Procedural Layout of a High-Speed Floating-Point Arithmetic Unit

Robert Clyde Armstrong

Technical Report 508

June 1985

Massachusetts Institute of Technology
Research Laboratory of Electronics
Cambridge, Massachusetts 02139

Procedural Layout of a High-Speed Floating-Point Arithmetic Unit

Robert Clyde Armstrong

Technical Report 508

June 1985

Massachusetts Institute of Technology
Research Laboratory of Electronics
Cambridge, Massachusetts 02139

This work has been supported in part by the U.S. Air Force Office of Scientific Research Contract F49620-84-C-0004.

Procedural Layout of A High-Speed Floating-Point Arithmetic Unit

by

Robert Clyde Armstrong

Submitted to the
Department of Electrical Engineering and Computer Science
on May 2, 1985 in partial fulfillment of the requirements
for the Degree of Master of Science.

Abstract

This thesis presents a case study in the procedural design of the layout of a complex digital circuit. This is the task of writing a program which constructs a VLSI circuit layout given a set of variable parameters which specify the desired functionality of the circuit. We present a set of techniques for guaranteeing that the constructed circuit obeys the geometric and electrical design rules imposed by the underlying circuit technology. These include a set of simple circuit forms and composition rules for building precharged combinatorial circuits which are free of critical race conditions and charge-sharing problems. As an example, we carry out the creation of a program for building a floating-point addition unit which has selectable number of bits of exponent and fraction in the floating-point representation. The high-level design of a companion floating-point multiplication unit is also discussed.

Thesis Supervisor: Jonathan Allen

Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I wish to thank:

Jonathan Allen for providing valuable advice, criticism, and support throughout this project;

Cyrus Bamji for illuminating discussions on the topic of automated design;

Steve Mccormick for keeping his circuit extraction program EXCL working despite my best efforts to break it;

John Wroclawski for maintaining the computer systems on which this research was done;

And finally, my parents for much love and support over the years.

This research was supported in part by U. S. Air Force Office of Scientific Research Contract F49620-84-C-0004.

Table of Contents

Table of Figures

Table of Tables

CHAPTER ONE

Introduction

This thesis presents a case study in procedural design of the mask layout of a complex logic circuit module. By "procedural design", we mean that the layout is described by a computer program (or procedure) which contains instructions for constructing the final layout. This is in contrast to "hand design" where the entire layout is manually described and perhaps encoded in machine readable form.

A major advantage of procedural design is that the design procedure can accept user specified input parameters which control the process of building the target layout. Rather than hand design a new layout for each application, we can encode the method for constructing a related set of layouts into a single design procedure. A classic example of this is PLA generation. A typical PLA generation procedure accepts as input the number of inputs, outputs, and product terms, and tables specifying the desired logic function of the PLA. The PLA generator then constructs a PLA of the appropriate size and functionality. For highly regular structures such as the PLA, the procedurally designed layout can easily match the performance of a manually designed layout.

We can envision building a set of construction procedures for a wide range of circuit modules. There are several advantages to having available a library of layout design procedures:

- Design time for building a large project is greatly reduced since the designer does not have to manually design the component modules.
- Modification of the component modules is simplified since only the input parameters to the generation procedure need to be changed.

There is also a major disadvantage to procedurally designed layout: Except when the layout is naturally highly regular (e.g., PLA generation), a procedurally designed layout cannot approach the performance (speed, power consumption, and area) of a manually designed layout.

1.1 Overview

This thesis investigates the construction of a design procedure for a kind of module which is not obviously highly regular. One module which we might consider building procedurally is a floating-point arithmetic unit. Such a module would be useful in a signal processing application where the entire signal processor is to be placed on a single chip.

At present, the most efficient method for incorporating floating-point arithmetic into a signal processing unit is to combine a general-purpose floating-point chip (or chip set) with an application-specific chip. The available general-purpose arithmetic units each support a particular fixed floating point representation format and a set of basic operations (either the proprosed IEEE standard P754 or very similar). These chips fall into two performance categories, high-speed (> 1 Mflop) and medium-speed (10 to 100 Kflop). The medium-speed chips are implemented using microprogrammed sequential logic and are suitable for general purpose computing in microprocessor based systems [1]. The high-speed chips are implemented using parallel combinatorial logic and are suitable for signal processing and high-performance general-purpose computing [2] [3]. The medium-speed units consist of a single chip while the high-speed units have one chip per major function type (addition/subtraction, multiplication, division).

This design approach has the following disadvantages:

- At least two or three chips are required to implement a complete system.
- Since the arithmetic units are "general-purpose" they include features for use in many different applications. In a particular application where we would want to build a single chip processor, it is likely that only one set of features is needed for that application. Procedural design of the arithmetic unit would allow the designer to select exactly the features he needs.
- The standard arithmetic units have fixed word size in the floating-point representation format. This obligates the designer to use the full word size in all of the data paths of his system. A procedurally designed arithmetic unit could have adjustable word size so that the designer could select the proper amount of numeric precision for his particular application. For applications where the standard word sizes are excessive, the designer could reduce the width of the data paths throughout his system to obtain decreased area, delay, and power at the expense of numeric precision.

In the following chapters we carry out the design of procedures for constructing floating-point

adder and multiplier units. We place special emphasis on the use of techniques to guarantee correctness of the layouts generated by the procedures without having to directly verify the correctness of every design that each procedure is capable of producing.

Chapter 2 defines the basic format for floating-point representation which is used. This format is based on the proposed IEEE standard.

Chapter 3 presents the circuit methodology used in the construction of the arithmetic units. This methodology is based on self-timed circuit principles and results in race-free precharged combinatorial circuits. A special purpose simulation program for verifying the adherence of a switch-level circuit to the rules of the design methodology is also presented.

Chapter 4 describes the development of the floating-point adder unit in detail from basic algorithm description to layout construction. The programming methodology for the procedural design is described in detail here.

Chapter 5 describes the high-level design of the floating-point multiplier unit based on previous work on the construction of variable-size fixed-point multiplier units. Conclusions and other comments are presented in chapter 6.

CHAPTER TWO

Floating-Point Arithmetic

Having decided to design a floating-point arithmetic unit we need to select a specific format for representing floating-point numbers. An obvious first choice is the proposed IEEE standard 754 for binary floating-point arithmetic [4]. This standard has become widely accepted in the integrated circuit industry.

2.1 Adapting the IEEE Standard for Special-Purpose Use

The standard provides for a wide variety of exceptional events (e.g. underflow, overflow, illegal operation, etc.) as well as several modes of round-off. General-purpose single-chip floating-point arithmetic units typically implement most of the functionality called for in the standard. The user then selects the operational modes and the style of exception handling best suited for the task at hand. This is most useful for applications where many different kinds of computational tasks are required of the same hardware.

For our special-purpose hardware we expect that a particular instance of a procedurally generated design will be used for only one kind of task. To get the fastest and most compact circuit we want to specify the operational mode at design time. This way we do not add extra hardware to handle conditions which will never arise.

The features of the IEEE standard which we will use are the actual data format and the criteria for arithmetic accuracy (methods of round-off.) The standard specifies two basic formats:

- Single precision format divides 32 bits into an 8-bit exponent, a 23-bit fraction, and a sign bit.
- Double precision format divides 64 bits into an 11-bit exponent, a 52-bit fraction, and a sign bit.

The interpretation of a single precision number X is specified as follows: Let e denote the value of the 8-bit exponent considered as an unsigned integer, f denote the value of the 23-bit fraction with radix point to the left, and s denote the value of the sign bit. Then the value v of X is as follows:

1. If $e = 255$ and $f \neq 0$, then v = NaN (Not a Number).
2. If $e = 255$ and $f = 0$, then $v = (-1)^s\infty$.
3. If $e = 0$ and $f \neq 0$, then $v = (-1)^s 2^{-126} f$.
4. If $e = 0$ and $f = 0$, then $v = (-1)^s 0$ (zero).
5. Otherwise, $v = (-1)^s 2^{e-127}(1 + f)$.

For our special purpose application we can simplify the interpretation rules above.

