (32) *bch0 – 1.mag*
13. New Row: (16) *dff2.bsh.mag*, *bshifter.top3.mag*

1.2 Miscellaneous Cells

The figures below show schematic diagrams for miscellaneous cells that make up the barrel shifter and were not shown in Chapter 6. Transistor sizes are in λ.

A block diagram for the shifter control slice is shown in Figure C.1. It contains the gated buffers for the shifter control inputs and several other buffers. The names of magic subcells are shown. Each control signal is gated by the **Ibsh** signal input so that the DCVSL 2:1 MUX circuits used in the shifter core can be simplified as explained in Chapter 5. Figure C.2 shows the schematic for a control input *cntl_in2.mag* which

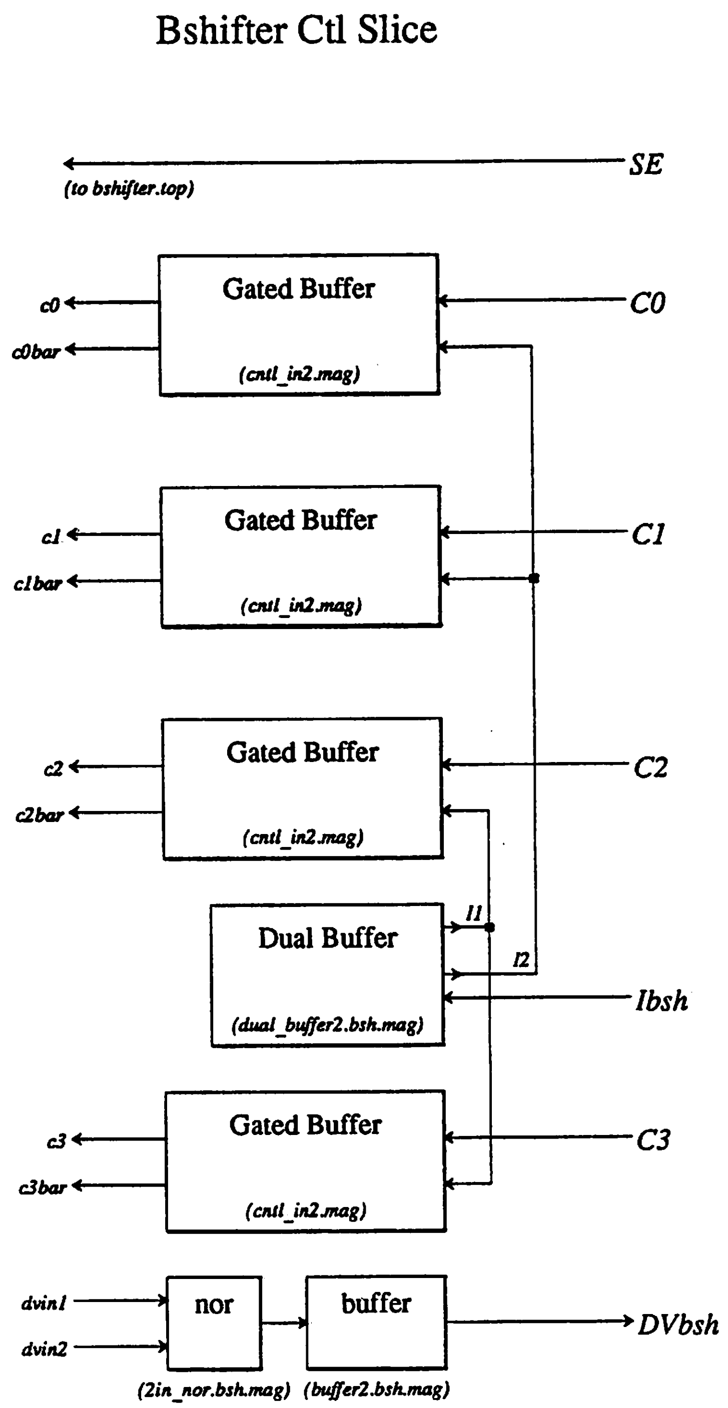


Figure C.1: Block diagram of control slice of barrel shifter.

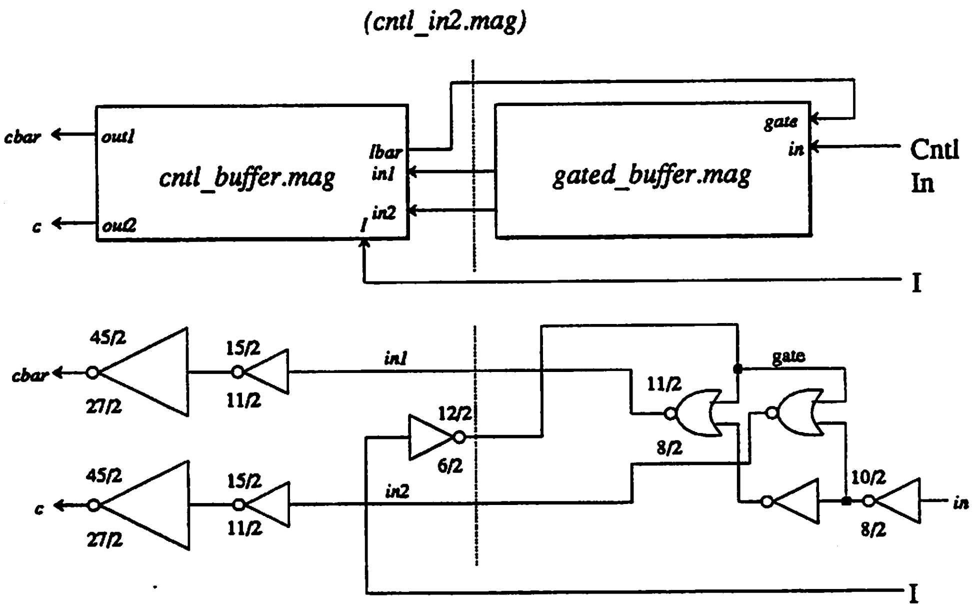


Figure C.2: Control input schematic for barrel shifter.

contains two subcells. Data Valid signals from several bits of the shifter are brought to the control slice where the final **DV** signal is produced. The signals *dvin*1 and *dvin*2 shown in Figure C.1 are active low. A simple NOR gate is used to generate **DV** and the result is buffered before being made available to the outside world. The schematics for the NOR gate and buffer are shown below in Figures C.3 and C.4. The **Ibsh** signal is buffered

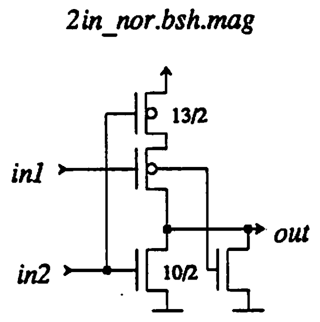


Figure C.3: 2-input NOR gate used in the barrel shifter control slice (*bshift.ctldp1.mag*).

by two parallel buffers each of which drives one half of the shifter MUXes. The schematic for the buffer is shown in Figure C.5. The data bit outputs of the shifter are buffered by the circuit shown in Figure C.6. Several cells are used to derive the completion signal for

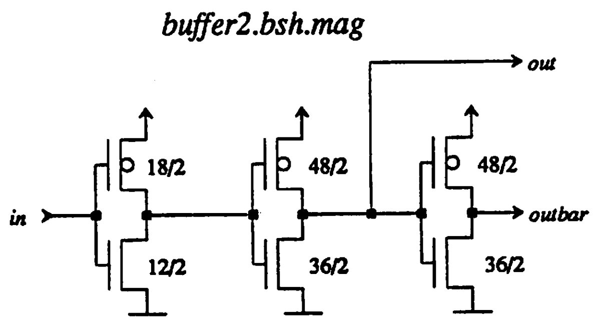


Figure C.4: Buffer cell used in the barrel shifter control slice to buffer $DVbsh$.

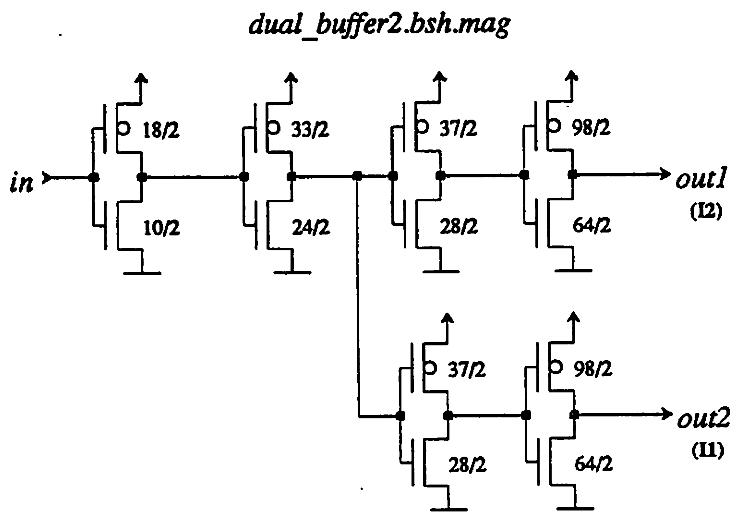


Figure C.5: Buffer circuit used in the barrel shifter control slice to drive **Ibsh**.

the shifter. As explained in Chapter 6, not all outputs are examined for the data valid condition due to the symmetry of the shifter circuitry. The *DVb.mag* cell is used to NOR the complementary outputs of a DCVSL MUX in the last row, generating a $\overline{DV}$ signal for that particular bit. The schematic for the cell is given in Figure C.8. For bits which not used in determining DV, the cell *DVa.mag* is placed and it contains only a routing wire to pass the data output to the edge of the shifter layout. Figure C.7 shows this symbolically. Finally, the cell *DVc.mag* is used for a bit far from the control slice. It contains an extra buffer to drive the capacitance of the long wire spanning the shifter feeding its output to the control slice. The output of a single *DVb* cell and a single *DVc* cell become the *dvin*1 and *dvin*2 signals of the control slice which are sent to the NOR gate generating the final data valid signal. The *bshifter.top3.mag* cell contains a buffer for the *Acki* signal which

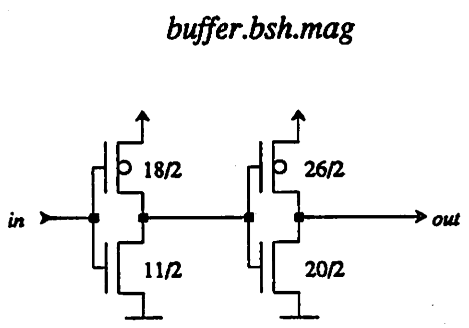


Figure C.6: Buffer circuit used in the barrel shifter at the data outputs.

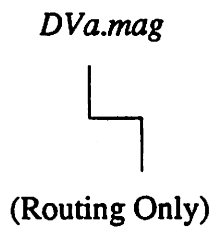


Figure C.7: Routing cell for shifter output bits which are not used in determining the DV signal.

must drive the input data register cells. It also contains the sign extension logic. When the SE control signal is asserted, then the upper 16-bits of the shifter input must be set to the value of the sign bit of the data input. This is accomplished with an AND gate and buffer as shown in Figure C.10. The transistor schematics for *and_buffer.bsh.mag* and

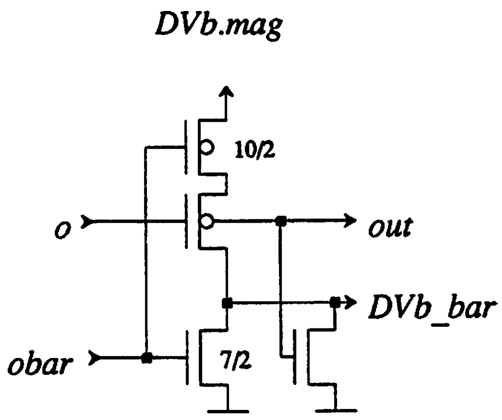


Figure C.8: NOR gate used to derive the completion signal for a single bit in the shifter.

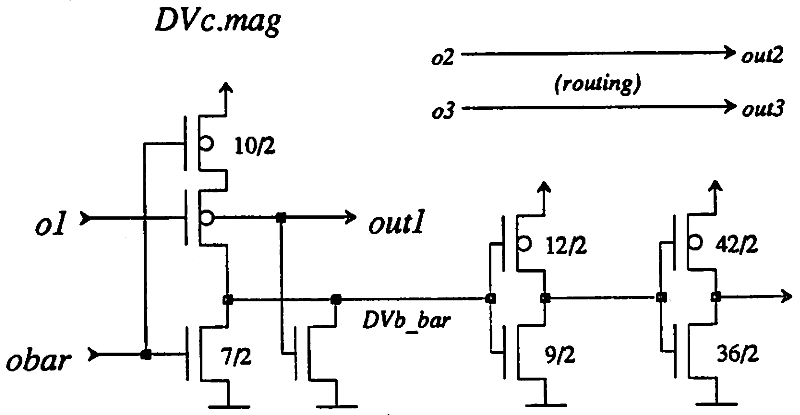


Figure C.9: NOR gate with buffer used to derive the completion signal for a single bit in the shifter.

clkdrv3.bsh.mag are shown in Figures C.11 and C.12. The routing between the MUX

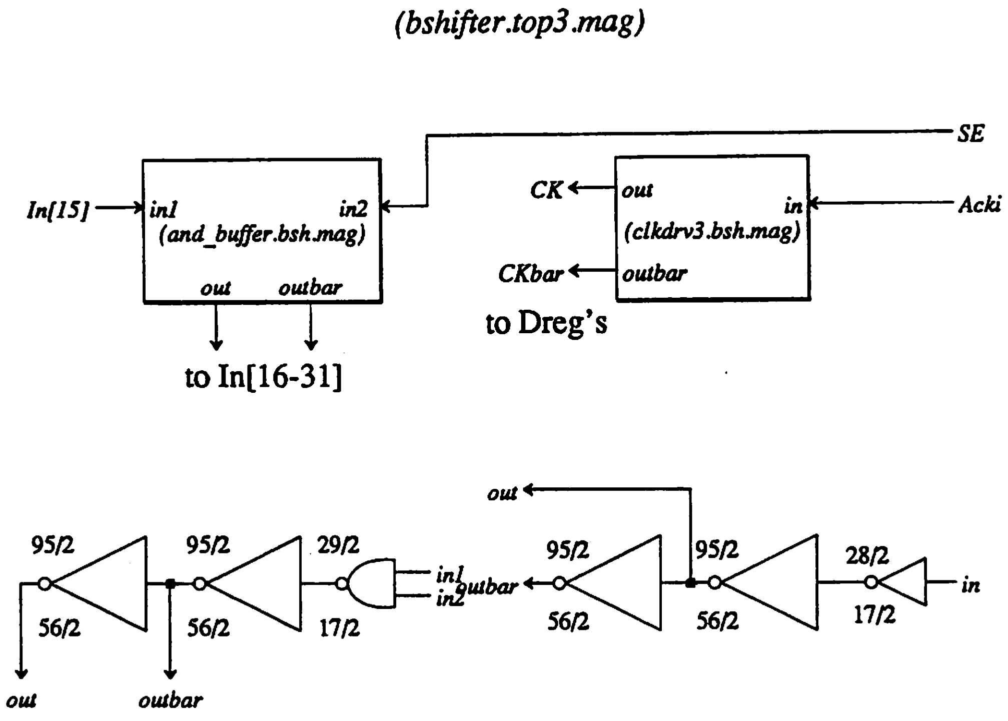


Figure C.10: Block diagram and schematics for bshifter.top cell.

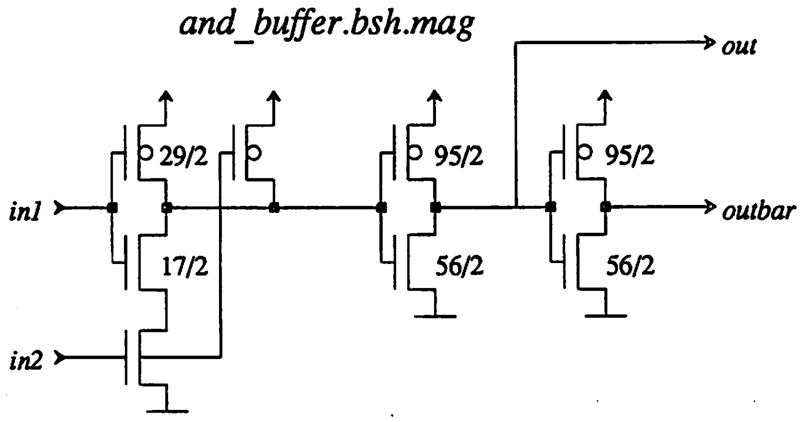


Figure C.11: AND gate with buffer used in the bshifter top cell for sign extension logic.

rows of the barrel shifter is accomplished by four cells. These contain routing wires only as symbolically represented in the next four figures. The number of rows over which the

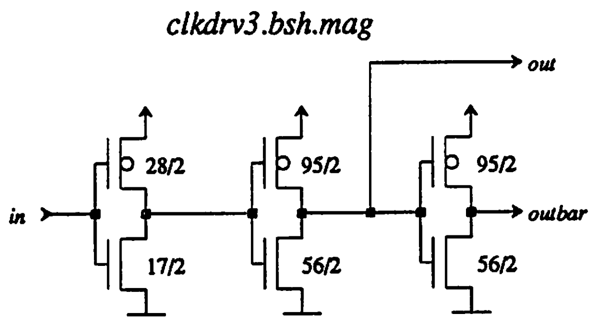


Figure C.12: Clock driver circuit used in barrel shifter top cell to buffer input register clocks.

routing extends is the last number in the name of the cell. So, for example, 32 of the *bch0-1.mag* cells are required but only 16 of the *bch0-2.mag* cells are required for the 32 bit wide shifter.

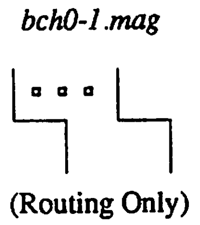


Figure C.13: Routing cell between input register and row 1 of the MUXes in the barrel shifter.

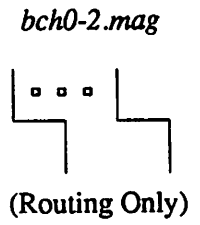


Figure C.14: Routing cell between MUX rows 1 and 2 in the barrel shifter.

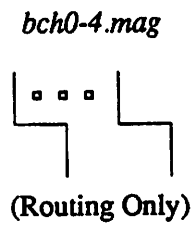


Figure C.15: Routing cell between MUX rows 2 and 3 in the barrel shifter.

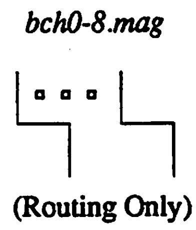


Figure C.16: Routing cell between MUX rows 3 and 4 in the barrel shifter.

2. ALU

The figures below show schematic diagrams for miscellaneous cells that make up the ALU and were not shown in Chapter 6. Transistor sizes are in λ.