The first two kinds of values are used to represent exceptional results; infinity for overflows and NaN for meaningless results such as 0/0 or $0 \cdot \infty$. The purpose of these kinds of values is to allow an erroneous intermediate result to propagate through a series of calculations in a natural fashion so that the errors can be seen in the final result(s). Since we are interested in signal processing applications, we want our system to be able to recover from transient overflow conditions. This can be achieved by replacing an overflow value with a value of the same sign and largest possible magnitude as in fixed-point saturation arithmetic [5]. Since we are only interested in addition, subtraction, and multiplication this eliminates the need for the infinity and NaN values.

The third kind of value is called *denormalized* and is used to represent underflowed numbers less than the smallest normalized number (in magnitude). This form of number is useful for minimizing the step between the smallest non-zero number and zero [6]. However, it is expensive to implement in both delay and area. We will emulate commercial floating-point ALU's and provide for replacement of underflow results with zero. Denormalized arithmetic support will be left as an option to the constructor procedure (and will not actually be implemented in this thesis).

The fourth kind of value is a zero. Note that the sign of zero may be negative! This feature is designed to be of use in interval arithmetic and other special forms [7]. For our special purpose application we only need one zero value.

The remaining kind of value is the *normalized* number. Note that the effective value of the fraction is $1 + f$; the most significant bit of the normalized number is always 1 so there is no need to include it in the actual representation. A *bias* is applied to the the exponent so that every normalized number has a normalized reciprocal. Since we have no need for the NaN and infinity values we can extend the range of normalized numbers by one binary order of magnitude.

The final modification to the standard format is to allow arbitrary exponent and fraction sizes: N_e bits and N_f bits respectively. Using the modified format, the value v of the $1 + N_e + N_f$ bit string X is as follows:

1. If $e = 0$ and $f = 0$, then $v = 0$.
2. If $e = 0$ and $f \neq 0$, then v = illegal (reserved for denormalized values).
3. Otherwise $v = (-1)^s 2^{e - bias}(1 + f)$ where $bias = 2^{N_e - 1} - 1$.

The illegal values are never produced as results and should not be given as operands.

The remaining features of the IEEE standard which need to be considered are the round-off modes. The standard specifies that all arithmetic operations must be performed as if using infinite precision followed by the selected rounding operation. The types of rounding are:

- Round to "nearest", with round to "even" in case of a tie. The closest representable number to the infinite precision result is used. If two numbers are equally close then the one with least significant fraction bits zero is used. This corresponds to the usual definition of "round-off".
- Round toward zero. The representable number with the largest magnitude less than or equal to the infinite precision result is used. This is often called "unbiased truncation".
- Round toward positive infinity.
- Round toward negative infinity.

The last two forms of round-off represent "biased truncation" where round-off errors are always of the same sign. The first two forms are the most useful for signal processing applications since the average error introduced is zero.

The round-to-nearest mode provides the best signal to noise ratio, the advantage increasing with more complex algorithms such as the FFT [8]. This mode requires three extra bits of raw result for correct rounding in all cases [4]. Since the round-to-zero mode requires only two extra bits and very little additional circuitry it is useful to have both modes available to the designer so that he can select the best speed/area vs. precision trade-off for his application.

2.2 Basic Addition and Multiplication Algorithms

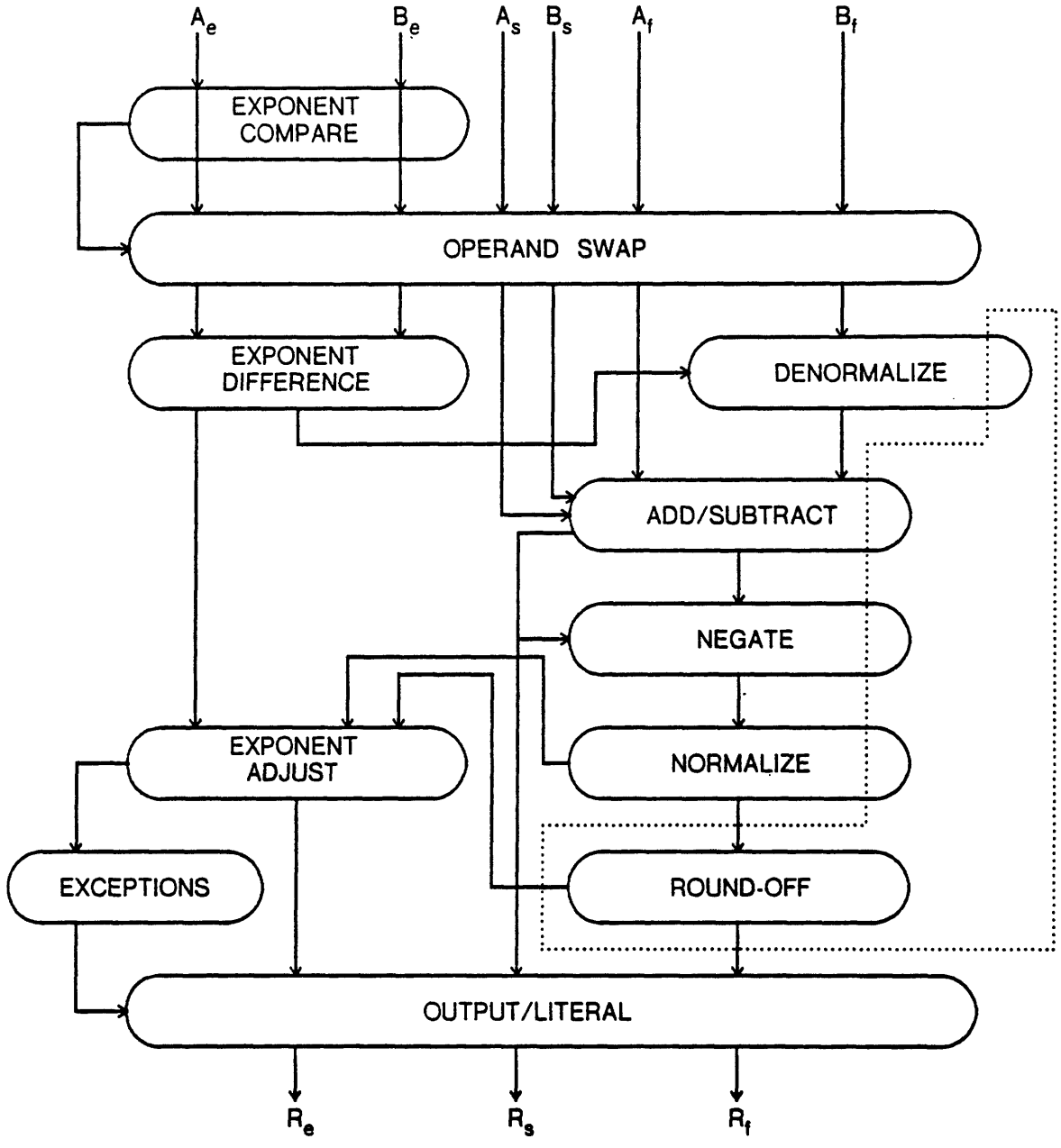


Figure 2-1: General algorithm for floating point addition

Figures 2-1 and 2-2 show simplified flow graphs of the floating-point addition and multiplication processes respectively. The dotted lines enclose the additional circuitry needed for round-to-nearest operation. The details of these processes will be filled in in later chapters since they depend somewhat on the technology and circuit forms used in the implementation.

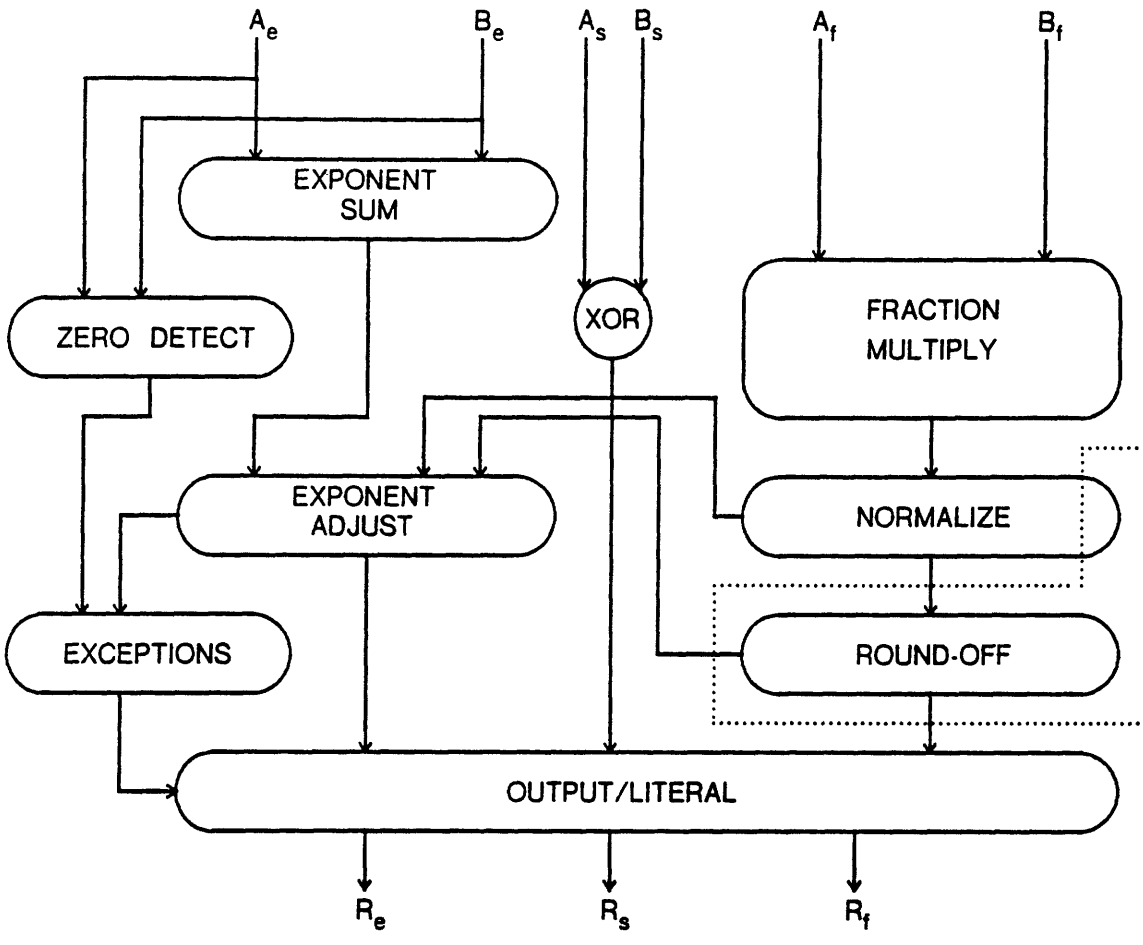


Figure 2-2: General algorithm for floating point multiplication

The steps of the addition process are:

1. Compare exponents of the operands: swap the operands as necessary to route the operand with smaller exponent through the denormalizer.
2. Compute the positive difference between the exponents and use this value to denormalize the smaller operand into alignment with the larger.
3. Add or subtract the adjusted operands. If the result is negative, then negate it to obtain the magnitude.
4. Normalize the result value by shifting it until the most significant bit is a 1.
5. Adjust the larger exponent by subtracting the value of the normalizer left shift count (this count may actually be -1 in case of a right shift after adder overflow.)
6. (For round-to-nearest operation only) Examine the three guard bits below the least

significant bit of the result and, if necessary, increment the result. A final exponent adjustment may be necessary if the increment overflows.

7. Examine the results of the exponent adjustment for underflow or overflow. If underflow is indicated, then output a zero result. If overflow is indicated, then output a value with largest possible magnitude and same sign as the result.

Note that subtraction can be done by simply inverting the sign of the operand to be subtracted before performing the addition.

The steps of the multiplication process are:

1. If either operand is zero then immediately output a zero result.
2. Compute the product of the unsigned fraction parts of the operands.
3. Compute the sum of the exponents, compensating for the bias.
4. Normalize the fraction product. Since the operand fractions are in the range $1.0 \leq f < 2.0$, the result fraction will be in the range $1.0 \leq f < 4.0$. This means that at most one right shift will be needed to normalize the result.
5. If a normalization shift is required, increment the result exponent.
6. (For round-to-nearest operation only) Examine the guard bits of the normalized result and increment the result if necessary. Adjust exponent if necessary.
7. Compute the result sign as the exclusive-or of the operand signs.
8. Examine the result exponent for underflow or overflow. Handle these exceptions the same way as for addition.

Note that the only functional block common to both the addition and multiplication processes is the result round-to-nearest unit. Since the multiplication and addition processes are so radically different there is very little advantage in speed or area to be gained by attempting to build a single piece of high-speed hardware to perform both tasks. For this reason, we will want to design two independent modules, one for addition and subtraction and the other for multiplication. This gives the end user the greatest freedom in configuring his system.

CHAPTER THREE

Circuit Structures

In order to build easily verifiable circuits which implement the floating-point arithmetic functions we first need to specify a circuit design methodology. Since we are building modules which will become part of larger VLSI systems, we need to make the module interface simple so as to place the fewest constraints on the rest of the system. The simplest interface is obtained by implementing the modules as static combinatorial gate networks. In this case, the surrounding network simply applies a set of stable input values and waits long enough for input changes to propagate to all outputs before sampling the output values. Unfortunately, there is a trade-off between simplicity and performance. The static combinatorial modules would not be able to make use of the clock signals which are usually available in complex MOS systems. In particular, such modules would not be able to make effective use of a common technique for improving the performance of MOS circuits, namely: *precharging*.

Precharging is used to decrease both power dissipation and propagation delay in MOS circuits. A simple example is to replace the depletion pullup of an NMOS gate with a clocked pullup. Figure 3-1 shows the circuit and timing waveforms for a precharged inverter in a two-phase clocked NMOS system. During the precharging period the pullup is switched on and the pulldown network is switched off, charging the output capacitance of the inverter to a high voltage. During the compute period, the pullup is switched off and the pulldown may switch on or stay off, depending on the final state of the input. Since the pullup and pulldown networks are never simultaneously on, static power dissipation is eliminated and ratioing of pullup and pulldown sizes is not required.

Precharging is usually only applied to a small subset of nodes in a network where the relative timing of switches connected to these nodes can be carefully controlled. In large combinatorial networks it is almost impossible to make effective use of precharging. The exclusive-nor (or equivalence) gate provides a simple example of a circuit which cannot be precharged in the manner

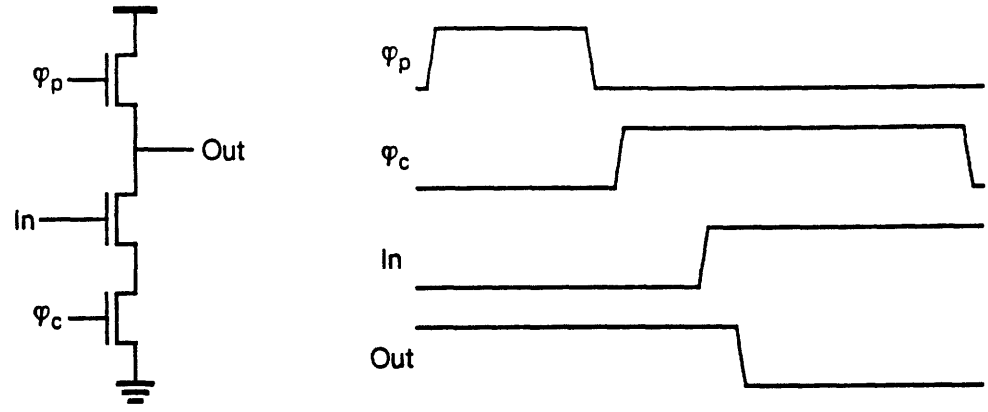


Figure 3-1: Precharged inverter

described above. Figure 3-2 shows the schematic and timing diagram of this gate. The pulldown network is shown as a "black box" since its implementation is irrelevant: it is on when the inputs are at the same logic level and off otherwise. If both inputs are low during the precharge period the output will be properly precharged high. But if both inputs make a low to high transition during the compute phase and the transitions are not simultaneous then the output capacitor will get discharged and the final value will not be a valid high level!