2.1 Bitslice

The block diagram of the ALU bitslice is shown in Figure C.17. Subcell names are given in the figure. The A-input of the ALU is stored in a register during computation. The register output can be held low to clear the A-input. Figure C.18 shows the schematic for the flip-flop used in the register. The function to invert the A-input of the ALU is performed with a DCVSL 2:1 MUX which selects either *A* or *Abar* to be fed to the following circuitry. The schematic for the MUX is shown in Figure C.19. The schematics for both the full adder (*adder5.mag*) and the gate performing the logical functions (*logic2.mag*) were shown in Chapter 6. Depending on whether the ALU is doing an addition or logical operation, the output of one of these two gates is passed on to the Accumulator. The selection is made by another DCVSL MUX circuit which is shown in Figure C.20. The Accumulator is incorporated into the ALU and the flip-flop which is used for it is shown in Figure C.21. Data outputs are buffered by the buffer shown in Figure C.22. The B-input

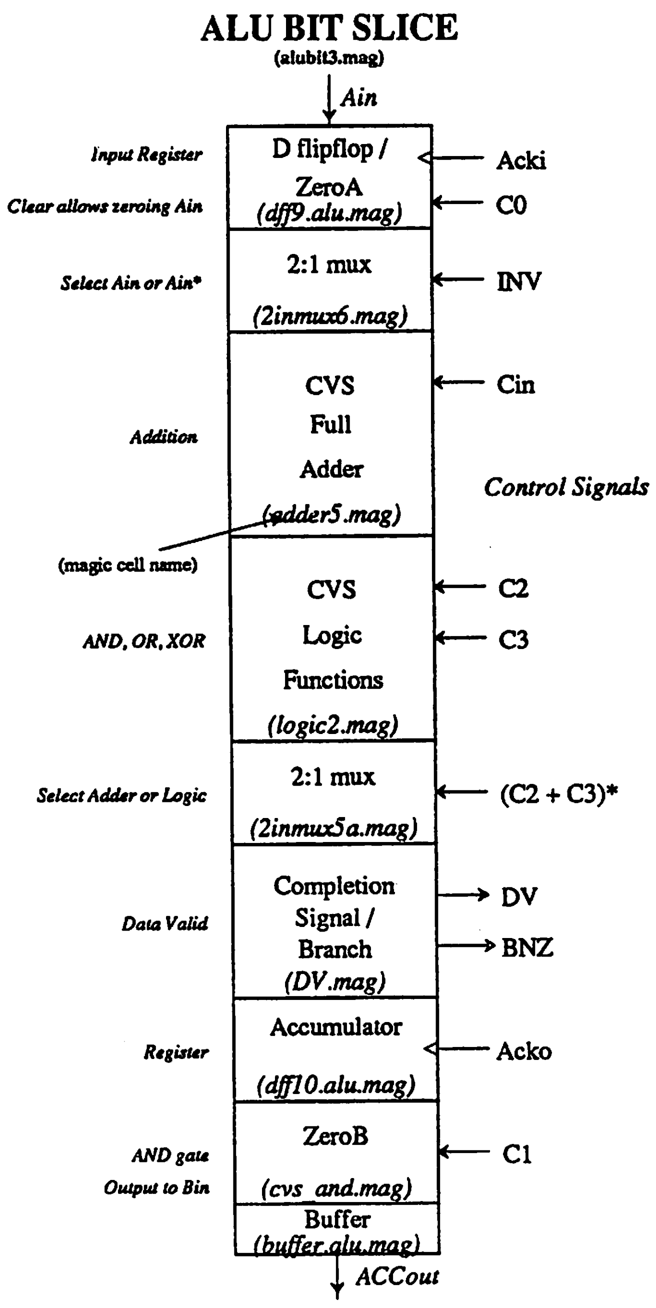


Figure C.17: Diagram of the ALU bitslice with MAGIC subcell names.

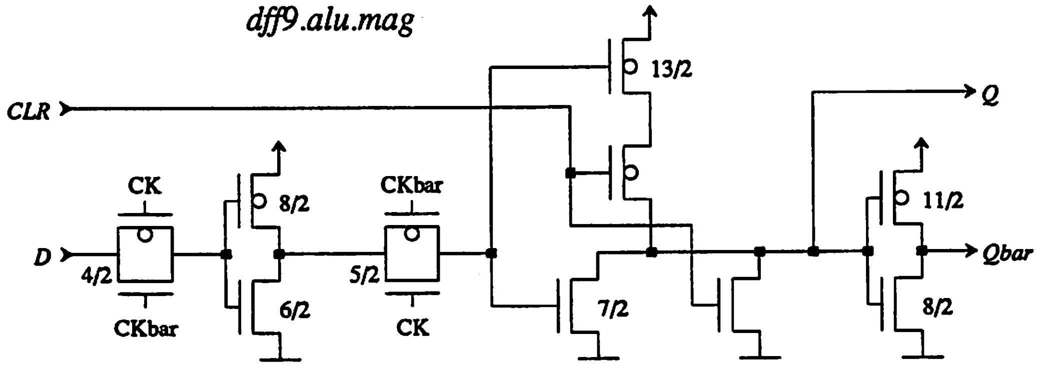


Figure C.18: D flip-flop used for A-input register in the ALU bitslice.

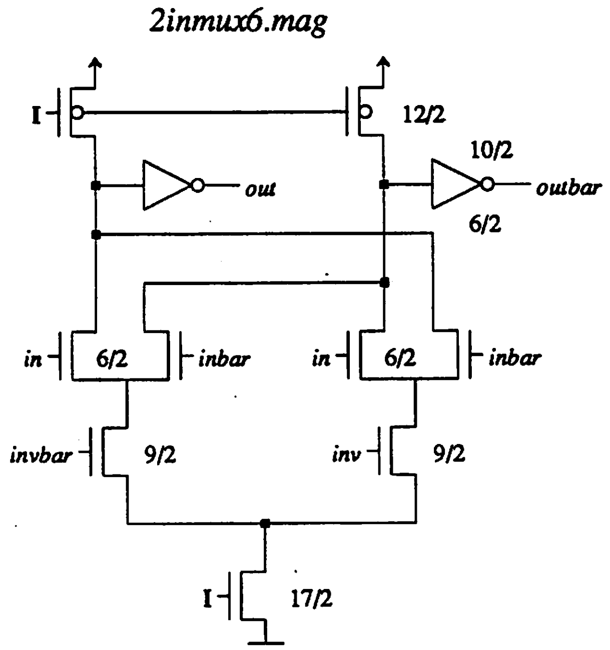


Figure C.19: Schematic of 2:1 MUX used to select *A* or *Abar* input in the ALU bitslice.

2inmux5a.mag

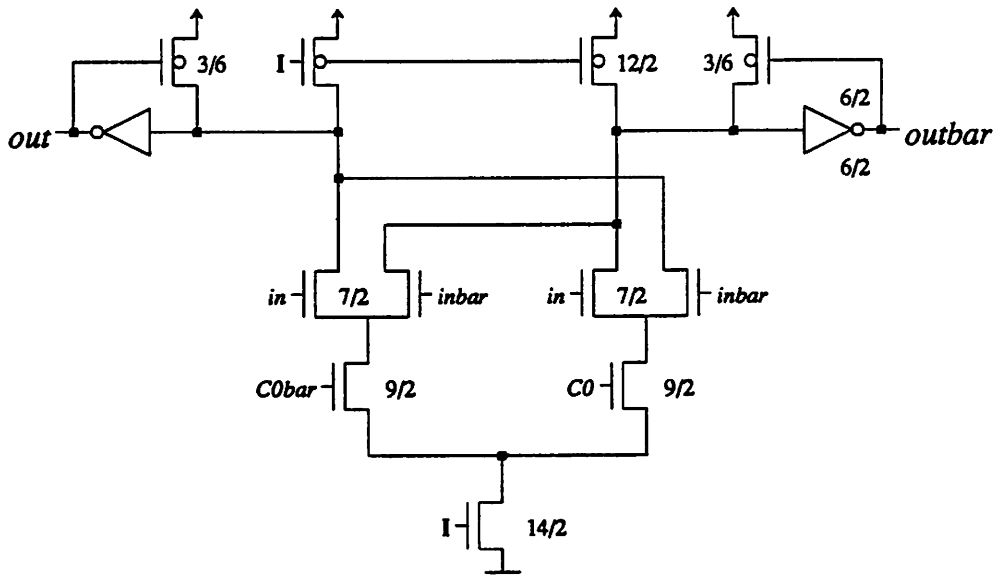


Figure C.20: Schematic of 2:1 MUX used to select logical or addition function in the ALU bitslice.

is zeroed by the use of a DCVSL AND gate which is shown in Figure C.23. Note that the routing of the Accumulator output (which is the B-input to the ALU) to the bitslice gate inputs is done internally to each bit.

2.2 Control Slice

The ALU control slice contains buffering and logic required to interface the bit slices to the control signal inputs. A block diagram of the control slice is shown in Figure C.24. The buffers used on each control input are shown in Figures C.25 and C.26.

The buffers used for the *Acki* and I signal inputs are shown in Figures C.27 and C.28. The SEL control signal which selects the output from the adder or the logical gate is derived from control signal inputs $C2$ and $C3$. The gates to do this are shown in Figure C.29. The completion signal busses dvB and dvC must be fed to one final NOR gate and buffered to form the **DValu** signal fed to the outside world. The schematics for the gates doing this are shown in Figures C.30 and C.31. The gates shown in Figures C.32 and C.33 are also used in the ALU control slice.

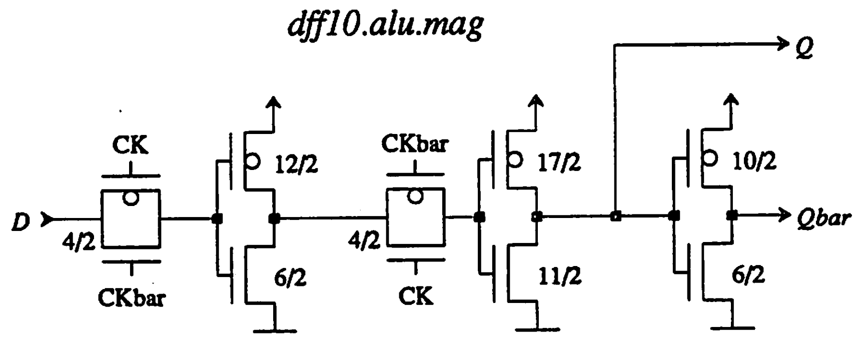


Figure C.21: D flip-flop used for Accumulator register in the ALU bitslice.

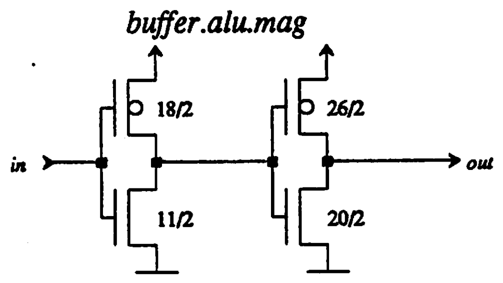


Figure C.22: Buffer cell used in ALU control slice.

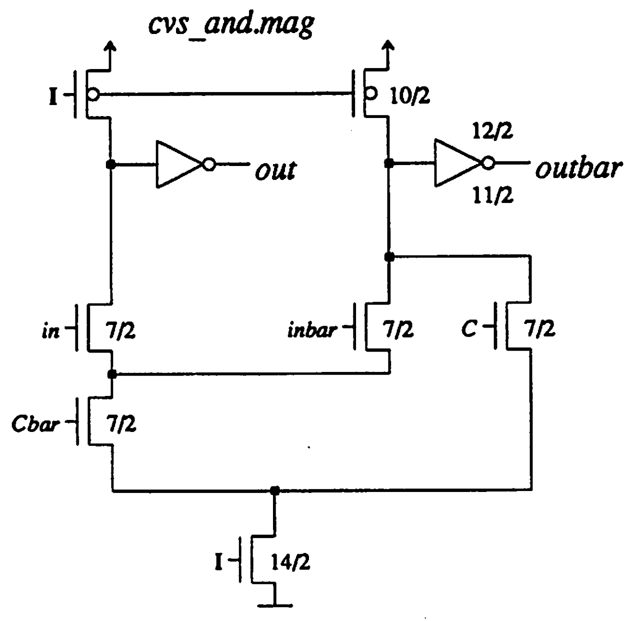


Figure C.23: DCVSL AND gate used to zero the B-input of the ALU in the bitslice.

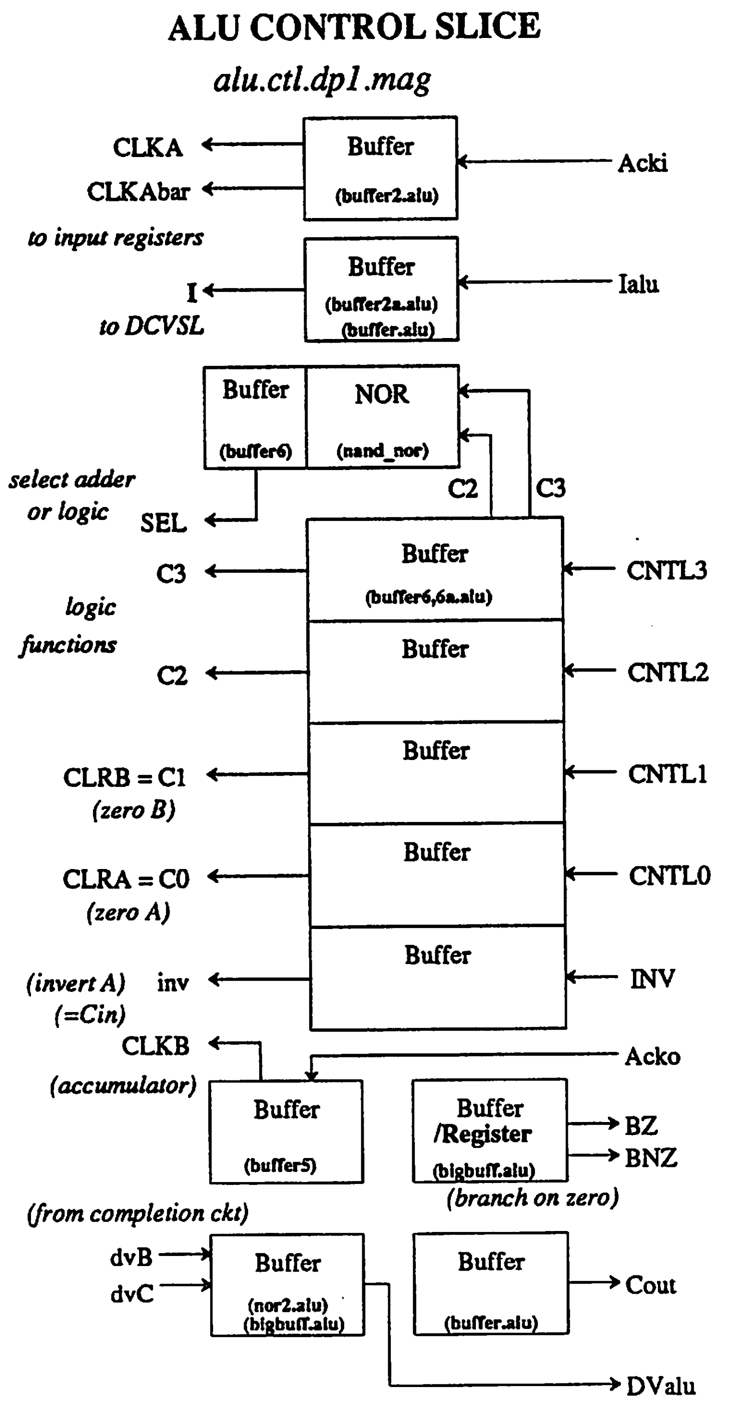


Figure C.24: Block diagram of ALU control slice showing subcell names.

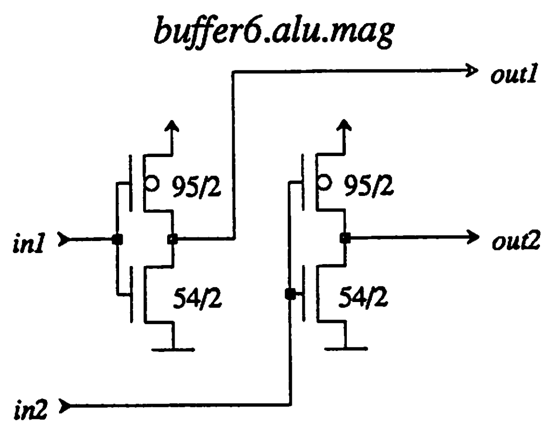


Figure C.25: Buffer used in ALU control slice.

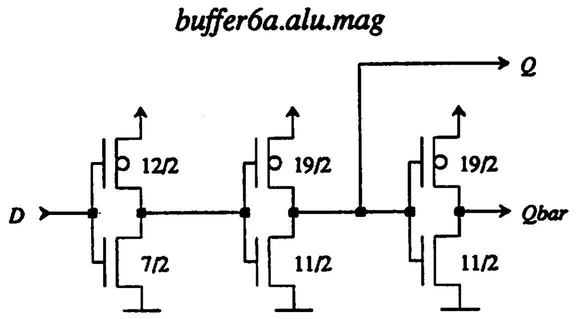


Figure C.26: Buffer used in ALU control slice.

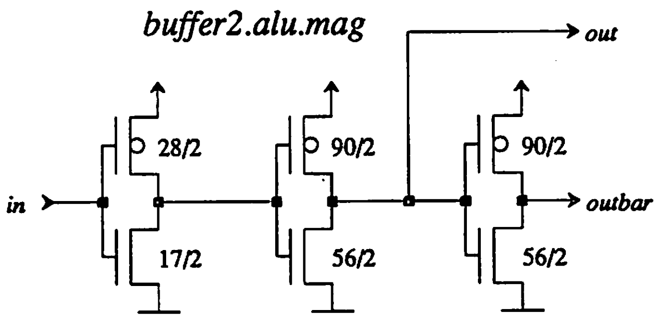


Figure C.27: Buffer cell used in ALU control slice.

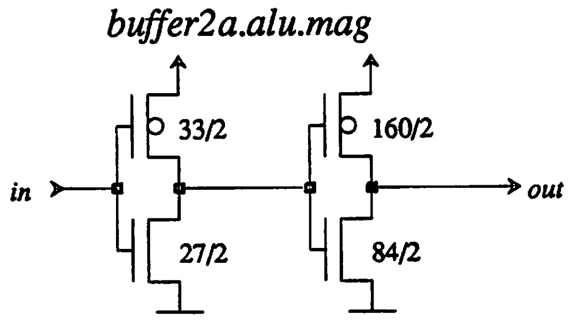


Figure C.28: Buffer cell used in ALU control slice.