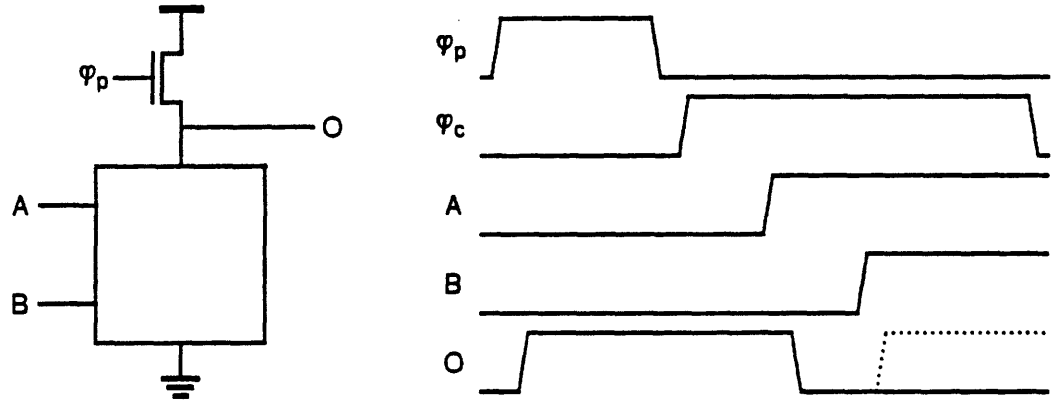


Figure 3-2: Precharged exclusive-nor gate

The rest of this chapter describes a circuit methodology which allows precharging to be used extensively throughout a large combinatorial network. We will use a form of *self-timed* logic where signal transitions are controlled so that the critical race problems described above cannot occur. A

form of this kind of circuitry was first used by Masumoto [9]. This methodology will be carefully developed to allow the proper behavior of a network to be verified in a modular fashion.

3.1 Overview

In order to develop the self-timed circuit forms we will need to make a clear distinction between two levels of abstraction used to describe logic networks. The upper level is the *logical* level in which nodes have boolean truth values and elements implement boolean functions of arbitrary complexity. The lower level is the *physical* level in which nodes have voltage values (e.g. *high* or *low*) and basic elements perform very simple operations as dictated by the available device technology. In conventional digital circuitry the mapping between the logical and physical domains is straightforward: Each logical mode maps one-to-one to a physical circuit node and each logical element maps to a specific set of interconnected physical elements (gates.) Usually the only important feature of the mapping is whether a node is "active low" or "active high" (i.e. {*high low*} => {*false true*} or {*high low*} => {*true false*}.)

In the self-timed circuit discipline described below, the mapping between the logical and physical domains is more complex. Listed below are brief descriptions of the most important terms which will be used in the rest of the chapter. The full meaning of each of these terms will be defined in later sections.

- A *module* is a combinatorial logic element with a set of input pins and a set of output pins. Its internal structure is hidden from view. At the logical level of abstraction the module produces at its outputs a boolean function of its input signals.
- A *signal* is a time-varying logical value which is associated with the input or output pins of a logic module. A signal corresponds to a node of a logical network and does not necessarily correspond to any single node in the physical implementation of the logical network.
- A *network* is a set of interconnected logical modules. Like a module, a network has a set of input and output pins. We will show that a well-formed network can be considered to be a module itself; that is, the internal structure can be ignored and the outputs treated as boolean functions of the inputs. In later sections we will give rules for interconnecting modules so that the useful properties of the underlying modules are preserved by the network considered as a module.
- A *logical state* is the Boolean truth value of a signal at a particular point in time.
- A *circuit* is a physical implementation of a logical module or network.

- A *wire* is an interconnect node in the physical implementation of a logical network. A logical signal is implemented as a set of one or more physical wires. The logical state of the signal is a function of the logic levels of its component wires.
- A *logic level* is a physical wire state at a particular point in time. It is one of the set {*low high*} (or {0 1}.)

3.2 Generalized Self-timed Combinatorial Circuits

We now need to specify the desired behavior of a self-timed combinatorial module.

Definition 1: A logical module is *well-behaved* if it obeys all of the following:

a. Each module pin (input or output) has a definite state at a given point in time. This state is either the null or undefined state (Ø) or is one of a set of one or more defined states (usually the set {*true*, *false*}.) At any point in time, the pin must be in one of these *valid states* or must be making a transition between the null state and one of the other valid states; there is never a transition between two non-null states.

b. Each module pin is either an input or an output, not both. An input pin must not have any adverse effect on the state of the signal source which drives it, i.e., the input must not cause an otherwise well-behaved module output pin to misbehave. An output pin must be a *signal source*. A signal source is a logical entity which is analogous to a low-impedance voltage source at the physical level.

c. If all input pins are set to the null state then all of the output pins must eventually switch to the null state. The module is then said to be in the *null state*.

d. If each input pin is set to a non-null state then each output pin must eventually switch to a non-null state. The module is then said to be in the *terminal state*.

e. If the network was previously in the null state and each input made a single transition to a non-null state then each output must make one and only one transition to a non-null state. In making a transition, it is permissible for the signal to oscillate between the null state and its terminal state.

f. The terminal state of each output must be dependent only on the terminal state of the inputs. In particular, the terminal state of the outputs does not depend on either the ordering of the input transitions or any previous state of the module. In short, the terminal output state must be a static function of the terminal input state.

This set of constraints is based on a subset of Seitz's "weak conditions" for fully asynchronous circuit modules [10] (chapter 7).

Some discussion of the purpose for each section of this definition is needed. Paragraph a is axiomatic and defines the characteristics of the logical signals in the circuit. If the state of a signal is Ø then we know its value has not yet been computed.

Paragraph b ensures that information flows from inputs to outputs only.

Paragraph c guarantees that the module outputs can be set to the null state merely by setting all the inputs to the null state. This property is vital for precharged circuits.

Paragraph d guarantees that when all the inputs to the module are defined the outputs must eventually become defined. This property is required to prevent deadlock in logical networks.

Paragraph e guarantees that the module is free from static hazards at the logical level, i.e., no output ever enters a transient non-null state which is different from the terminal state to which it eventually settles. This is required so that there is no ambiguity about the identity of the terminal state of an output; the instant that the output enters a valid non-null state we can be sure that the output will eventually settle to that state hence the computation is effectively completed for that output. This property will also be required for proper operation of precharged circuits. We permit the oscillation between null and terminal state in this abstract model because it is impossible to exclude such behavior from the physical implementation where noise is present.

Paragraph f guarantees that the module is free from critical races. This is an obvious requirement for well-behaved combinatorial circuits.

Definition 1 sets forth the requirements for a well-behaved module. We now set forth rules for building well-behaved logical networks from well-behaved modules.

Theorem 1: Connect a set of modules $\mathcal{M}_k$ together to form a network $\mathcal{N}$ such that:

a. Each node in the network has at most one output pin connected to it.
b. For each node, every pin connected to that node has the same set of possible logical states.
c. There are no loops in the network. That is, starting at any node in the network, it is not possible to return to that node by traveling from a node to an input pin of a module, then to the output pin of that module, etc.

Consider $\mathcal{N}$ as a module where the input pins of $\mathcal{N}$ are those nodes which have no $\mathcal{M}_k$ output connected to them and the output pins are a subset of the set of output nodes of the $\mathcal{M}_k$.

If each module $\mathcal{M}_k$ is well-behaved then $\mathcal{N}$ is a well-behaved module.

In order to prove the theorem in general form we first need to prove it for two specific cases.

Lemma 2: Given two well-behaved modules $\mathcal{M}_1$ and $\mathcal{M}_2$:

a. If $\mathcal{N}_P$ is the *parallel composition* of $\mathcal{M}_1$ and $\mathcal{M}_2$ then $\mathcal{N}_P$ is well-behaved. The parallel composition of two modules is the network whose input pins are the set of all input pins of the two component networks (with perhaps some of the inputs connected together) and whose outputs are the set of all output pins of the component networks.

b. If $\mathcal{N}_S$ is the *serial composition* of $\mathcal{M}_1$ and $\mathcal{M}_2$ then $\mathcal{N}_S$ is well-behaved. Here $\mathcal{N}_S$ is the network formed by connecting the outputs of $\mathcal{M}_1$ to some or all of the inputs of $\mathcal{M}_2$ (the interconnection rule in theorem 1b must be obeyed.) The inputs to $\mathcal{N}_S$ are the inputs to $\mathcal{M}_1$ and the inputs to $\mathcal{M}_2$ which do not connect to outputs of $\mathcal{M}_1$. The outputs of $\mathcal{N}_S$ are the outputs of $\mathcal{M}_2$ and perhaps some of the outputs of $\mathcal{M}_1$.

Figure 3-3 shows examples of serial and parallel composition.

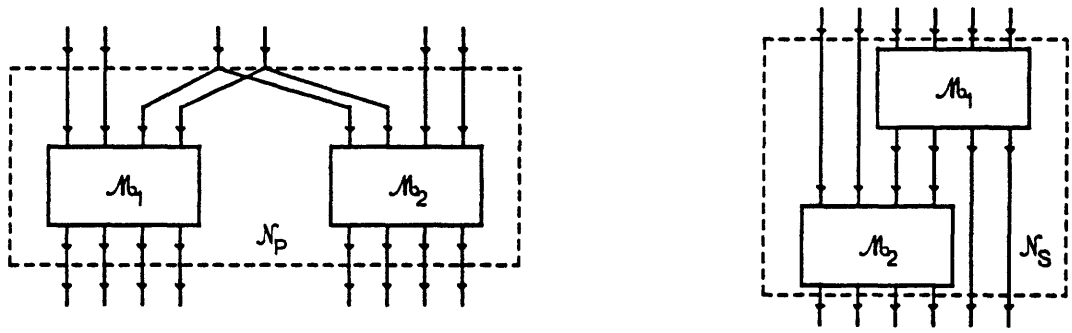


Figure 3-3: Basic forms of module composition

Proof: Lemma 2a is trivially true. By definition 1b, the interconnection of inputs of the sub-modules cannot have adverse effects on the module behavior. Therefore since each input or output pin of the sub-modules is a pin of $\mathcal{N}_P$ and each sub-module conforms to the criteria in definition 1, the entire module is well-behaved.

Lemma 2b is not so obvious. The pins of $\mathcal{N}_S$ obey definition 1a,b since the pins of $\mathcal{N}_S$ are pins of modules which obey 1a,b. If all inputs to $\mathcal{N}_S$ are set null then eventually all outputs of $\mathcal{M}_1$ will go null causing the outputs of $\mathcal{M}_2$ and $\mathcal{N}_S$ to go null. Similarly, when all inputs to $\mathcal{N}_S$ are set non-null, eventually each output of $\mathcal{N}_S$ will go non-null. If each input to $\mathcal{N}_S$ makes exactly one transition from null to non-null then each output of $\mathcal{M}_1$ makes exactly one transition from null to non-null which causes each output of $\mathcal{M}_2$ and $\mathcal{N}_S$ to make exactly one transition. Similarly, $\mathcal{N}_S$ obeys definition 1f since both sub-modules obey it. Thus $\mathcal{N}_S$ is well-behaved.

To prove theorem 1 we first partition the $\mathcal{M}_k$ into two sets $\mathcal{M}_{in}$ and $\mathcal{M}_{out}$ where $\mathcal{M}_{in}$ is the set of all modules in $\mathcal{N}$ whose inputs are all inputs of $\mathcal{N}$ itself (i.e. no module in $\mathcal{M}_{in}$ has an input connected to the output of any other $\mathcal{M}_k$.)

We know that $\mathcal{M}_{in}$ is non-empty since there are no loops in $\mathcal{N}$. We partition $\mathcal{N}$ into two networks $\mathcal{N}_{in}$ and $\mathcal{N}_{out}$ which consist of the nodes in $\mathcal{M}_{in}$ and $\mathcal{M}_{out}$ and their interconnections respectively. Then $\mathcal{N}_{in}$ is a parallel composition of well-behaved modules $\mathcal{M}_{in}$ and is therefore well-behaved. Also $\mathcal{N}$ is a serial composition of $\mathcal{N}_{in}$ and $\mathcal{N}_{out}$.

We can prove that $\mathcal{N}_{out}$ is well-behaved by repeating the proof above recursively until the output partition is null (since there are only a finite number of $\mathcal{M}_k$ and each input partition is non-null.)

Since $\mathcal{N}$ is a serial composition of well-behaved modules it is well behaved.

3.3 Gate Level Implementation of Self-timed Modules

Now that we have general set of rules for building well-behaved networks given well-behaved modules we need a set of primitive well-behaved modules for building useful networks. The forms given in this section and later sections do not represent the complete set of all possible forms which could be used to implement the modules described above. They do constitute a set of forms which is sufficient to implement any desired logic function and is straightforward to implement and verify as MOS circuits.

In order to build a physical circuit module we need to find a physical form for the multi-state logical signals. We will build a multi-state logical signal using a set of *physical wires* each of which has two possible states, namely *high* or *low*. An *encoding* is a mapping between the states of a logical signal and the states of its components wires.

In the examples below all signals have three possible valid states {Ø, *true*, *false*}. We will represent each signal using a pair of wires. Upper case letters are used to denote logical signals while lower case letters are used to denote physical wires.

The *encoding* of a signal, A, is shown in figure 3-4a. Both wires are *low* in the null state. Wire *a* is asserted *high* for the A *true* state. Wire *a'* is asserted *high* for the A *false* state. This is called the *active-high dual-rail* encoding (or simply the active-high encoding.) Note that there are four possible states of the physical wire pair. State X where both wires are *high* is not permissible as a valid state in our design methodology; to switch from the null state to the X state without passing through the *true* or *false* states would require simultaneous transitions on the two wires. In a practical circuit we cannot guarantee simultaneous transitions.

The implementation of a logical AND gate module using physical AND and OR gates is shown in figure 3-4c.

a	a'	A
L	L	Ø
L	H	F
H	L	T
H	H	X

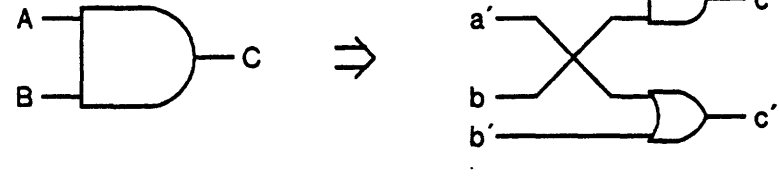


Figure 3-4: Basic AND gate implementation

To prove that this module is well-behaved we need to provide a definition of well-behavedness for physical gates. This definition is a modification of definition 1 for logical modules.

Definition 2: A physical gate is well-behaved if it obeys the following constraints:

a. Each physical pin of the gate has two possible states, *high* or *low*. One state is the inactive state, the other is the active state. If the active state is *high* then the pin is called *active-high* otherwise it is called *active-low*. The initial state of a pin is always inactive, the terminal state of *physical* pin can be either active *or* inactive.

b. The gate must have a set of input pins and a set of output pins which are distinct. Each input pin must present a capacitative load to its driving wire. Each output pin must be able to drive the inputs connected to it to valid states.

c. When all inputs are inactive, all outputs must eventually become inactive.

d. When all inputs have reached their terminal state the output must eventually change, if logically necessary, to obey the gate's function.

e. After all inputs have assumed their terminal states, if each input made at most one transition then the output can only make, at most, one transition. If a wire makes a transition from inactive to active state then it is permissible for the wire to oscillate between the two states before stabilizing in the active state. If the terminal state of a wire is the inactive state the wire *must not leave the valid inactive state.*

f. The output of a gate must be a fixed boolean function of its inputs.