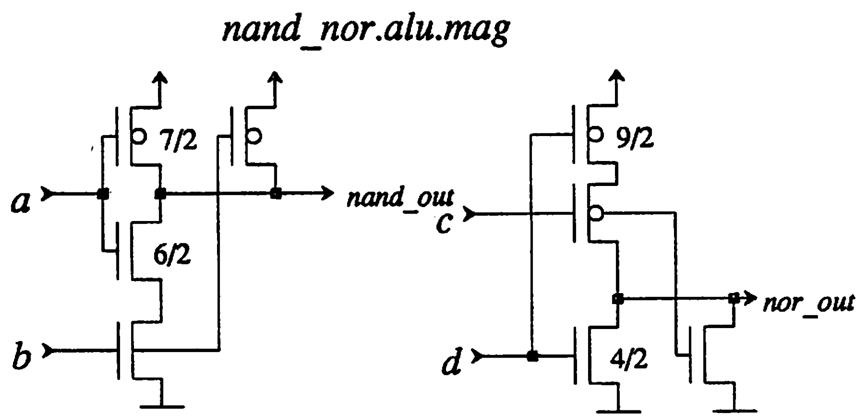


Figure C.29: NAND and NOR gate cell used in the ALU control slice.

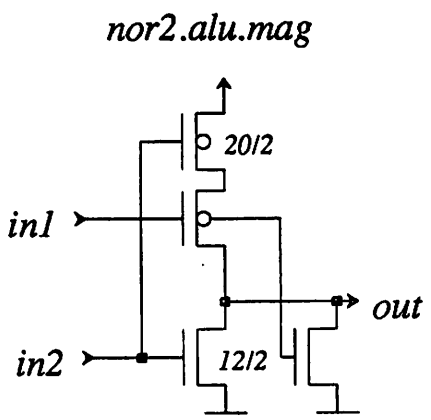


Figure C.30: NOR gate cell used in the ALU control slice.

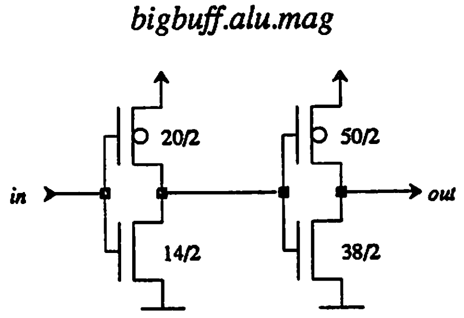


Figure C.31: Big buffer cell used in ALU control slice.

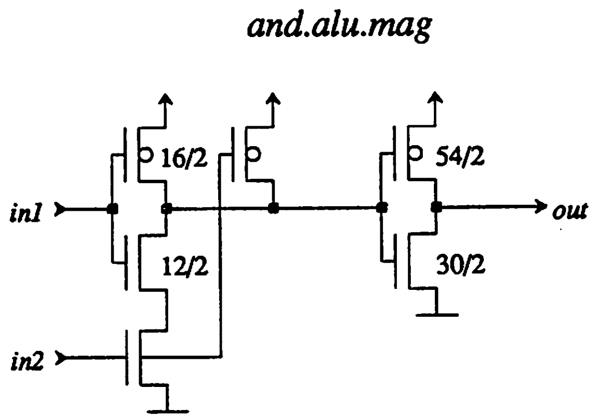


Figure C.32: AND gate used in ALU control slice.

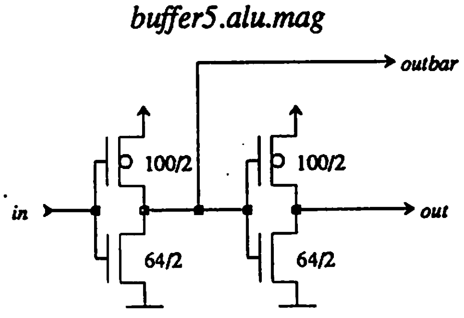


Figure C.33: Buffer cell used in ALU control slice.

2.3 LSB Slice

The ALU least significant bit slice contains a Ground bus that feeds all of the cells and a single inverter which drives the adder carry input high when the *inv* control signal is high. When the ALU is set to subtract two numbers, the A-input is only 1's complement inverted and the extra "one" at the carry input is required to implement a true 2's complement inversion. One side effect is that there is no discrimination between logical operations and addition when setting the carry input. Therefore the syntax for 1's complementing a number in the assembly code does not work if the ALU is using the adder. Instead, a full 2's complement inversion is performed. In other words, the instruction

ACC = ACC + ram1;

actually performs the operation

ACC = ACC - ram1;

For logical operations, the 1's complement operator works correctly.

3. Multiplier

This section contains a more detailed explanation of the multiplier timing and partitioning as well as schematics for all cells not shown in Chapter 6. A more detailed block diagram of the multiplier containing node names and MAGIC subcell names is given in Figure C.34. The shaded portion is all gates contained in the *recoder4.mult.mag* subcell of the control slice.

3.1 Bitslice

The individual bit slices contain the input registers, booth encoding circuitry, and carry save adders as diagrammed in Figure 6.12. Because the shift register for the Y-input shifts two bits per cycle, even and odd cells are required. These are shown in Figure C.35. The X-input register is a simple D flip-flop and its schematic is given in Figure C.36. The booth encoding DCVSL gate was shown in Figure 6.13. The carry save adder is implemented as two DCVSL gates as was done in the ALU full adder. Two inputs to the adder come from the sum and carry storage registers and the third input comes

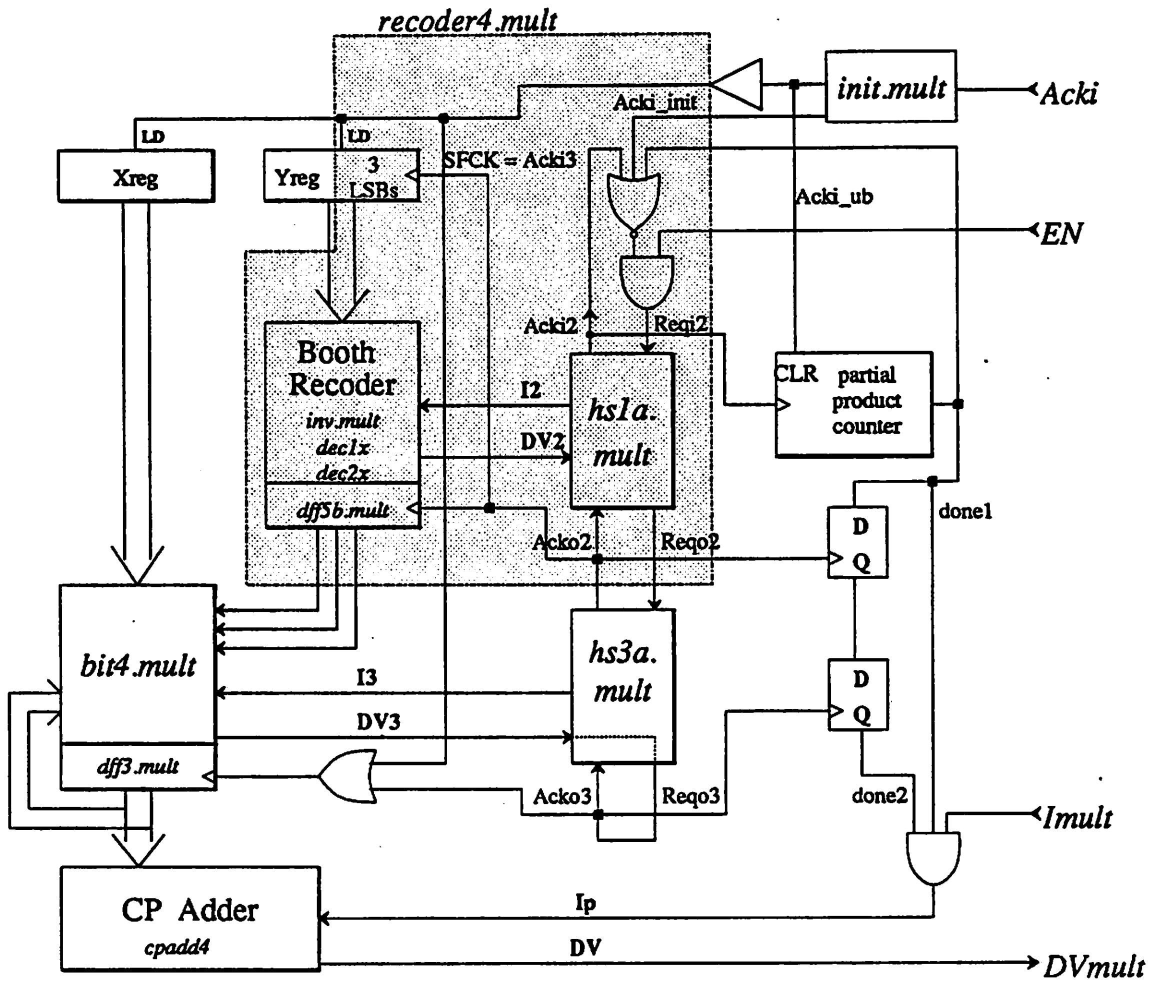


Figure C.34: Detailed block diagram of the iterative self-timed multiplier.

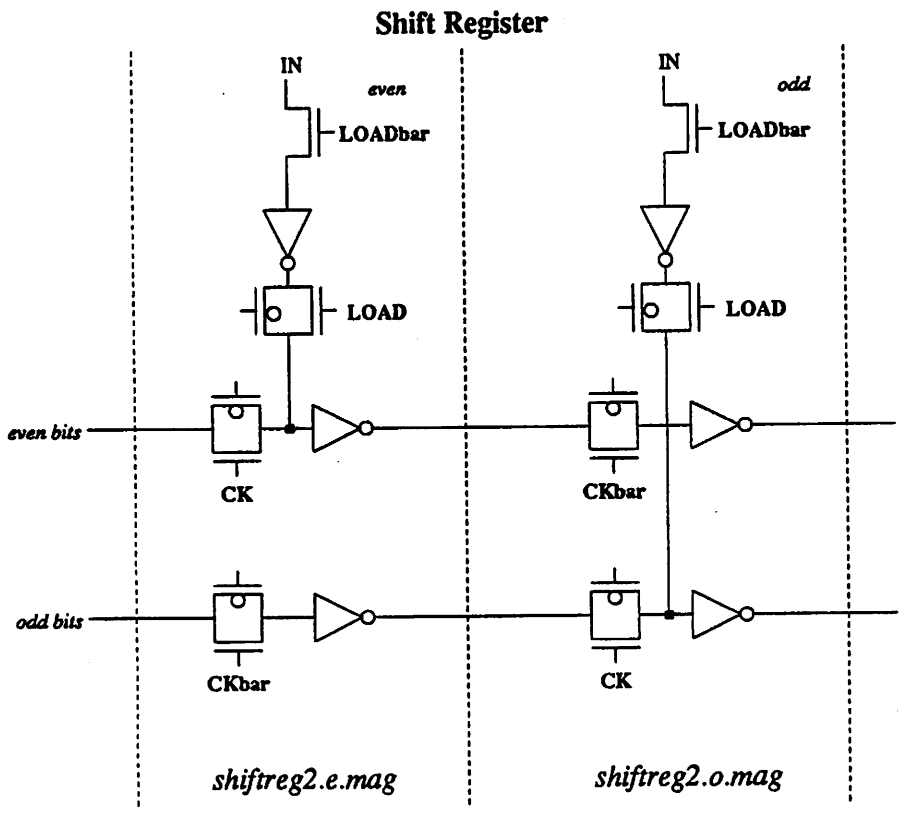


Figure C.35: Schematics for the shift register cells making up the multiplier Y-input register.

from the encoder gate. The schematics for the sum and carry DCVSL gates are given in Figures C.37 and C.38 respectively. Routing between the bitslices gives the two-bit shift required between partial product calculations. Thus, the input to the *sum* gate of bit_i actually comes from the sum storage register of bit_{i-2}. The storage registers for

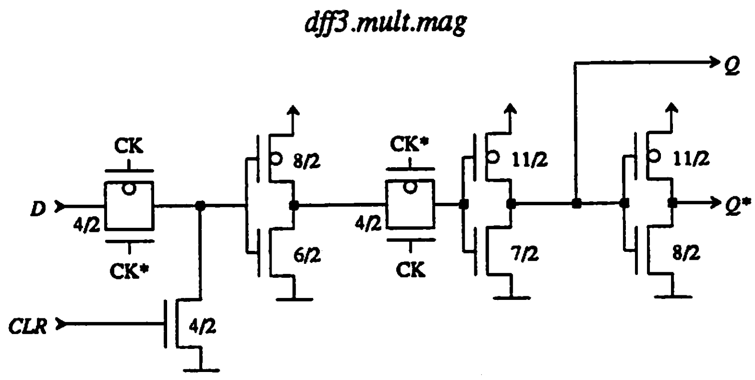


Figure C.36: D flip-flop used for the X-input register of the multiplier bit slices.

the carry save adder are implemented by the cells *dff8.mult.mag* and *dff9.mult.mag* for the *carry* and *sum* values respectively. The schematics for the two flip-flops are given in Figures C.39 and C.40.

The carry propagate adder that computes the final product in the multiplier is another DCVSL adder similar to the one used in the ALU. The adder is made from the cell *cpadd4.mag* which is four bits wide to accommodate the completion signal scheme which is a tree of gates on the sum outputs as shown in Figure C.41. The full adder cell *cpadd2.mag* is shown in Figure C.42. It has a completion signal output as well as the sum and carry outputs.

3.2 Control Slice

The control slice for the multiplier is rather complicated, containing the Booth recoder, handshake circuits, and the partial product counter. A block diagram of the complete control slice is shown with MAGIC subcell names in Figure C.43. The multiplier is a self-timed subsystem and the *Acki* signal which clocks the input registers, is used to synchronize the beginning of a new multiplication. The multiplier internal handshaking circuits must wait for the falling edge of the the *Acki* signal before proceeding with a sum3.mag

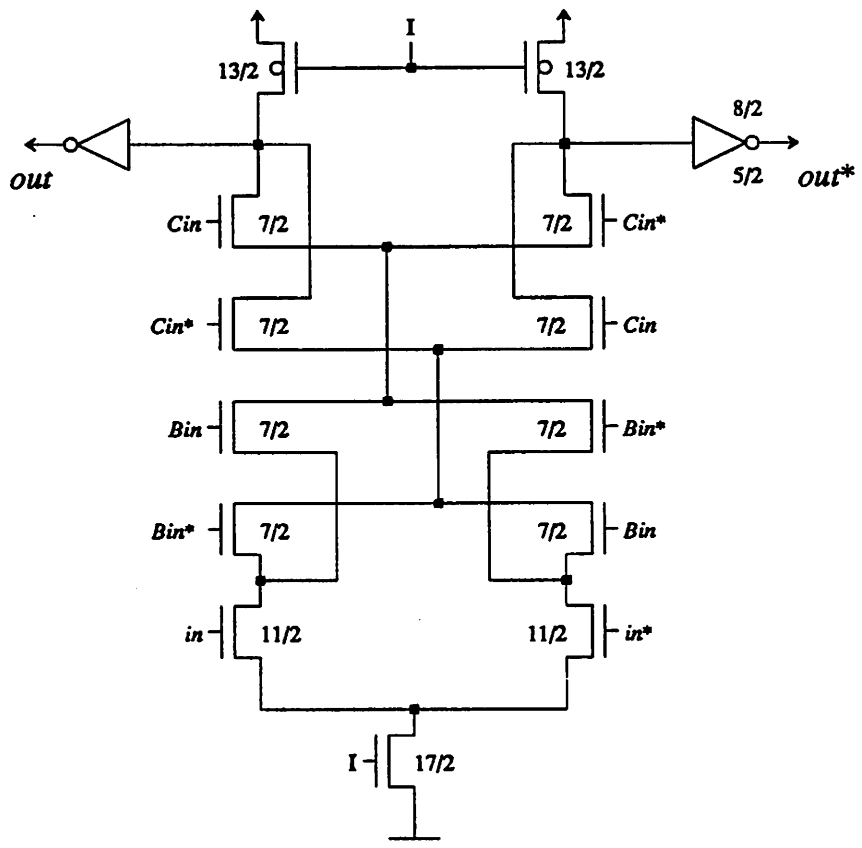


Figure C.37: Schematic for the DCVSL gate performing the *sum* operation of the carry save adder.

carry8.mag

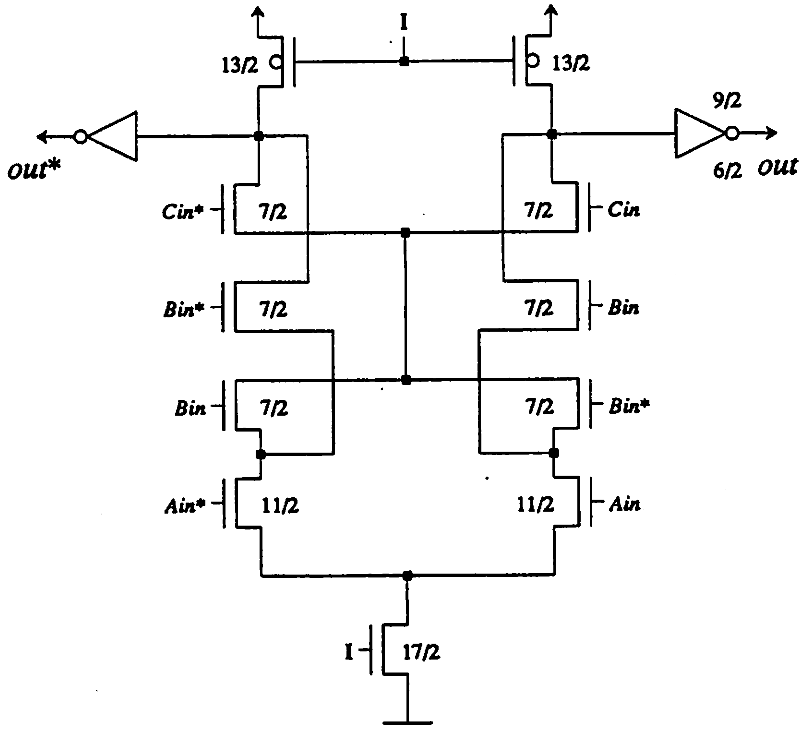


Figure C.38: Schematic for the DCVSL gate performing the *carry* operation of the carry save adder.

dff8.mult.mag

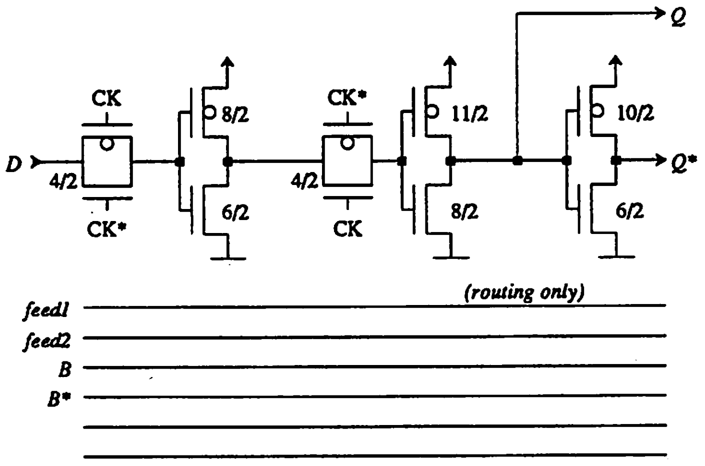


Figure C.39: Flip-flop used for storing the *carry* result of the carry save adder in the multiplier bitslice.