While each of the constraints above follows directly from the constraints on well-behaved modules, some additional discussion of paragraph e is necessary. This constraint is present to exclude gate behavior which would interfere with the proper operation of precharged circuits. If, during a particular computation, the terminal state of a wire is the same as the initial (inactive) state, then we guarantee that the wire never leaves this state during the computation. This ensures that a precharged wire is never discharged by a transient "glitch".

To return to the logical AND gate we observe that the input/output properties of the physical gates are inherited by the logical gate thus definition 1a,b is obeyed.

When the states of A and B are both null then eventually C becomes null since all of the physical gate inputs are *low* thus definition 1c is obeyed.

A	B	a	a′	b	b′	c	c′	C
Ø	Ø	L	L	L	L	L	L	Ø
F	F	L	H	L	H	L	H	F
F	T	L	H	H	L	L	H	F
T	F	H	L	L	H	L	H	F
T	T	H	L	H	L	H	L	T

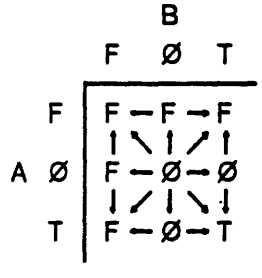


Figure 3-5: Truth tables for self-timed AND gate

The truth table in figure 3-5a shows that when both A and B are non-null C eventually assumes a non-null value obeying 1d. The transition table in 3-5b shows all possible transitions (including simultaneous) of A and B from the null state to their terminal states. It is easy to see that if A and B each make one transition then the output makes exactly one transition, regardless of the ordering of the input transitions, thus definition 1e,f is obeyed.

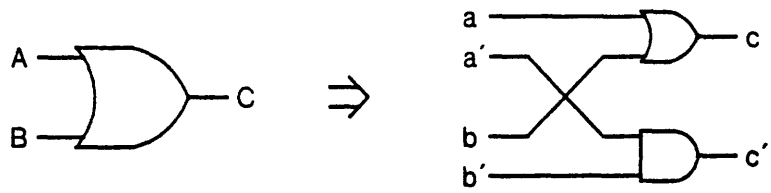


Figure 3-6: Basic OR gate implementation

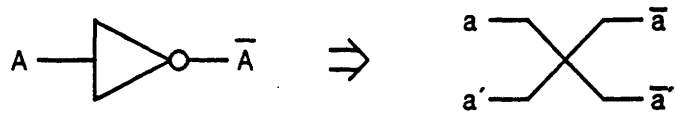
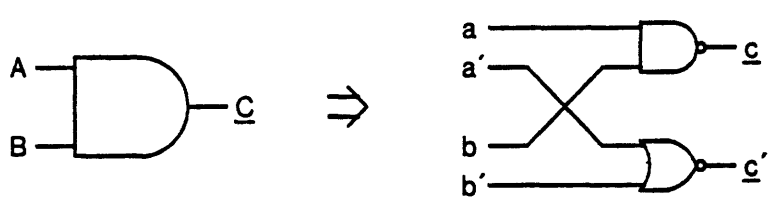


Figure 3-7: Basic inverter implementation

Figure 3-8 shows how a two input AND gate can be built using the inverting gates normally found in transistorized logic. Note that the encoding of the output signal is different from the encoding of the two inputs. If both output wires are *high* then the logical output is null. If the un-primed output is *low* then the logical output is *true* and if the primed output is *low* then the logical output is *false*. This encoding is called the *active-low dual-rail* encoding (or simply the active-low encoding.)



c	c'	C
L	L	X
L	H	T
H	L	F
H	H	Ø

Figure 3-8: Basic AND gate with inverting logic

The underline in the signal names signifies active-low encoding. Note that the negation implied by the inverting gate elements has *no effect* on the truth value of the output signal. It only affects the type of encoding scheme used for the output signal.

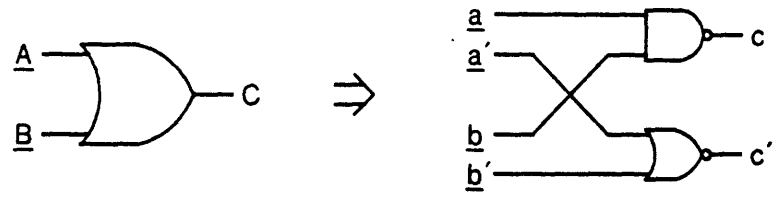


Figure 3-9: Basic OR gate using AND gate circuit

Figure 3-9 shows a different way to encode the pins of the circuit used to implement the AND gate. If the inputs are encoded active-low and the output is encoded active-high then the gate performs the boolean OR function. This is analogous to the way DeMorgan's theorem is used to represent a positive logic AND gate as a negative logic OR gate in conventional digital logic.

Since the fundamental logic gates in NMOS are naturally inverting we will need to have both active-low and active-high encodings present in the network circuits. Since more than one encoding is in use we need additional module interconnection rules to meet the requirements for well-behaved networks. In particular, when two signals are connected together, their encodings must *conform* to each other.

Definition 3: Two connected logical signals A and B *conform* if:

a. There is a one-to-one correspondence between valid states of A and those of B.

b. The null state of A corresponds to the null state of B.

Definition 3 makes explicit the assumption in theorem 1b that connected signals share a common set of valid states. Applying this definition, we find that the active-high and active-low encodings are not *conformable*, i.e. there is no way to connect the wires in, say, A and $\underline{B}$ such that A and $\underline{B}$ conform. Thus in any network of the dual-rail elements discussed above, an active-high pin can connect only to other active-high pins and similarly for active-low pins.

The encoding constraint gives rise to the need for a new type of logical module, the *encoding inverter*. Figure 3-10 shows an encoding inverter built from physical inverters. Its only function is to invert the encoding of the input signal without changing its logical state.

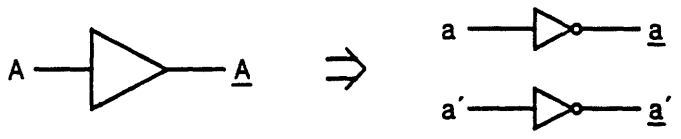


Figure 3-10: Basic encoding inverter

Now that we have a set of basic well-behaved modules "and", "not" and "encoding invert" we can build a well-behaved logical network implementing any given combinatorial function.

3.4 Switch Level Implementation of Self-timed Circuits

Now that we know how to build self-timed networks using logic gates, we need to know how to build gates from available integrated circuit technology. Switch-level or "relay" logic is very useful as a first-order model of both NMOS and CMOS technologies. In this section, we will show how to build well-behaved gates out of networks of switch elements.

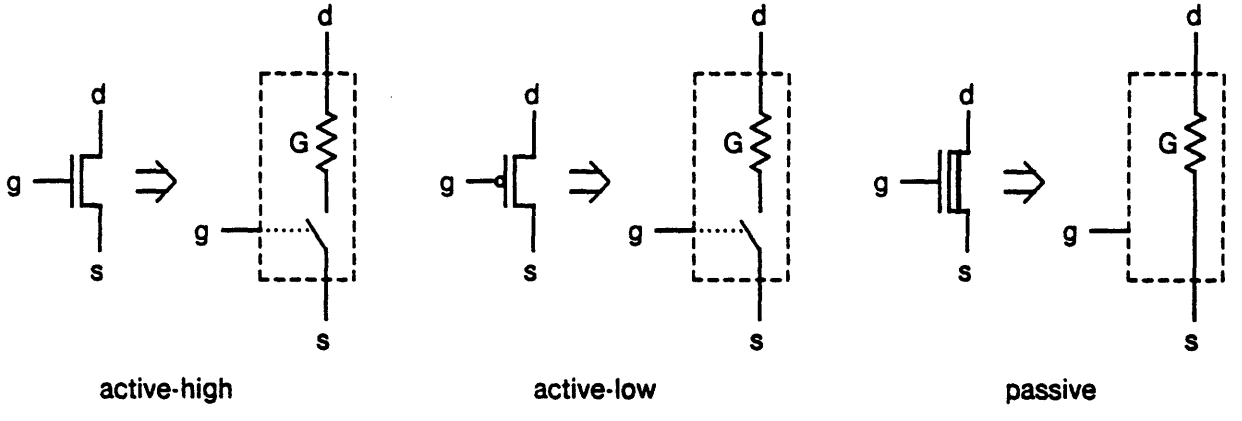


Figure 3-11: Basic switch elements

Figure 3-11 shows the three kinds of *simple switch elements* we will use. The active-high element is only *on* when pin *g* is *high*. The active-low element is only *on* when pin *g* is *low*. The passive element is always *on* independent of the state of its *g* pin. The passive element serves as a static conductance.

Note that MOSFET symbols are used to represent the various switch elements. These symbols represent the devices which are used to implement the switch elements in actual MOS circuits. The active-high symbol is an n-channel enhancement mode MOSFET. The active-low symbol is a p-channel enhancement mode device. The passive element is represented by an n-channel depletion mode device. The pin names *g*, *s*, and *d* are derived from the MOSFET pin names *gate*, *source*, and *drain* respectively.

When discussing logic gates built from MOSFET's there is often ambiguity to the meaning of the word "gate". To remove this ambiguity in the discussion below, we will take the word "gate" alone to mean the gate pin of a switch element and add additional words to specify more abstract objects (e.g., "gate circuit" for a physical logic gate and "logical gate" for well-behaved logical gate module).

To model an arbitrary physical gate circuit we will use a generalized form of switch element.

Definition 4: A *composite switch element* is any interconnection of simple switch elements such that:

a. The composite element has one *s* pin, one *d* pin, and N gate pins g_k.
b. Each gate of each component switch connects to one of the g_k pins.
c. Each *s* and *d* pin of each component switch connects only to other *s/d* pins or to the composite *s* or *d* pins.

From now on, we will use the term *switch element* for both composite and simple switch elements when there is no need to distinguish between the two types.

The conductance between the *s* and *d* pins of switch element E will be referred to as G(E). This value ranges from 0 (open circuit) up to some finite value. As with the simple elements, if G(E) = 0 then we say that E is *off* and if G(E)>0 then E is *on*.

We now define an important property which will be needed later to prove well-behavedness of generalized switch level gate circuits. We will need to know what constraints to place on the structure and gate inputs of a switch element so that any sequence of input transitions during computation results in at most one opening or closing of the switch element.

Definition 5: A switch element E is called *monotone* if given *any* initial state of its gate input(s) with corresponding *s/d* conductance G_0:

- If E is *on* and any open switch in E is closed then E does not switch *off*.
- If E is *off* and any closed switch in E is opened then E does not switch *on*.

The three simple switch elements are obviously monotone. To demonstrate that complex elements are monotone we need a theorem.

Theorem 3: Any parallel or series connection E of monotone switch elements is itself monotone.

Proof: For a series E of switch elements we reason as follows: If E is *on* then all of its sub-elements must be *on*; if any switch is closed then none of the sub-elements can switch *off* hence E remains *on*. If E is *off* then at least one of the sub-elements is *off*; if any switch is opened then none of the *off* sub-elements can switch *on* hence E remains *off*.

For a parallel set of switch elements we find: If E is *on* then at least one of its sub-elements is *on*; if any switch is closed then none of the *on* elements can switch *off* and E remains *on*. If E is *off* then all of its sub-elements are *off* and opening any switch cannot cause any sub-elements to switch *on* hence E remains *off*.

In the following discussion we need to use some terms for certain special states of a switch element E. If all the simple elements in E are *on* then we say that E is *totally on*. If all the non-passive elements in E are *off* then we say that E is *totally off*. Note that a single passive element is *both* totally *off* and totally *on*!

The useful property of monotone switch elements is that if a monotone element E is totally *off* (totally *on*) then G(E) is at a minimum (maximum) and closing (opening) switches one by one in E will cause at most one transition from *off* (*on*) to *on* (*off*).

Figure 3-12 shows the generalized form of switch-level gate which we will use. It has a pull-down element $\mathcal{N}_{PD}$ which has a set of inputs I_{PD}. Its pull-up element $\mathcal{N}_{PU}$ has a set of inputs I_{PU}. In addition there are N pass elements $\mathcal{N}_{PA_k}$ each having a single pass input I_{PA_k} and a set of gate inputs I_{PG_k}. Finally there is a capacitance C between the output node and ground. We will discuss the effects of parasitic capacitances later.

To simplify the statement and proof of the theorem below, we define two terms to describe the switch elements in the basic logic gate: If O is active high then we call $\mathcal{N}_{PD}$ the *inactive source*

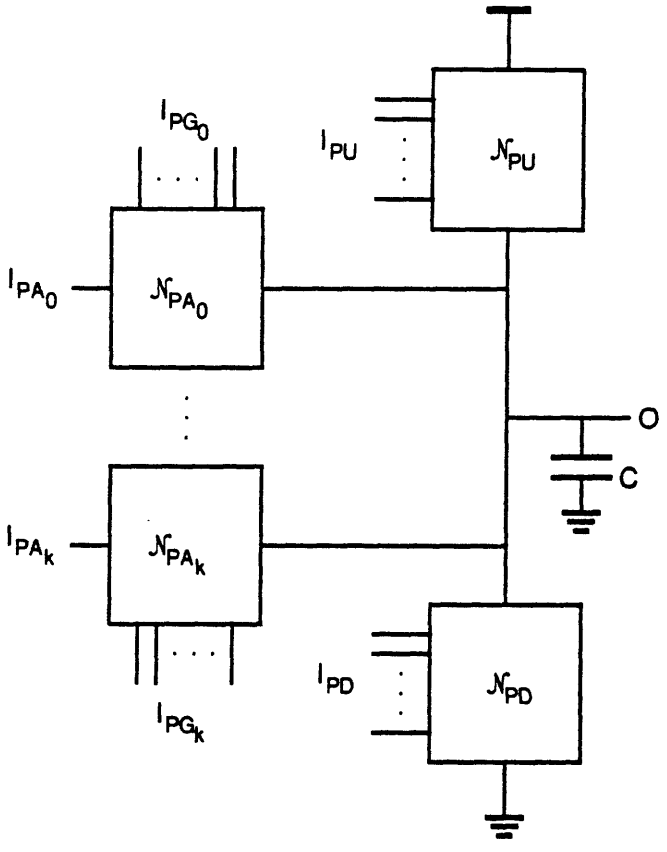


Figure 3-12: Generalized switch gate module

otherwise if O is active low then we call $\mathcal{N}_{PU}$ the inactive source; If an element is not the inactive source then it is called an *active source*.

We can now present the theorem for well-behavedness of switch level gates.

Theorem 4: If a switch level gate of the form shown in figure 3-12 meets each of the following constraints:

a. The gate must have at least one active source in addition to the inactive source.
b. At no time may more than one active source be *on*. If there is more than one active source then all active sources must be *off* when all inputs are inactive.
c. If one of the $\mathcal{N}_{PA_k}$ is *on* then the inactive source must be *off*.
d. If all inputs are inactive then all active sources must be totally *off*, the inactive source must be totally *on*, and the output must be in the inactive state.
e. When all inputs have reached terminal state the output must eventually reach a valid state.

f. The I_k and O must be either all active-high or all active-low.

g. All the switch elements must be monotone.

then the gate circuit will be well-behaved according to Definition 2.