dff9.mult.mag

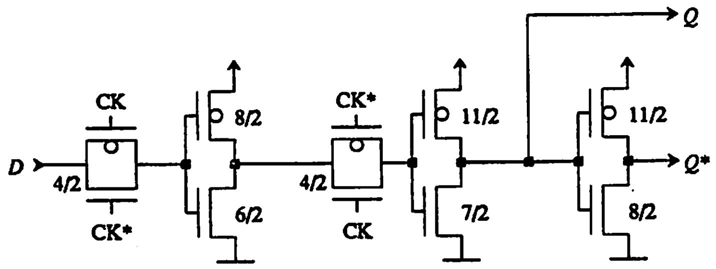


Figure C.40: Flip-flop used for storing the *sum* result of the carry save adder in the multiplier bitslice.

new multiplication. An additional signal denoted "EN" is present over a simple DCVSL macrocell. In the signal processor design, the multiplier is located in parallel with the barrel shifter within the datapath. Because the *Acki* signal may toggle during use of the shifter, and the *Imult* signal does not change until some time after the *Acki* signal, the EN input allows for disabling the multiplier. The master reset signal of the DSP chip is also fed to the multiplier to clear the handshake circuits. It is denoted INIT in the diagram. The recoder subcell of the control slice will be examined first since it contains the first stage of the internal multiplier pipeline. Dropping down one more level in the hierarchy, the *recoder4.mult.mag* cell block diagram is shown in Figure C.44. MAGIC subcell names are shown in the figure also. When the multiplier is running, the *Acki* signal of the first handshaking circuit (*hs1a*) is fed around to its *Reqi* input. The NOR gate in *3in_nor2.mult.mag* is used to stop the pipeline when the partial products are computed and to inhibit the pipeline from starting until the *Acki*1 signal falls. The schematic for the NOR gate cell is shown in Figure C.45. The EN signal function is made with a single AND gate in series with the *Reqi*2 signal as shown. The schematic for the AND gate is given in Figure C.46. For each partial product calculation, an appropriate Booth coefficient must be computed. The coefficient is a mapping of the three least significant bits of the Y-input shift register. These three bits are shifted by the *shiftreg2.lsb.mag* cell shown in Figure C.47. The three bits are fed to the recoder gates which were shown logically in Figure 6.11. The schematics for the three required gates are shown in Figure C.48, Figure C.49, and Figure C.50. The completion signal for the recoding stage is formed by an OR gate that is fed by the $2x$ and $2xbar$ signals. The schematic for the OR gate is

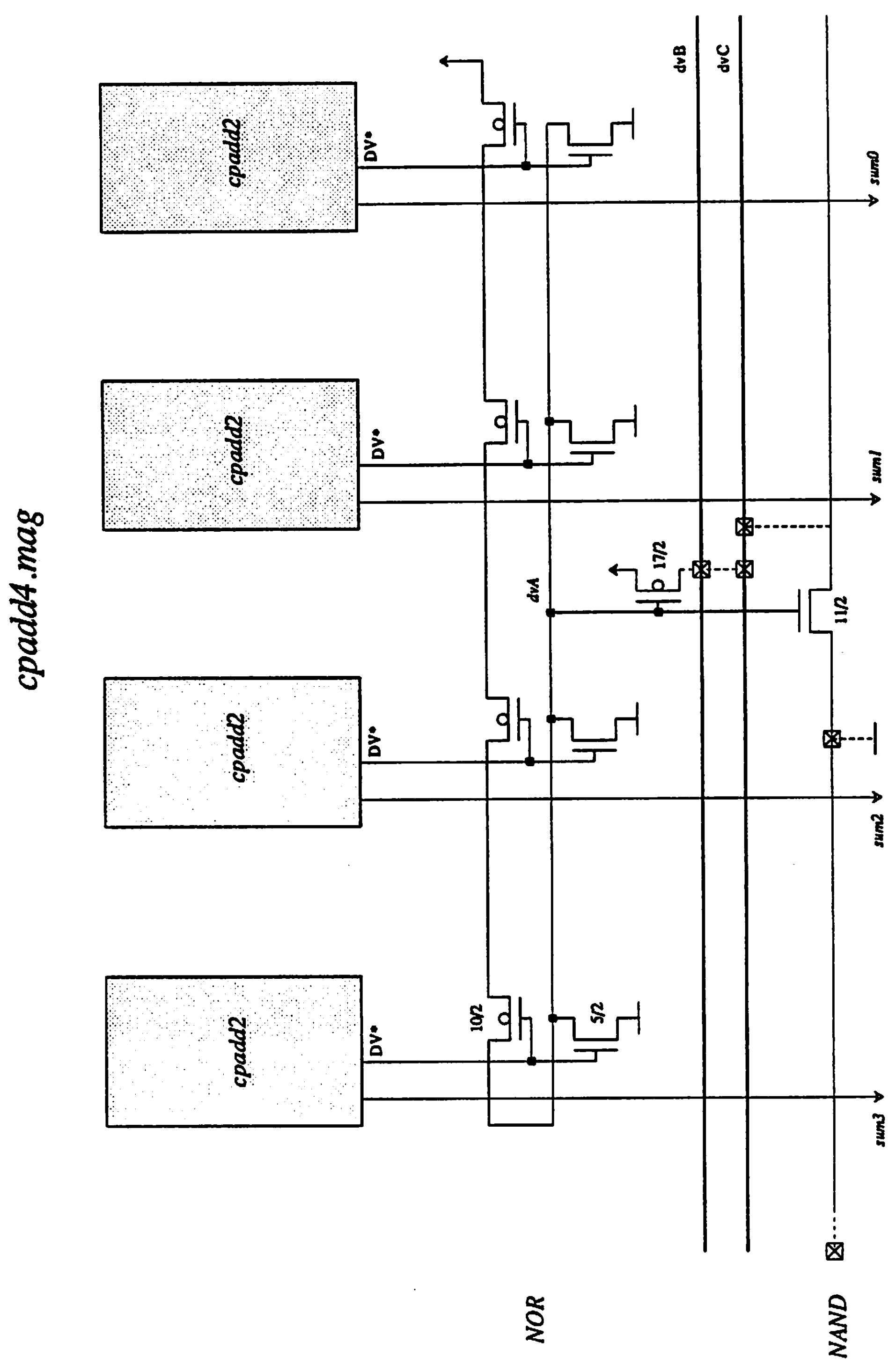


Figure C.41: Carry propagate adder cell used in the multiplier.

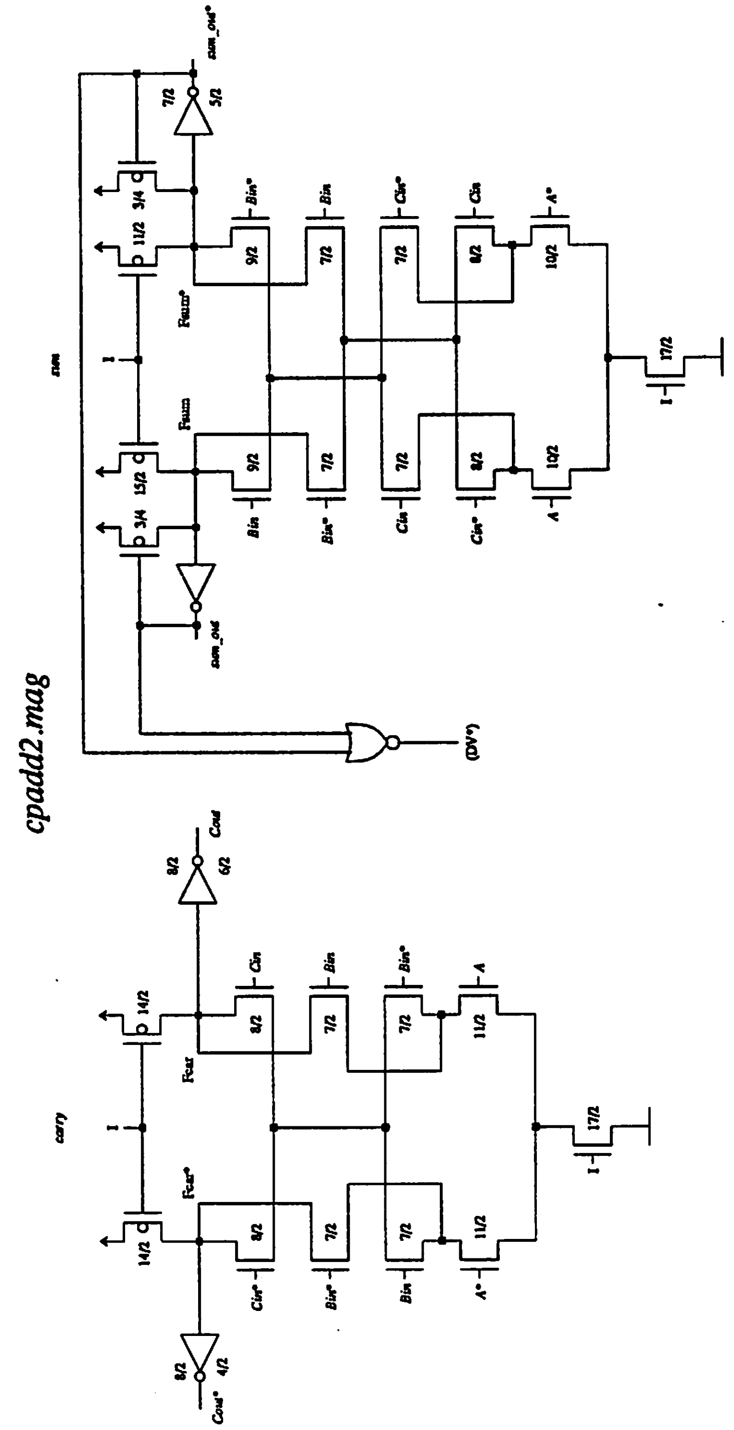


Figure C.42: Full adder subcell used in the carry propagate adder.

mult.ctl.dp1.mag

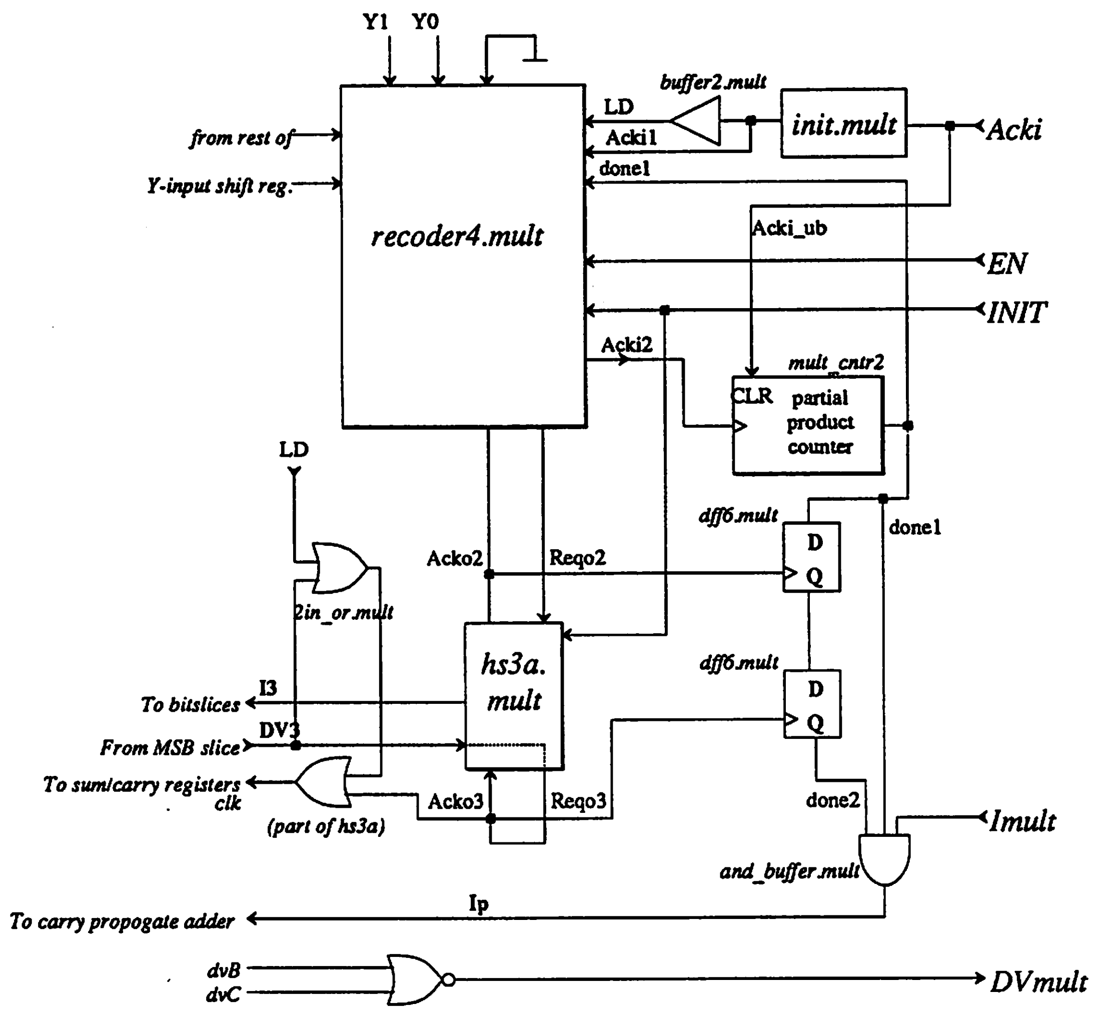


Figure C.43: Block diagram of the multiplier control slice.

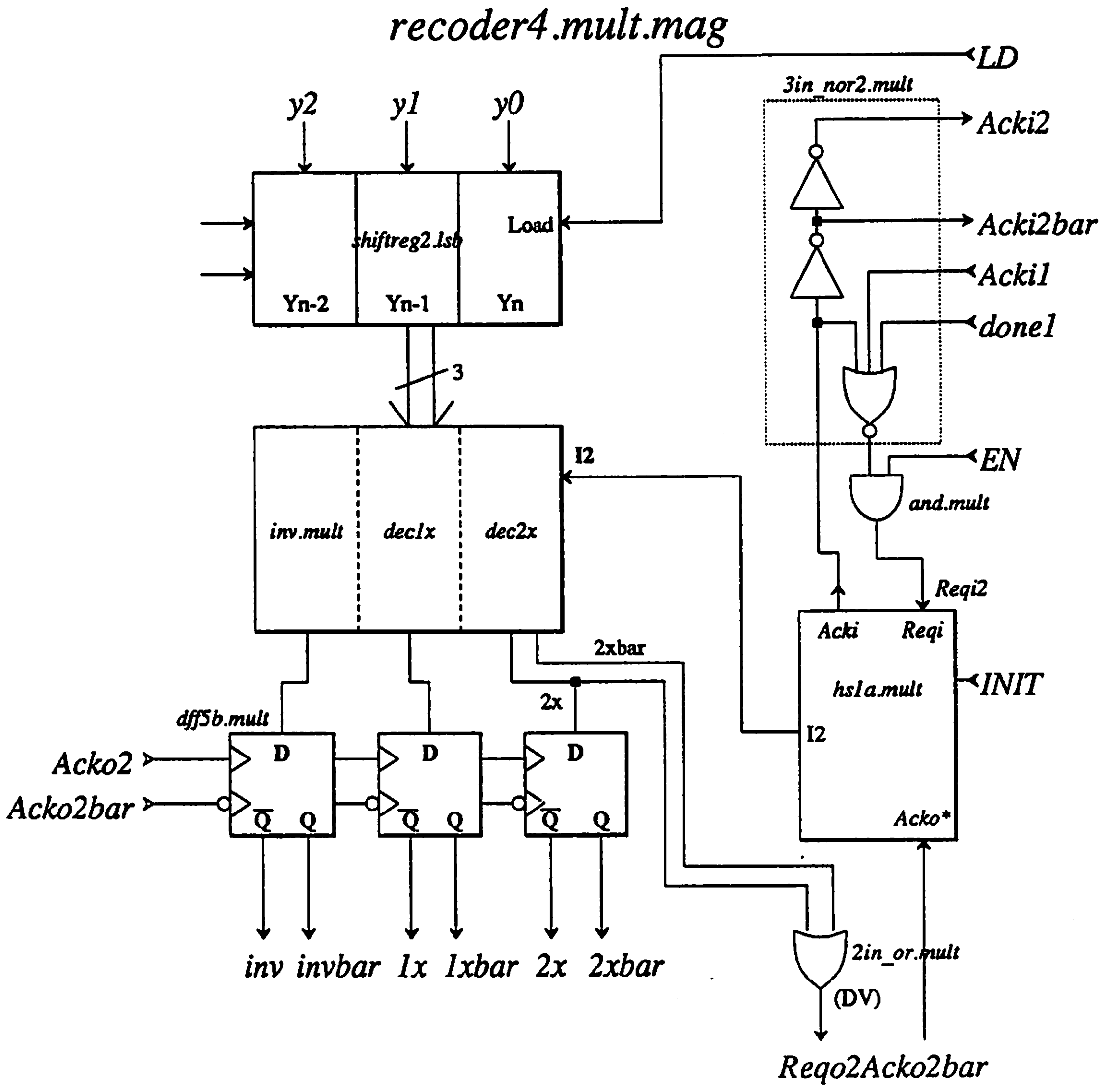


Figure C.44: Booth recoder and associated handshaking circuits contained in the multiplier control slice.

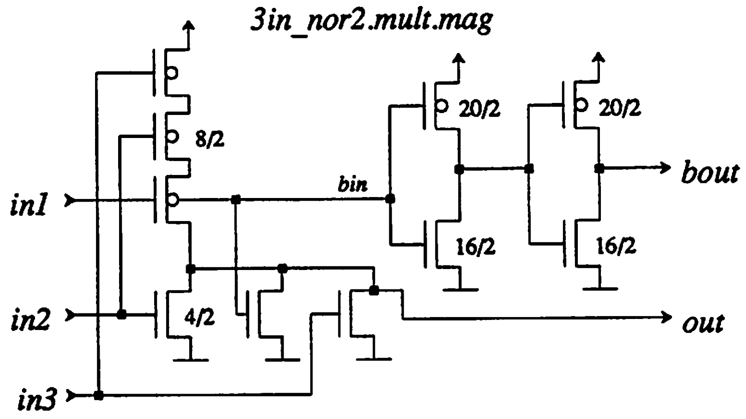


Figure C.45: 3-input NOR gate used in the Multiplier control slice.

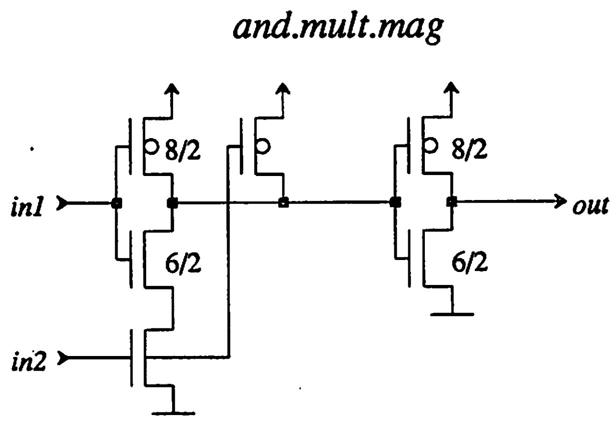


Figure C.46: 2-input AND gate used in the Multiplier control slice.

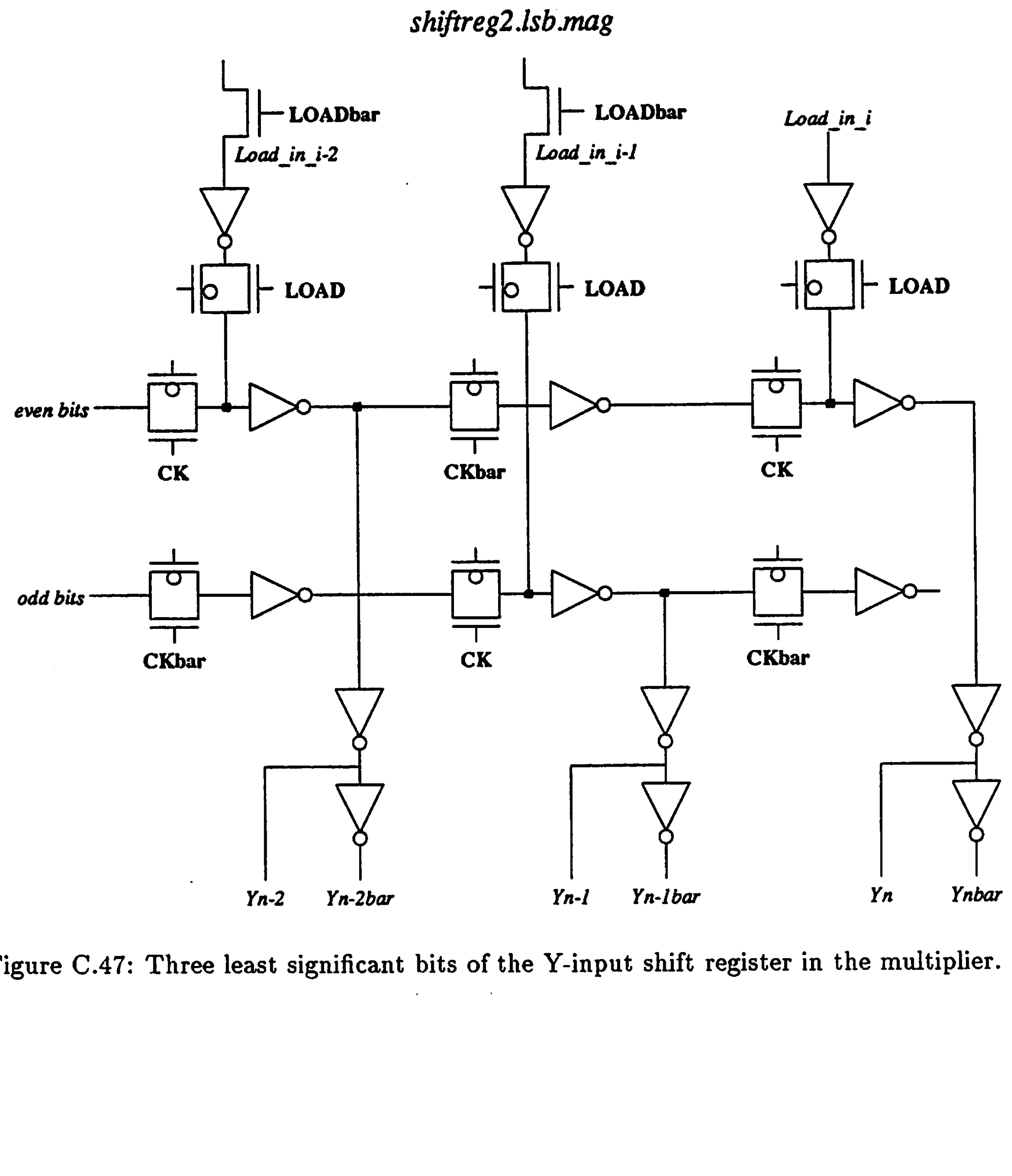


Figure C.47: Three least significant bits of the Y-input shift register in the multiplier.

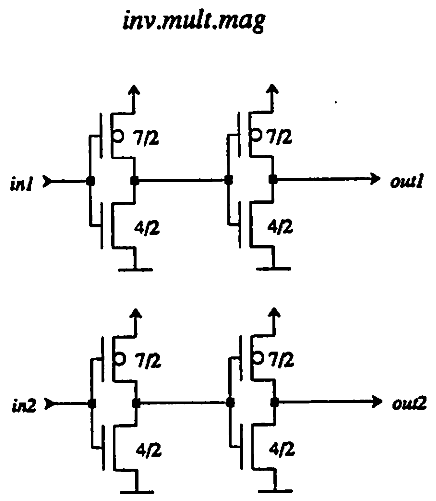


Figure C.48: Inverter cell used in the Booth recoder.

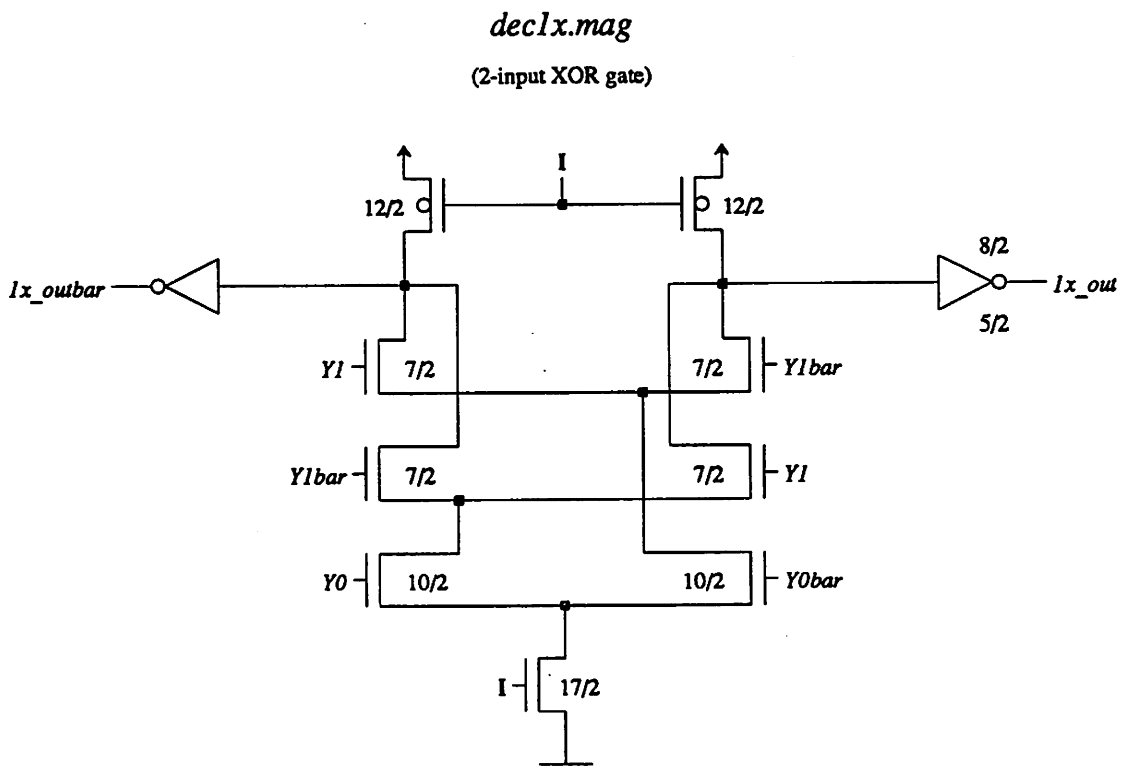


Figure C.49: DCVSL gate used in the Booth recoder for generating the "1x" signal of the Booth coefficient.

dec2x.mag

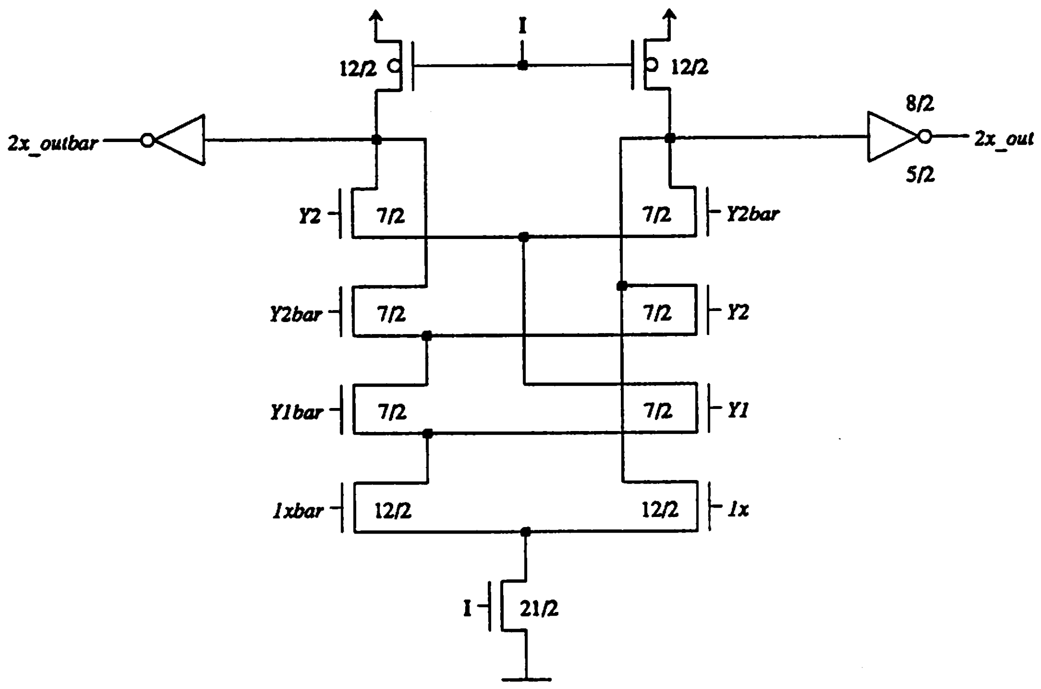


Figure C.50: DCVSL gate used in the Booth recoder for generating the "2x" signal of the Booth coefficient.

given in Figure C.51. The Booth coefficient is stored a register made up of *dff5b.mult* cells.

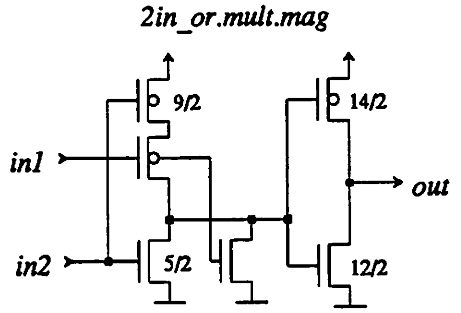


Figure C.51: 2-input NOR gate used in the Multiplier control slice.

The register has larger output devices for driving the bitslices. (The coefficient signals go to the bitslice Booth encoder (*enc3.mag* cell)) The schematic for the flip-flop is given in Figure C.52.

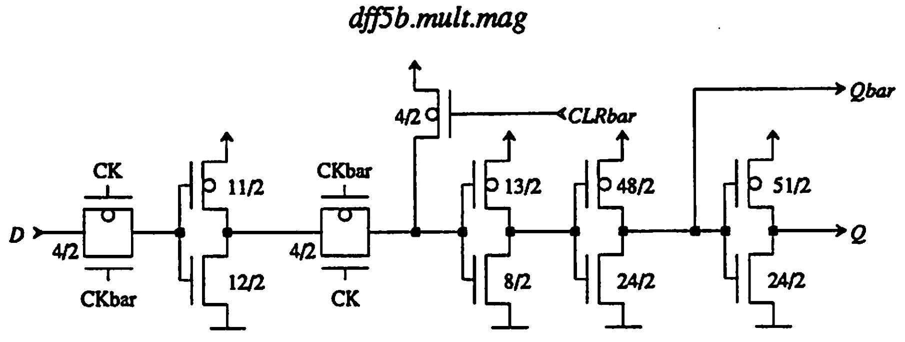


Figure C.52: Flip-flop cell used to store the Booth coefficient in the recoder stage.

The handshaking of the recoder stage is handled by the subcell *hs1a.mult.mag.* This is a simple 4-cycle handshake circuit made up of 2 clatches as shown in Figure C.53. The schematic for the clatch was given in Figure 5.11. Moving back up the hierarchy again from the recoder cell into the control slice, (refer to Figure C.43) the *Acki* signal is fed to a special cell that is used for initialization of the multiplier. The schematic for the *init.mult.mag* cell is shown in Figure C.54. The initialization circuit keeps *Acki*1 high between the time that the chip reset is high and the first acknowledge signal is sent to the multiplier. If this were not done, the multiplier would run once before being ever being called after the chip was reset. The *Acki* signal sent to multiplier clears the partial product

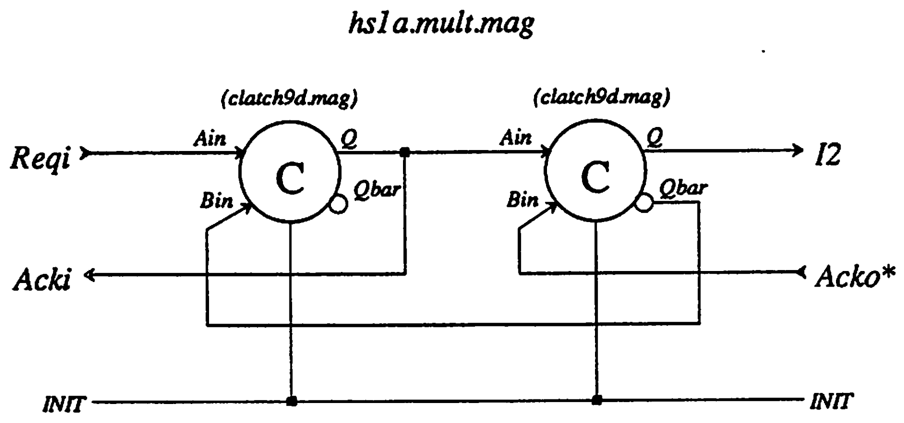


Figure C.53: Handshake circuit used in the recoder stage of the multiplier control slice.

counter and is buffered by *buffer2.mult.mag* before being sent to the input registers to load new data. The buffered signal, denoted LD in the drawings also is applied to a OR gate which causes the carry save adder storage registers to be clocked by the *Acki* signal. This loads them with zeros which is required before a new multiplication begins. The buffer and OR gate schematics are shown in Figures C.55 and C.51.

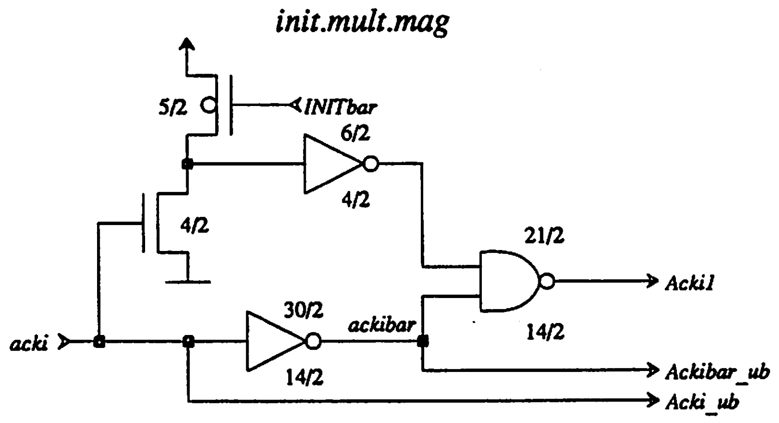


Figure C.54: Initialization circuit for *Acki* signal of multiplier.

The delay line for the "done" signal of the partial product counter is constructed from *dff6.mult.mag* cells, for which the shematic is shown in Figure C.56. The *Imult* signal is not applied to the carry propagate adder until the partial products are calculated. An AND gate is used for this function and it is shown in Figure C.57. The handshake cell for the carry-save stage of operation is *hs3a.mult.mag*. It contains a 4-cycle handshake

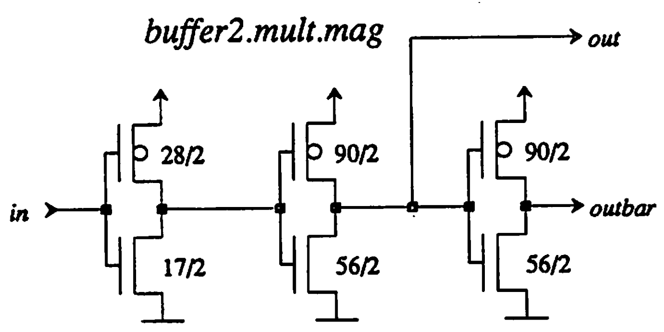


Figure C.55: Buffer circuit used in the Multiplier control slice.

circuit and an OR gate which drives the storage register clocks. The OR gate actually is redundant with the operation of the *2in_or.mult.mag*. A schematic for the *hs3a* cell is shown in Figure C.58. The schematic for the subcell *buffer3a.mult.mag* which was not given earlier is in Figure C.59.

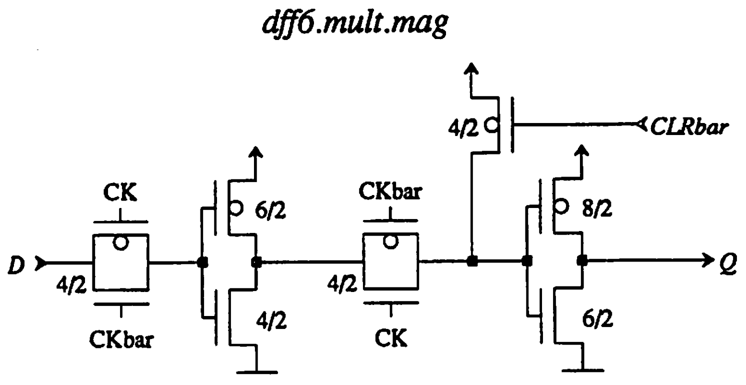


Figure C.56: Flip-flop used for the *done* signal delay line in the multiplier control slice.

3.3 MSB Slice

Because the Booth coefficient can have a value of 2, an extra bit on the MSB side of the multiplier is required. The MSB slice contains the carry-save circuitry for the extra bit. Additionally, it contains circuitry which generates the completion signal for the carry-save operation. Symmetry is exploited in the completion circuit. Since all the bits have the same carry-save circuitry and the carry-save addition operation is not data dependent, a single adder circuit is used to generate the completion signal for all bits. As

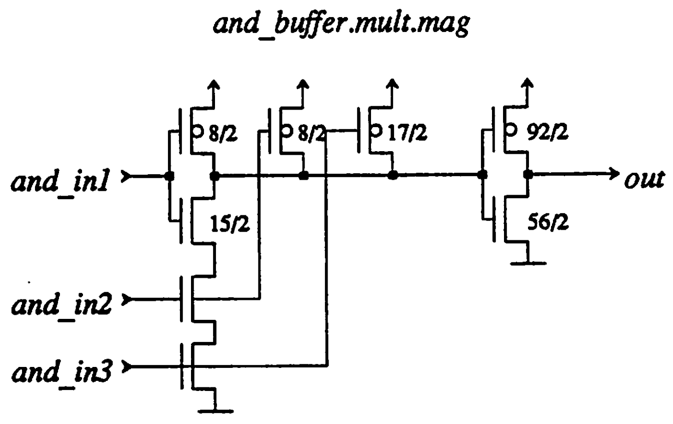


Figure C.57: AND gate and buffer used for the I signal of the carry propagate adder.

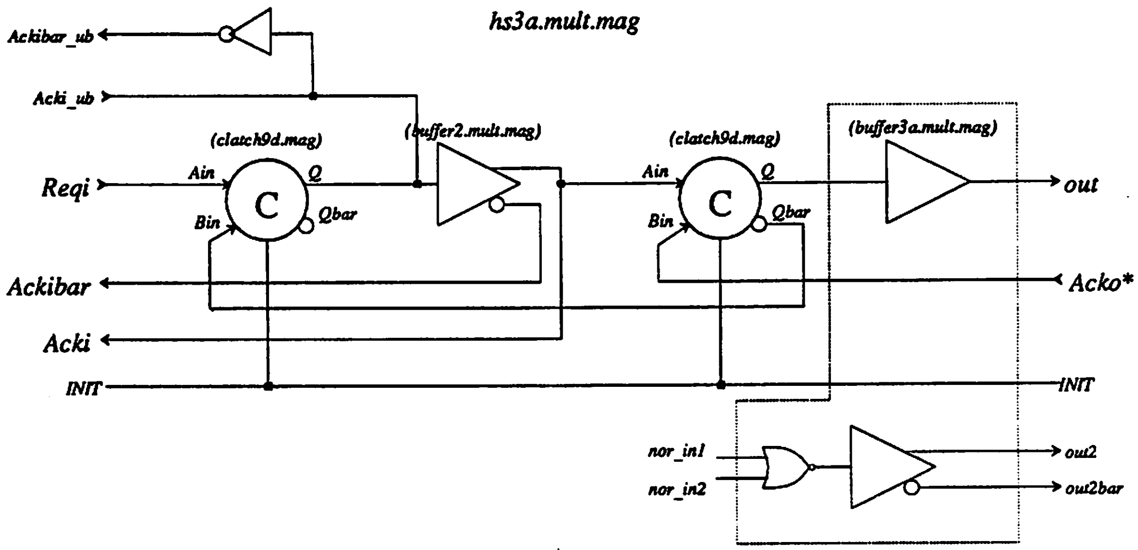


Figure C.58: Handshake circuit for the carry-save stage of the multiplier.

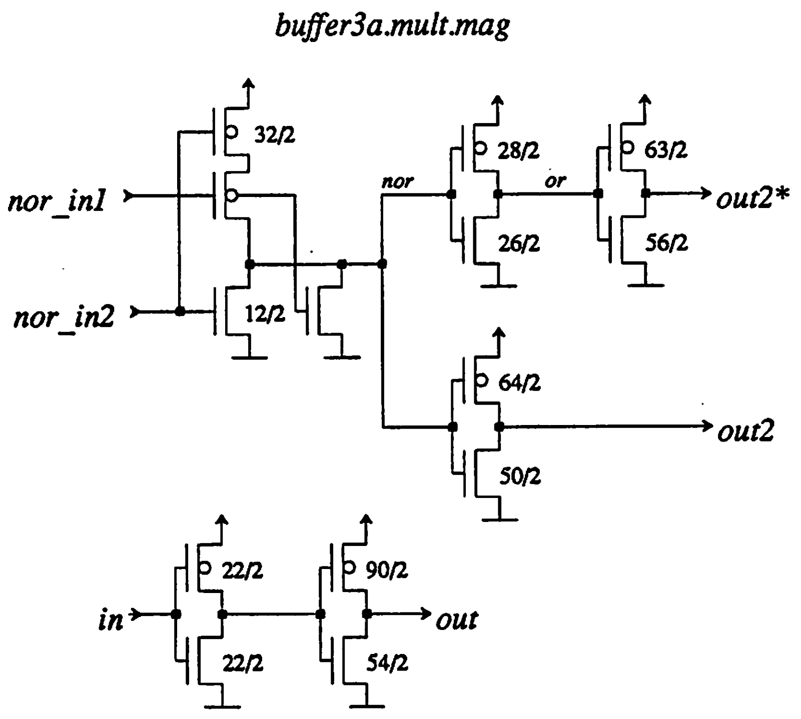


Figure C.59: Buffer inside of *hs3a.mult.*

shown in Figure C.60, the output of the Booth encoder, is fed to a "dummy" sum circuit, which generates the **DV3** signal.

4. Parameterized Cells

The LAGER system that was used for assembling the DSP chip allows for parameterized macrocell generation. The designer supplies a set of the lower level cells (called "leafcells") and a tiling routine that specifies how the cells should be placed together. The tiling can depend on parameters specified in the design files for the chip.

In the DSP design, there were several places that used the same type of macrocell. For example, the RAM feedback path uses three registers that are 16-bits wide. In the datapath, several MUXes of different wordlength are required. These cells were made using parameterized macrocell generation. The description of the circuits for the tilable cells is given in this section. The tiling routines are written in 'c'-language syntax using several library functions to do the tiling. The "sdl" file gives the names of the connectors on the cell boundary and the parameters. For a full description of these file formats, the

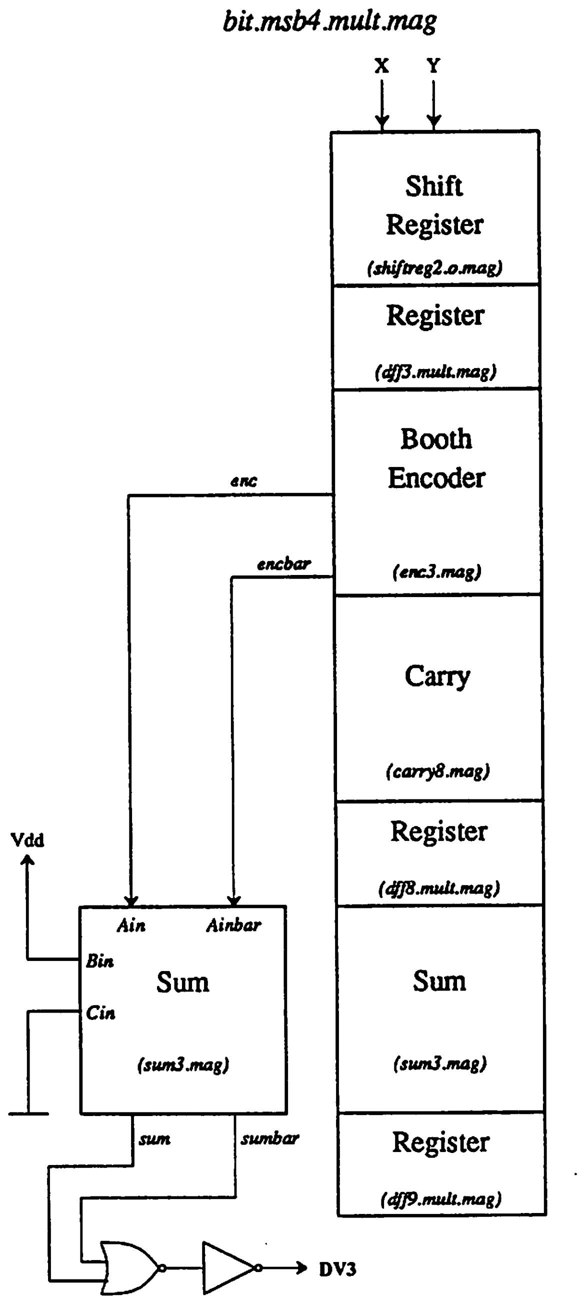


Figure C.60: MSB bitslice used in the multiplier.

reader is directed to the LAGER system documentation[74].

4.1 Dregister

The *dreg* cell is used in the instruction pipeline, RAM feedback path, and the I/O FIFO's. The *sdl* file for the cell is shown below:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Name    : dreg.sdl
;;; Purpose : D-Register w/clr
;;; Author  : Gordon Jacobs
;;; Date    : 5/30/88
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(parent-cell dreg)

(layout-generator TimLager)

(parameters width buffered)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; NETS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(net out (NETWIDTH width) ((parent out)))
(net in (NETWIDTH width) ((parent in)))
(net CK ((parent CK)))
(net clr ((parent clr)))

(net Vdd (NETTYPE SUPPLY) ((parent Vdd)))
(net GND (NETTYPE GROUND) ((parent GND)))

(end-sdl)
```

There are two parameters that are specified in the assembly of a register. The *width* is the number of flip-flops or the wordlength of the register. The *buffered* parameter is a binary valued parameter which specifies which leafcells to use. If it is non-zero, then a buffered cell is used that contains larger output drive. The cell *regcell.mag* is the flip-flop used in the normal register and its schematic is given in Figure C.61. The *reg.left.mag* clock buffer cell for the normal register is shown in Figure C.62. The cell *reg.right.mag* contains only an extension of the metal Vdd and GND lines to the cell boundary.

The buffered version of the register has the same floorplan, however it uses different leafcells. The schematics for the buffered version leafcells are shown in Figures C.63 and C.64.

The tiling routine for these cells in given below:

```
/*  dreg.c  */
/***********************************************************************/
/* TimLager routine for row of D flip-flops  (register) w/clr
/***********************************************************************/

#include "TimLager.h"

dreg()
```

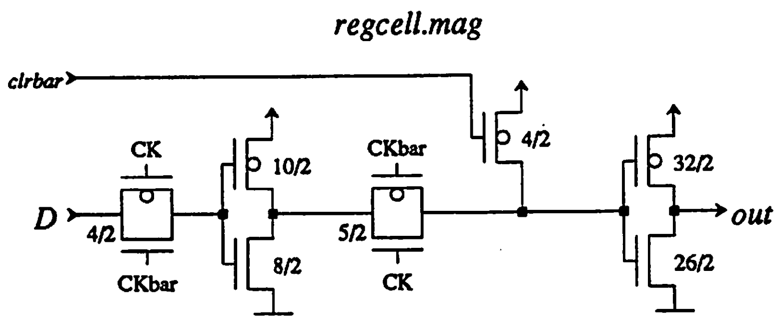


Figure C.61: Dynamic flip-flop cell used in parameterized register for DSP chip.

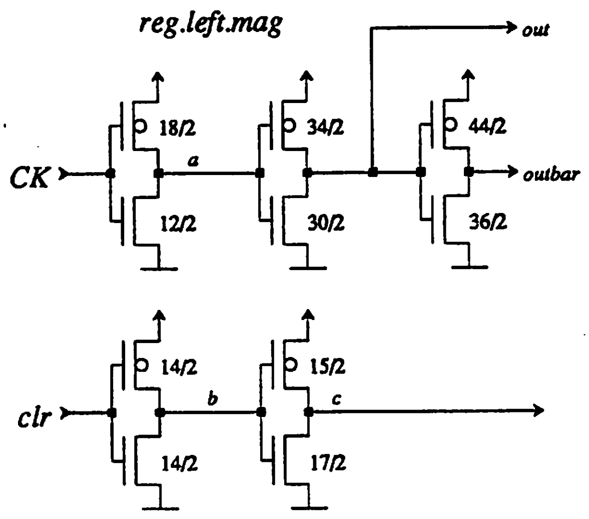


Figure C.62: Clock buffer cell used in parameterized register for DSP chip.

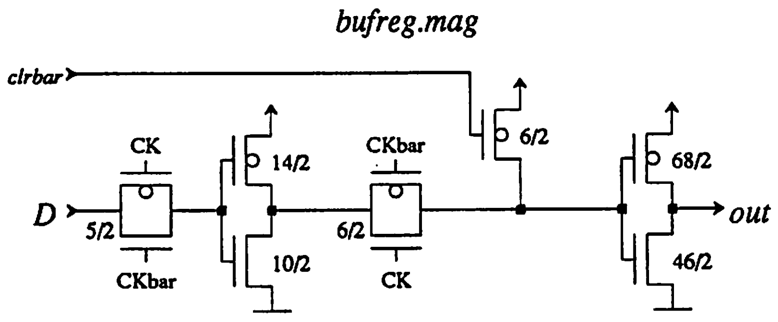


Figure C.63: Buffered output flip-flop cell used in parameterized register for DSP chip.

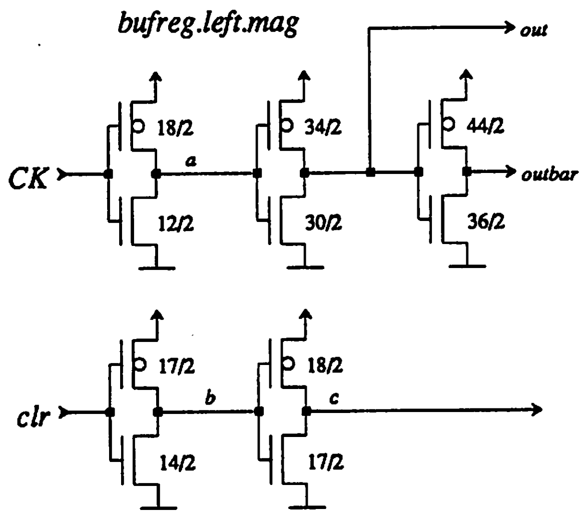


Figure C.64: Clock buffer cell for buffered version of parameterized register for DSP chip.

```
{
   int i,j;
   int word_length;
   int buffered;

   word_length = Getparam("width");
   buffered = Getparam("buffered");

   Open_newcell(Read("name"));

/*  Add blocks */

  if(!buffered) {
   Addright("reg.left",LEFT,TD,"CK",ALIAS,"CK",
            TD,"CKbar",ALIAS,"CKbar",
            TD,"clr",ALIAS,"clr",
            TD,"clrbar",ALIAS,"clrbar",
            END);

   Addright("regcell",TOP|BOTTOM,OFFSETY,-2,TD,"D",ALIAS,"in",INDEX,0,
                               TD,"out",ALIAS,"out",INDEX,0,
                               END);
   for(i=1; i < word_length; i++) {
       Addright("regcell",TOP|BOTTOM,TD,"D",ALIAS,"in",INDEX,i,
                               TD,"out",ALIAS,"out",INDEX,i,
                               END);
   }
   Addright("reg.right",RIGHT,OFFSETY,2,END);
  }
  else {  /* use bigger buffered reg cells... */
   Addright("bufreg.left",LEFT,TD,"CK",ALIAS,"CK",
            TD,"CKbar",ALIAS,"CKbar",
            TD,"clr",ALIAS,"clr",
            TD,"clrbar",ALIAS,"clrbar",
            END);

   Addright("bufreg",TOP|BOTTOM,OFFSETY,-3,TD,"D",ALIAS,"in",INDEX,0,
                               TD,"out",ALIAS,"out",INDEX,0,
                               END);
   for(i=1; i < word_length; i++) {
       Addright("bufreg",TOP|BOTTOM,TD,"D",ALIAS,"in",INDEX,i,
                                    TD,"out",ALIAS,"out",INDEX,i,
                                    END);
    }
    Addright("bufreg.right",RIGHT,OFFSETY,2,END);
   }

    Close_newcell();
 }
```

4.2 Dlatch

The *dlatch* cell is used in the controller. The *sdl* file for the cell is shown below:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Name    : dlatch.sdl
;;; Purpose : N-bit Transparent D-Latch w/clr
;;; Author  : Gordon Jacobs
;;; Date    : 5/31/88
;;;
;;;         : 2/22/89  Dynamic version of latch, requires clrbar, clkbar
;;;         : 2/27/89  Clock buffer added.  Only CK and clr signals
;;;                    required now as in static version but due to
;;;                    presence of clock buffer, signals can only
;;;                    come into one side of latch now.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(parent-cell dlatch)

(layout-generator TimLager)

(parameters width inverted)
;;;
;;; If "inverted" parameter is made non-zero, output is inverted
;;;     from input.  If "inverted" == 0, then there is no inversion
;;;     from input to output.  Non-inverted version of this latch
;;;     is slightly larger and slower since it has extra inverter on top.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; NETS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(net out (NETWIDTH width) ((parent out)))
(net in (NETWIDTH width) ((parent in)))
(net CK ((parent CK)))
(net clr ((parent clr)))

(net Vdd (NETTYPE SUPPLY) ((parent Vdd)))
(net GND (NETTYPE GROUND) ((parent GND)))

(end-sdl)
```

There are two parameters that are specified in the assembly of a latch. The *width* is the number of latch cells or the wordlength of the assembled latch. The *inverted* parameter is a binary valued parameter which specifies which leafcells to use. If it is non-zero, then only the single *latchcell.mag* is used and it inverts the signal. If *inverted* is made zero, then an additional cell *latch.inv.mag* is added so that the signal is does not see an inversion. The schematic for *latchcell.mag* is given in Figure C.65. The *reg.right.mag* clock buffer cell is shown in Figure C.66. When the inverter cell is added, the dlatch becomes taller, so different end cells must be used. For a non-inverting dlatch, the cell *latch.right2.mag* is used, but its schematic is identical to the *latch.right.mag* cell. The cells *latch.left.mag* and *latch.left2.mag* contain only an extension of the metal Vdd and GND lines to the cell boundary.

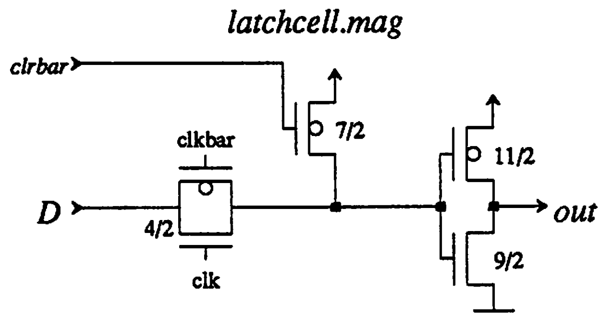


Figure C.65: Latch cell used in parameterized D-Latch for DSP chip.

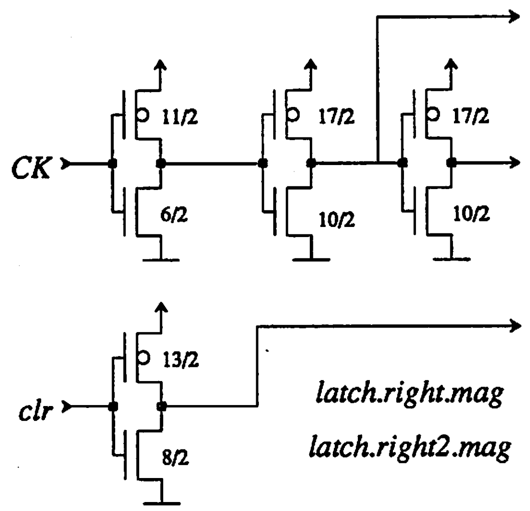


Figure C.66: Clock buffer cell used in parameterized D-Latch for DSP chip.

The tiling routine for these cells in given below:

```
/*  dlatch.c  */
/****************************************************************************/
/* TimLager routine for row of D latches  (register) w/clr
/****************************************************************************/

#include "TimLager.h"

dlatch()
{
    int i,j;
    int word_length;
    int inverted;
```

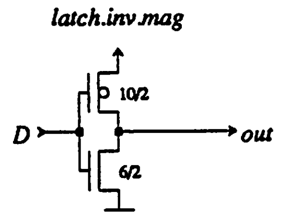


Figure C.67: Inverter cell used in parameterized D-Latch for DSP chip.

```
    word_length = Getparam("width");
    inverted = Getparam("inverted");

    Open_newcell(Read("name"));

/*  Add blocks */

    if(inverted)
        Addright("latch.left",LEFT,END);
    else
        Addright("latch.left2",LEFT,END);


    if(inverted)
        Addright("latchcell",TOP|BOTTOM,OVERLAP,
                    TD,"D",ALIAS,"in",INDEX,0,
                    TD,"out",ALIAS,"out",INDEX,0, END);
    else
        Addright("latchcell",BOTTOM,OVERLAP,
                    TD,"out",ALIAS,"out",INDEX,0, END);

    if(inverted)
        for(i=1; i < word_length; i++)
            Addright("latchcell",TOP|BOTTOM,TD,"D",ALIAS,"in",INDEX,i,
                    TD,"out",ALIAS,"out",INDEX,i, END);
    else
        for(i=1; i < word_length; i++) {
            Addright("latchcell",BOTTOM,TD,"out",ALIAS,"out",INDEX,i, END);
    }
    if(inverted)
        Addright("latch.right",RIGHT,OVERLAP,
                TD,"clk",ALIAS,"CK",
                TD,"clr",ALIAS,"clr", END);
    else
        Addright("latch.right2",RIGHT,OVERLAP,
                TD,"clk",ALIAS,"CK",
                TD,"clr",ALIAS,"clr", END);
    if(!inverted) {
        Addup("latch.inv",TOP,OFFSETX,9,OFFSETY,-38,
                TD,"D",ALIAS,"in",INDEX,0, END);
        for(i=1;i < word_length; i++)
            Addright("latch.inv",TOP,TD,"D",ALIAS,"in",INDEX,i, END);
    }

    Close_newcell();
}
```

4.3 2inMUX

The *mux* cell is used in the datapath at the RAM write input port, the Y-input to the multiplier, and the ALU input. The *sdl* file for the cell is shown below:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Name    : mux.sdl
;;; Purpose : N-bit wide 2-Input MUX with or without register on Control Input
;;; Author  : Gordon Jacobs
;;; Date    : 5/30/88
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(parent-cell mux)

(layout-generator TimLager)

(parameters width clocked)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; NETS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(net Ain (NETWIDTH width) ((parent Ain)))
(net Bin (NETWIDTH width) ((parent Bin)))
(net out (NETWIDTH width) ((parent out)))
;;; electrically the same as out
(net outb (NETWIDTH width) ((parent outb)))
(net CTL ((parent CTL)))
(net CK ((parent CK)))
(net clr ((parent clr)))

(net Vdd (NETTYPE SUPPLY) ((parent Vdd)))
(net GND (NETTYPE GROUND) ((parent GND)))

(end-sdl)
```

There are two parameters that are specified in the assembly of a latch. The *width* is the number of MUX cells or the wordlength of the assembled MUX. The schematic for *muxcell.mag* is given in Figure C.68. The *clocked* parameter is a binary valued parameter which specifies which leafcells to use. If it is zero, then the control input to the assembled MUX is applied directly to the MUX cells through the buffer circuit *mux.left2.mag* shown in Figure C.70. If *clocked* is made non-zero, then the cell *mux.left1.mag* is used instead and it contains a flip-flop which can store the control input. The schematic is shown in Figure C.69. The cells *mux.top2.mag* and *mux.bot2.mag* contain only an extension of the metal lines to the cell boundary. The bottom cell has two contacts for the (same) output of the MUX which were required to work around a bug in the routing tool.

The tiling routine for the 2-Input MUX is given below:

```
/*  mux.c  */
/************************************************************************/
/* TimLager routine for row of 2-Input muxes
/************************************************************************/

#include "TimLager.h"
```

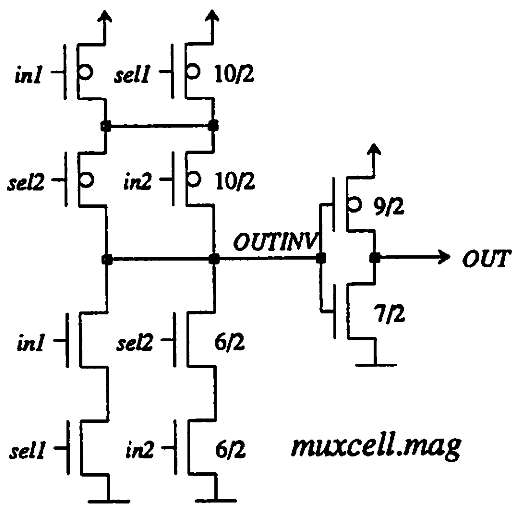


Figure C.68: 2-Input MUX cell used in parameterized MUX for DSP chip.

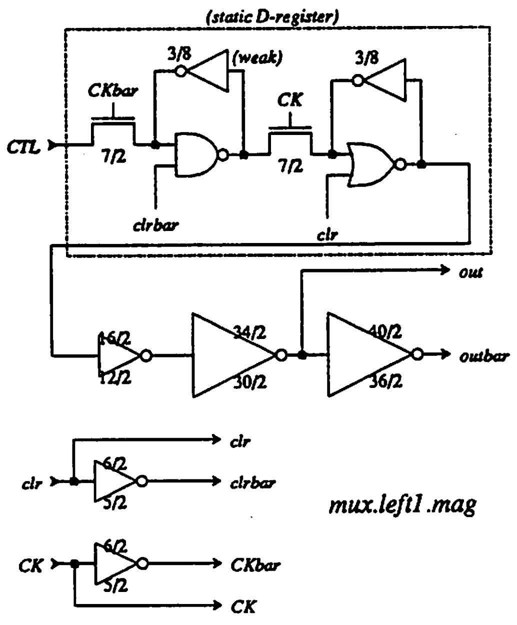


Figure C.69: Control input buffer for the parameterized MUX.

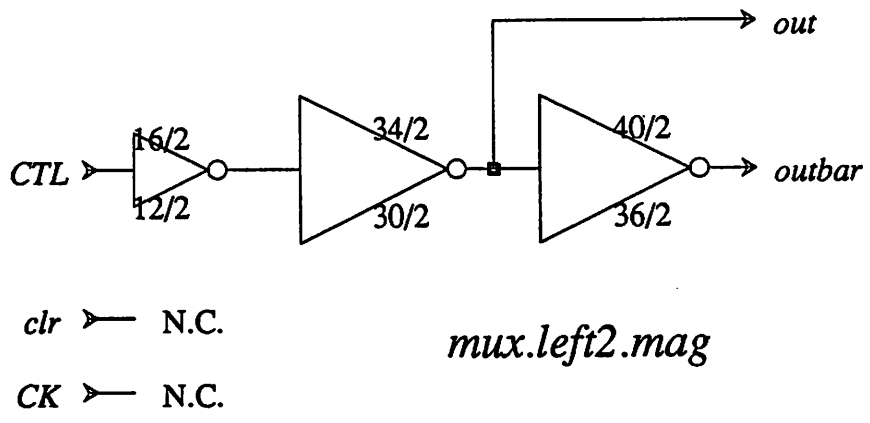


Figure C.70: Second control input buffer containing a flip-flop that is used in the parameterized MUX.

```
mux()
{
    int i,j;
    int word_length;
    int clocked;

    word_length = Getparam("width");
    clocked = Getparam("clocked");

    Open_newcell(Read("name"));

/*  Add blocks */

    if(clocked)
        Addright("mux.left1",LEFT,TD,"CK",ALIAS,"CK",
                TD,"CTL",ALIAS,"CTL",TD,"clr",ALIAS,"clr",END);
    else
        Addright("mux.left2",LEFT,TD,"CK",ALIAS,"CK",
                TD,"CTL",ALIAS,"CTL",TD,"clr",ALIAS,"clr",END);

    for(i=0; i < word_length; i++) {
        Addright("mux.bot2",BOTTOM,OVERLAP,TD,"out",ALIAS,"out",INDEX,i,
                                        TD,"outb",ALIAS,"outb",INDEX,i,END);
    }
    if(clocked)
        Addup("muxcell",NONE,OVERLAP,R270,OFFSETX,217,OFFSETY,-78,END);
    else
        Addup("muxcell",NONE,OVERLAP,R270,OFFSETX,90,OFFSETY,-78,END);

    for(i=1; i < word_length; i++) {
        Addright("muxcell",NONE,R270,END);
    }
    if(clocked)
        Addup("mux.top2",TOP,OVERLAP,OFFSETX,217,OFFSETY,-9,
            TD,"ain",ALIAS,"Ain",INDEX,0,
            TD,"bin",ALIAS,"Bin",INDEX,0,END);
    else
        Addup("mux.top2",TOP,OVERLAP,OFFSETX,90,OFFSETY,-9,
            TD,"ain",ALIAS,"Ain",INDEX,0,
            TD,"bin",ALIAS,"Bin",INDEX,0,END);
    for(i=1; i < word_length; i++) {
        Addright("mux.top2",TOP,OVERLAP,
            TD,"ain",ALIAS,"Ain",INDEX,i,
            TD,"bin",ALIAS,"Bin",INDEX,i,END);
    }

    Close_newcell();
}
```

4.4 LPC

The *lpc* cell is used in the controller and acts as a program counter, generating the address for reading the Program ROM. The counter was adapted from the LAGER cell library [74]. The *sdl* file for the cell is shown below:

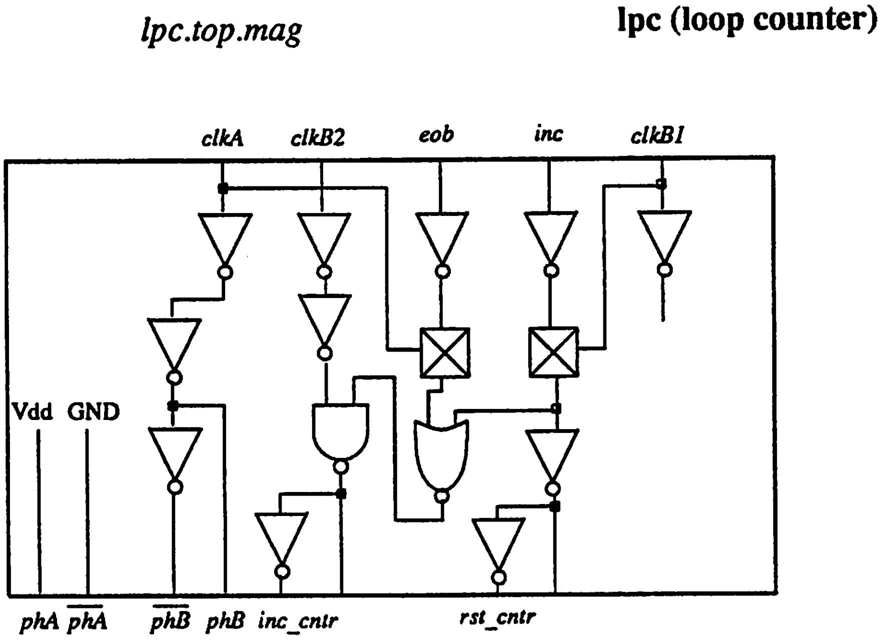
```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                          lpc.sdl                                      ;
;                   (was counter_p.sdl)                                 ;
;       TimLager module generation file for a loadable program counter  ;
;                                                                       ;
;                        Gautam Doshi, 11-12-87                         ;
;       Modified by Gordon Jacobs, 6/2/88                               ;
;       Counter Top cell changed (added buffers) and some signal        ;
;       names changed.  Otherwise, cells are same as in LagerIV/cellib  ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(parent-cell    lpc)
(parameters     width)
(layout-generator TimLager)


; LOADin is the input bus used to load the counter on reset
(net LOADin  (NETWIDTH width)  ((parent load_in)))

; "..L" and "..R" represent the side of the counter
; COUNT is the output of the counter
(net COUNT_L  (NETWIDTH width)  ((parent count_l)))
(net COUNT_L_BAR  (NETWIDTH width)  ((parent count_l_inv)))
(net COUNT_R  (NETWIDTH width)  ((parent count_r)))
(net COUNT_R_BAR  (NETWIDTH width)  ((parent count_r_inv)))

; CLOCKA and CLOCKB are the two nonoverlapping clocks
(net CLOCKA  ((parent clkbar)))
(net CLOCKB1 ((parent clkB1)))
(net CLOCKB2 ((parent clkB2)))

; RESET is active high
; removed...(net EOB  ((parent eob)))

; CNT is the counter increment signal (active high)
(net LOADbar  ((parent loadbar)))

(net Vdd (NETTYPE SUPPLY) ((parent Vdd)))
(net GND (NETTYPE GROUND) ((parent GND)))

(terminal Vdd (TERMTYPE SUPPLY))
(terminal GND (TERMTYPE GROUND))

(end-sdl)
```

The only parameter for the counter is the wordlength of the output. This determines how many of the actual counter cells are placed in the layout. The cells *lpc_Xo.mag* and *lpc_Xe.mag* contain the counter circuitry which is made from a half adder and a latch.

The two types are used alternately in the layout, the suffixes denoting "even" and "odd". The *lpc_top.mag* cell contains clock buffering and several gates which control counting and loading. The cells are shown in Figure C.71. Since the cell was adapted from a LAGER cell, the names are a little confusing. The way the counter is hooked up in the DSP is shown in Figure C.72. The *lpc_bot.mag* cell contains routing only.

The tiling routine for the program counter is shown below.

```
/***************************************************************************/
/*      TimLager module generation file for a programmable counter       */
/*                                                                         */
/*                       Khalid Azim, 6-10-87                              */
/*                       Gautam Doshi, 11-12-87                            */
/***************************************************************************/

#include "TimLager.h"
        int width,i,j,k,l;

lpc() {

        width = Getparam("width"); /* the bit width of the counter */

        /* On reset: count is loaded from external input ("load_in")*/

        Open_newcell(Read("name"));

        Addup("lpc_bot",NONE,OFFSETX,28,END);

        for (i=0; i<width; i++) {
              if (i%2)
                    Addup("lpc_Xo",LEFT|RIGHT,
                    TD,"in",ALIAS,"load_in",INDEX,i,
                    TD,"sum_l",ALIAS,"count_l",INDEX,i,
                    TD,"sum_l_inv",ALIAS,"count_l_inv",INDEX,i,
                    TD,"sum_r",ALIAS,"count_r",INDEX,i,
                    TD,"sum_r_inv",ALIAS,"count_r_inv",INDEX,i,END);
              else
                    Addup("lpc_Xe",LEFT|RIGHT,
                    TD,"in",ALIAS,"load_in",INDEX,i,
                    TD,"sum_l",ALIAS,"count_l",INDEX,i,
                    TD,"sum_l_inv",ALIAS,"count_l_inv",INDEX,i,
                    TD,"sum_r",ALIAS,"count_r",INDEX,i,
                    TD,"sum_r_inv",ALIAS,"count_r_inv",INDEX,i,END);
        }

        Addup("lpc_top",TOP|LEFT|RIGHT,
        /*      TD,"cout",ALIAS,"cout",
                TD,"phA",ALIAS,"phA",
                TD,"phAinv",ALIAS,"phAinv",
                TD,"phB", ALIAS,"phB",
                TD,"phBinv",ALIAS,"phBinv",
                TD,"rst",ALIAS,"rst",
                TD,"rstinv",ALIAS,"rstinv",
                TD,"cnt",ALIAS,"cnt",
                TD,"cntinv",ALIAS,"cntinv"
        */
                /* TD,"eob",ALIAS,"eob", removed */
                TD,"incr",ALIAS,"loadbar",
                TD,"clkB1",ALIAS,"clkB1",
                TD,"clkB2",ALIAS,"clkB2",
                TD,"clkA",ALIAS,"clkbar",
                END);

        Close_newcell();
}
```



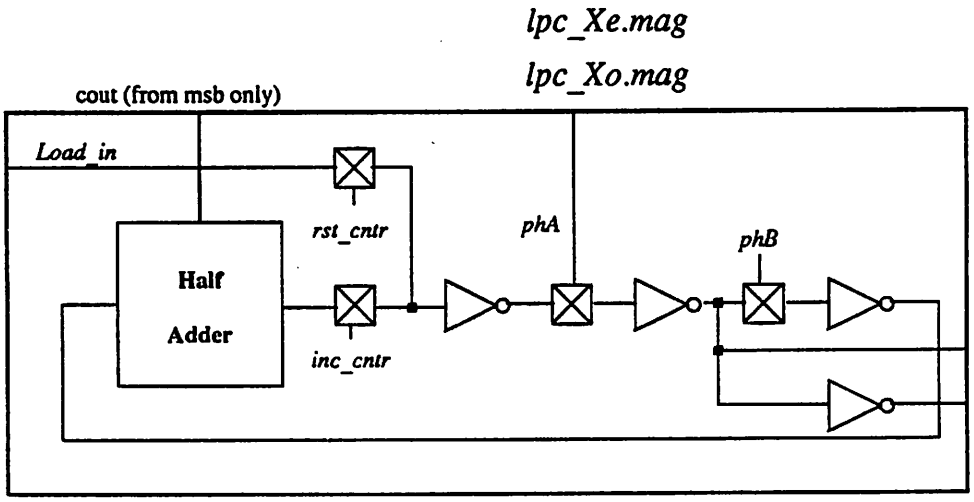
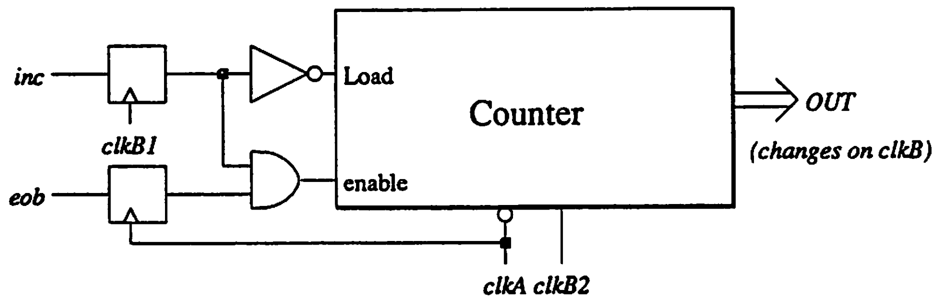


Figure C.71: Schematics for cells making up the counter in the DSP controller.

Connection of lpc.mag in DSP



- *Connect clkB1 and clkB2 to clk*
- *Connect clkA to clkbar*
- *Connect eob to Vdd (always enable)*
- *Connect inc to loadbar*
  (Loads counter on $\overline{clkB}$.inc)

Figure C.72: The connection of signals to the counter.

5. Handshake Std. Cells

The handshake circuits for the DSP shown in Chapter 7 were all assembled from a set of gates resembling standard cells. The gates have the same height and power supply bus locations so that they can be placed in rows and interconnected with routing channels. The layout in the DSP was done manually due to lack of a tool that recognized these particular cells. The schematics for the cells are shown in this section. The names match those subcell names shown in the handshake circuit figures in Chapter 7.

5.1 Simple Gates

The schematics for a 2-input NAND gate and a 2-input NOR gate used in the handshake circuits are shown in Figures C.73 and C.74. A three input NOR gate is shown in Figure C.77.

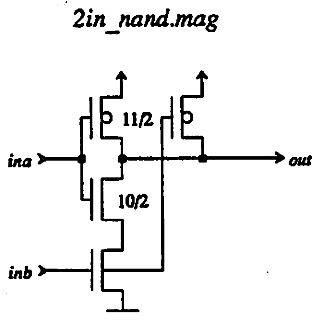


Figure C.73: Schematic for 2-input NAND gate used in the handshake circuits.

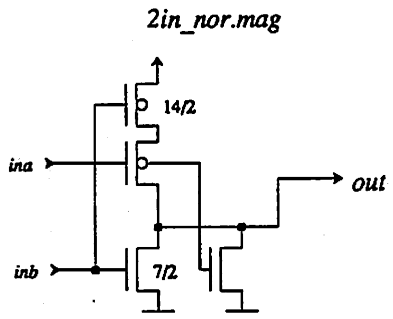


Figure C.74: Schematic for 2-input NOR gate used in the handshake circuits.

A 3-input NOR gate is shown in Figure C.77. In the handshaking circuits for the datapath, control signals re-configure the circuit. The use of a MUX and DEMUX was necessary for the switching of the circuit function. The schematics for a 2-input MUX and 2-output DEMUX are shown in Figures C.75 and C.76.

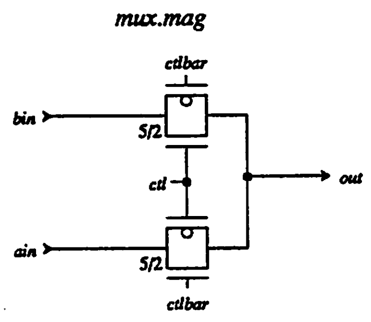


Figure C.75: Schematic for 2-input MUX used in the handshake circuits.

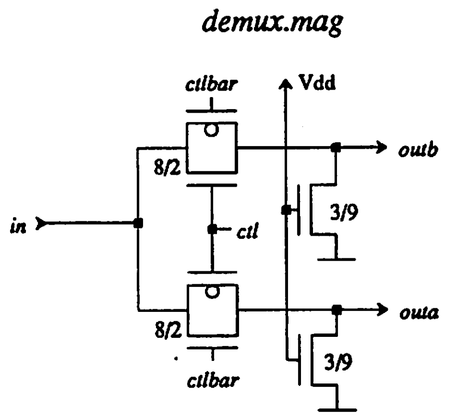


Figure C.76: Schematic for 2-output DEMUX used in the handshake circuits.

Several buffers were used to drive long lines between cells or to the pads. The schematic for *outbuff.mag* is shown in Figure C.78 and the schematic for *bigbuff.mag* is shown in Figure C.79. The inverter *ctlbuff.mag* was used mainly to invert control signals, providing the complementary inputs required in the MUX and DEMUX circuits to drive the CMOS switches. Its schematic is given in Figure C.80. An inverter to drive larger loads in shown in Figure C.81. It is used in the RAM and controller handshaking circuits.

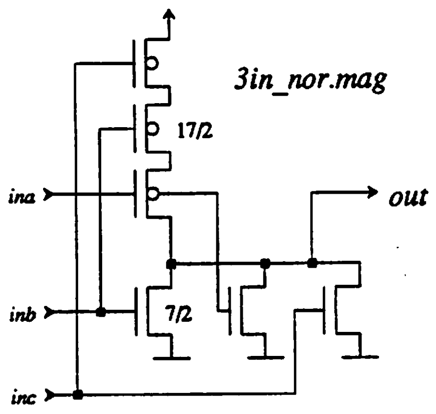


Figure C.77: Schematic for 3-input NOR gate used in the handshake circuits.

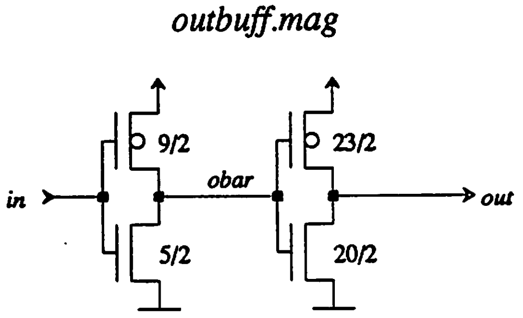


Figure C.78: Schematic for output buffer used in the handshake circuits.

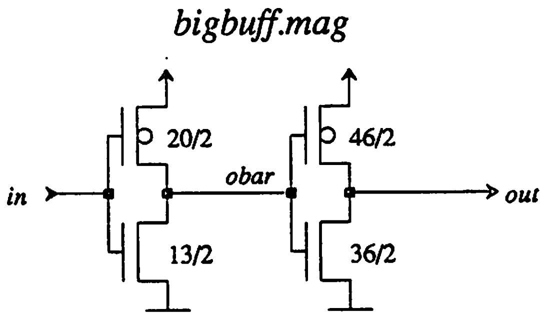


Figure C.79: Schematic for large buffer used in the handshake circuits.

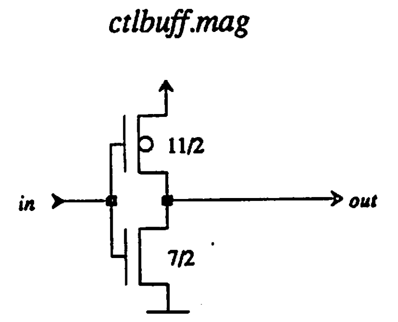


Figure C.80: Schematic for simple inverter used in the handshake circuits.

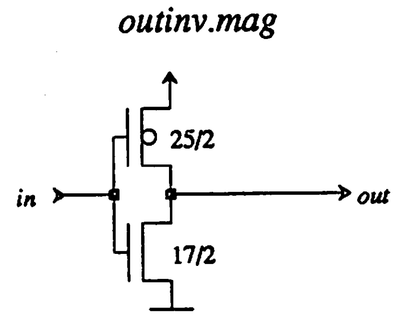


Figure C.81: Schematic for output inverter used in the handshake circuits.

5.2 C-elements

The c-elements used in the handshaking circuits are all derived from the c-element shown in Figure 5.11. *Clatch9e.mag* is shown in Figure C.82 and it is used in the HS4 circuits as well as the RAM and PROM handshake circuits. A three input c-element is *clatch10.mag* and it is shown in Figure C.83. A four input c-element used in the RAM handshake circuit is shown in Figure C.84.

5.3 Miscellaneous

In the controller handshaking, the four condition code signals from the datapath are brought in through a 4:1 MUX circuit. The instruction signals which condition code

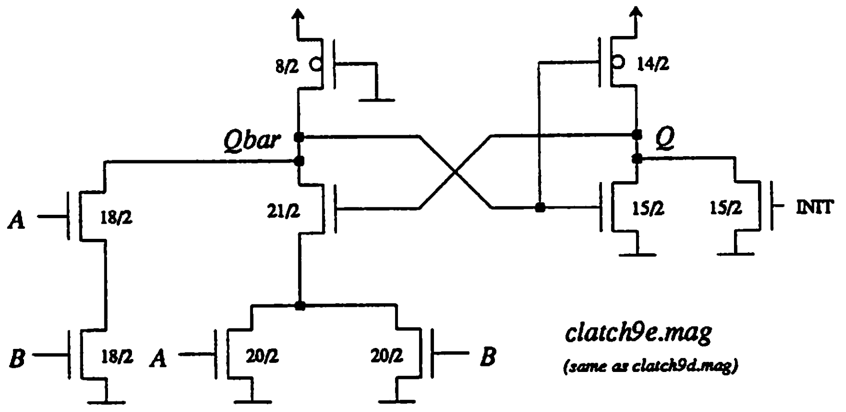


Figure C.82: Schematic for 2-input c-element used in the handshake circuits.

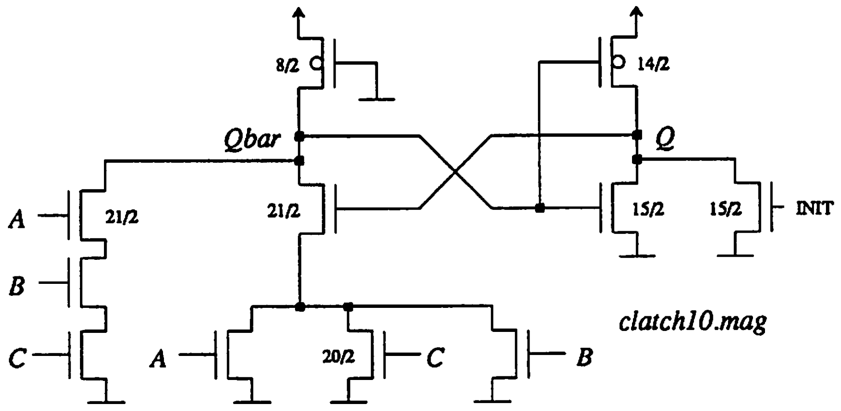


Figure C.83: Schematic for 3-input c-element used in the handshake circuits.

clatchl1.mag

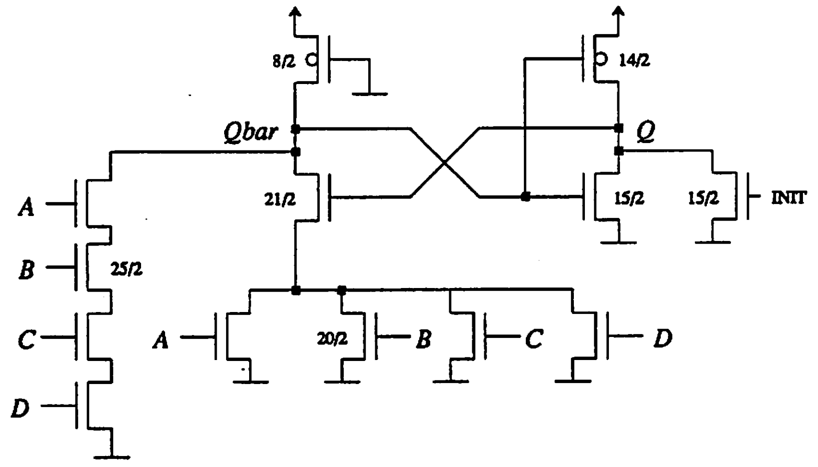


Figure C.84: Schematic for 4-input c-element used in the handshake circuits.

to select for a particular branch. The 4:1 MUX circuit used is shown in Figure C.85 and a decoder that drives its control inputs is shown in Figure C.86.

In the controller handshaking, the sequential handshake circuit that was shown in Figure 7.10 was constructed with two special SR latches which have built in AND ing action on their *set* inputs. The schematics for the latches are shown in Figures C.87 and C.88.

Finally, in the I/O handshake circuits a delay is used to simulate the delays of the FIFO registers. The cell to do this is named *regDV.mag* and it is shown in Figure C.89. The placement of the *regDV.mag* cell is illustrated in Figure 7.17.

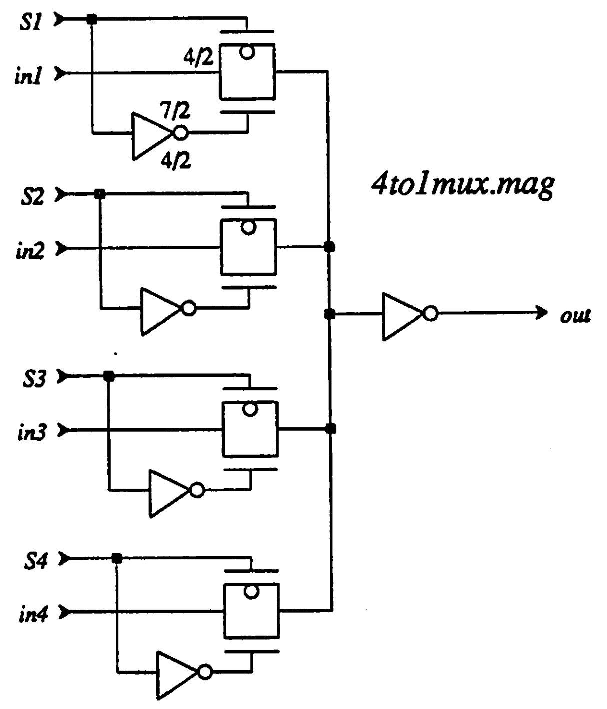


Figure C.85: Schematic for 4:1 MUX circuit which selects condition code fromthe datapath in the controller handshaking (*ROMhs.mag*).

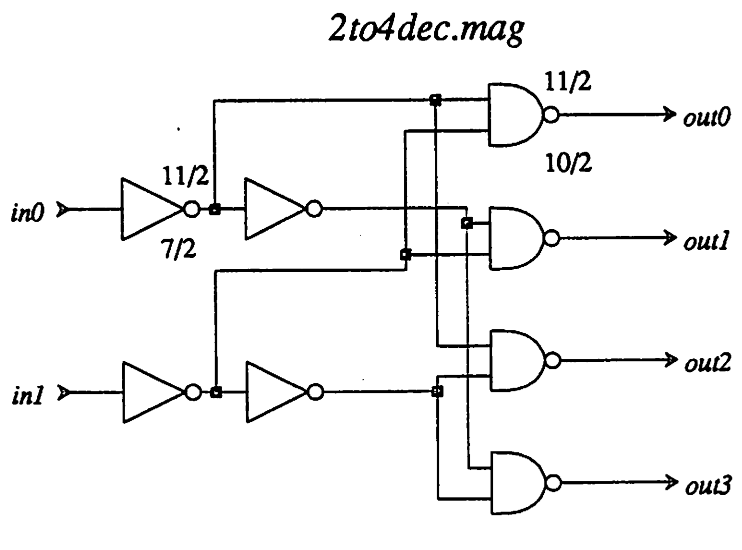


Figure C.86: Schematic for 2-to-4 decoder circuit used to drive the 4:1 MUX above.

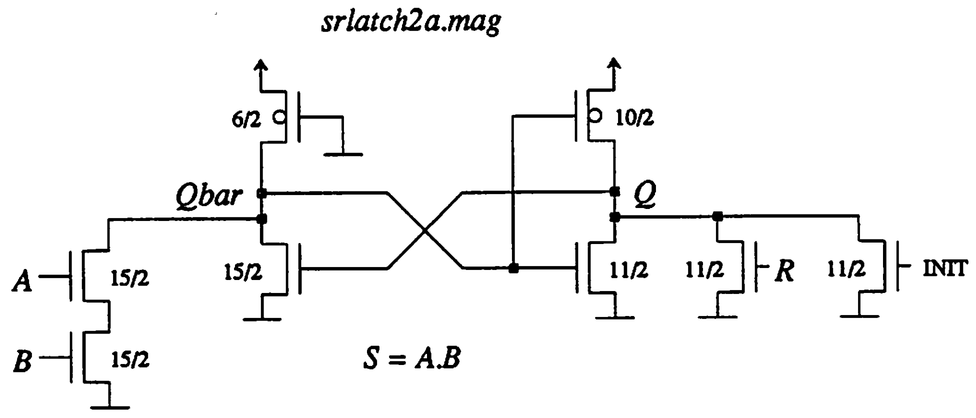


Figure C.87: Schematic for SR latch used in the sequential handshake circuit in the controller. Latch is *set* by raising both A and B inputs.

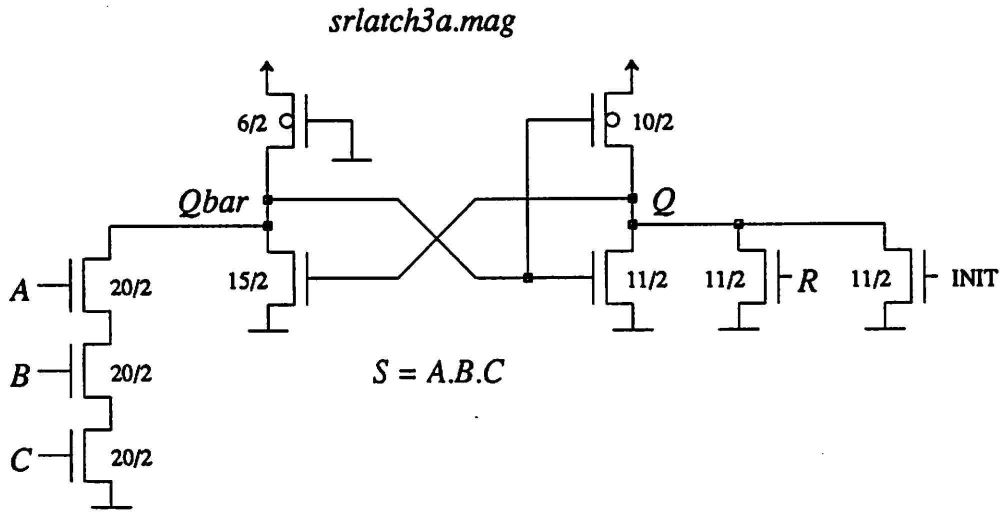


Figure C.88: Schematic for SR latch used in the sequential handshake circuit in the controller. Latch is *set* by raising A, B, and C inputs.

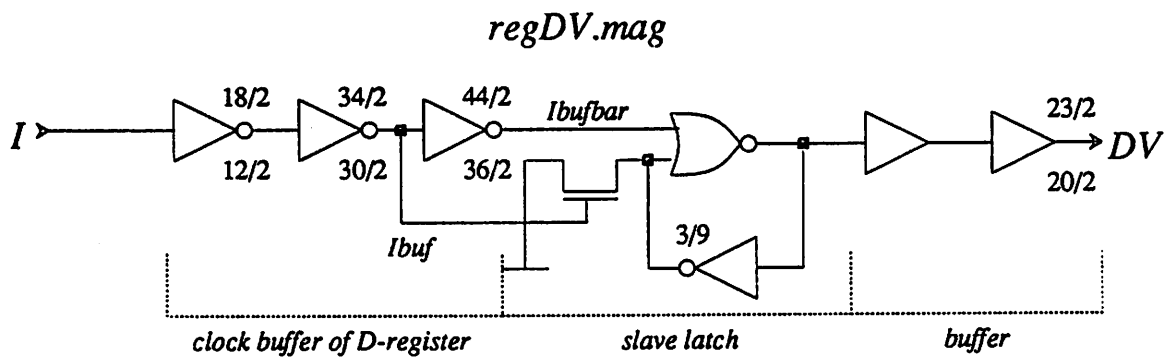


Figure C.89: Schematic register delay cell used in the Input/Output handshake circuits.