Proof: We will follow definition 2 paragraph for paragraph:

a. Constraints d and e imply that the output always settles to valid initial and terminal states respectively.

b. We know that the loads presented by the gate inputs I_{PU}, I_{PD}, and I_{PG_k} are always capacitative. We need to show that the load presented by the pass inputs I_{PA_k} is also capacitative. Constraint b implies that the pass inputs are never shorted together nor to any active level. Constraint c implies that pass inputs are never shorted to the inactive level. Thus, the only load seen at the pass inputs is the output capacitor when one of the pass elements is switched *on*.

c. Constraint d implies that when all inputs are inactive, the output is also inactive. Furthermore, the output will be a low impedance source for the inactive level.

d. Constraint e implies that when all inputs have reached their terminal state, the output will be either a valid *high* or *low*. If the output changed to the active state then we conclude that one of the active sources switched *on*. Hence constraints d and g imply that that source remains *on* and provides a low impedance path from the output to the active level.

e. Constraint d states that initially, all active sources are totally *off* and the inactive source is totally *on*. If each input makes at most one transition then we can argue that: constraint g implies that the inactive source can only switch *off* and the active source can only switch *on*; constraints b and f imply that the only effect of an I_{PA_k} transition is to pull the output from inactive to active state; hence the output can only move out of the inactive state if it will eventually settle in the active state.

f. From linear network theory, we know that the conductance of each switch element $\mathcal{N}$ is a fixed function of the state of its gate inputs. The output voltage is a fixed function of the pass input voltages and the conductances of the $\mathcal{N}$. Therefore the output voltage and hence state is a fixed function of the input states.

The constraints given in the theorem above place indirect constraints on the possible initial and terminal states of the input wires. We will now make these implied constraints explicit.

Corollary 5: If a switch level gate meets the constraints in theorem 4 then its inputs must also obey the following constraints:

a. When all inputs are inactive we know that all active sources must be totally *off* and all inactive sources must be totally *on*. This implies that within an active source element, an input driving a switch's gate is active high if and only if the switch is active high; within an inactive source, an input driving a switch's gate is active low if and only if the switch is active high.

b. Since at most one of the active sources can be *on* at any time, the gate inputs to the active sources must be constrained to be mutually exclusive. This is done by driving the gates using the physical wires of a logical signal since at most one of these wires may be active at a time.

c. The gate input(s) to the inactive source must be such that the source is switched *off* before any pass elements are switched *on*. This is the only relative timing constraint in the well-behaved gate form and will be discussed fully in the next section.

We are now ready to apply our general rules to a specific device technology, namely depletion-load NMOS.

3.5 Dual-rail Self-timed Circuits in NMOS with Precharging

Before we start building well-behaved NMOS gates we need to restrict the generalized switch level gate to use the devices available in NMOS. Standard NMOS provides only two of the three simple switch elements, active-high and passive. Also, the active-high switch is ill suited for use as a high-speed pullup.

We now formally introduce the precharge clock which will be used to reduce power dissipation and increase speed. The logical precharge clock signal is a single-rail signal with only two states: Ø and *false*. In NMOS it is physically implemented as a single active-low wire. During the initialization phase it is inactive and during the compute phase it always becomes *false*.

Figure 3-13 shows the most useful forms of NMOS gates. On the left is the basic form for a gate with active-high output. The pass elements are omitted since they are not suitable for coupling low to high transitions at high speeds. The gate consists of a passive pullup and a pulldown element whose inputs are all active-low. The ratio of pullup conductance to minimum pulldown "on" conductance must be such that the output is at a valid logic *low* level in this state.

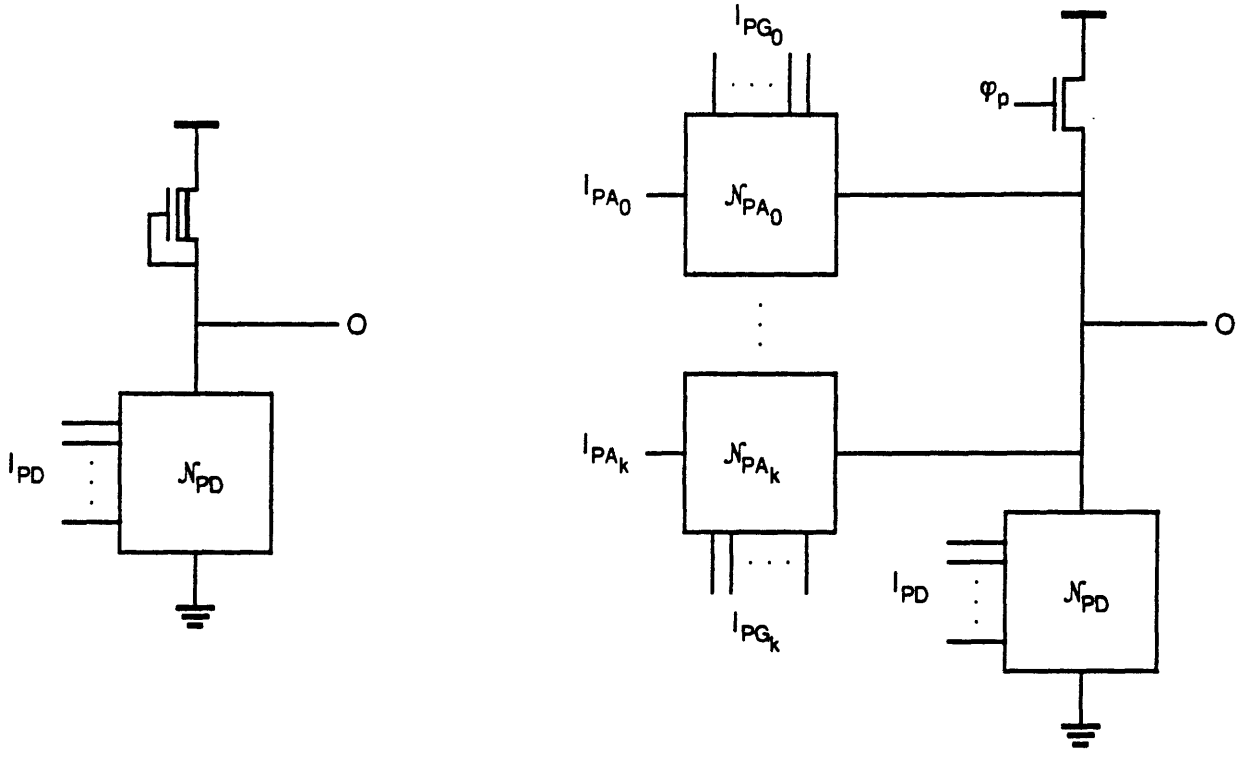


Figure 3-13: Well-behaved NMOS gate forms

The right side of figure 3-13 shows the basic active-low gate. The pullup is a simple active-high switch with the precharge clock (or buffered version) as input. The pulldown and pass elements are composed of active-high switches with active-high inputs. The inputs must be constrained so that either the pulldown is *on* or at most one of the pass elements is *on* at the same time. There are no restrictions on the relative sizes of the switch "on" conductances.

Since the inactive source is *off* during the compute phase we have to make sure that the output inactive level is never degraded by charge sharing between output and parasitic capacitors. Charge sharing can occur if an active source element has internal nodes. There are two remedies: make the total parasitic capacitance much smaller than the output capacitance or precharge the internal nodes in the same way as the output.

Another significant parasitic is the gate to source/drain capacitance in the switches. This kind of capacitance provides direct coupling between input and output nodes. In the pulldown network of

an NMOS gate form, the inputs always have active-high/low encoding opposite of the output encoding. This means that if an inactive to active transition on an input is coupled through to an output which is to remain inactive, the output will only be forced further into the inactive state region. This same reasoning holds for coupling from output to input nodes. Hence coupling through pulldown gate capacitance cannot cause a node state to shift out of the inactive region. The only problem caused by gate parasitic capacitances is in the clocked pullup of the active-low gate. In this case, the high to low transition of the precharge clock is coupled to the precharged node causing its voltage level to drop at the end of the precharge period. This effect is controlled by keeping the gate-source overlap capacitance small compared the output load capacitance. This is not a great problem since the active-low outputs generally drive large loads.

Since the pullup of an active-low gate must be switched *off* before any of its pass elements switch *on* we must constrain the precharge clock to become active before any other network input becomes active. This is the only timing constraint in a well-behaved network of precharged NMOS gates.

These gate forms can be used effectively to take advantage of precharging. If logic modules are built so that all inputs and outputs are active-low then the output driving transistors can be sized to handle any load capacitance without affecting static power dissipation. The active-low outputs are kept internal to the logic modules and thus have lower load capacitances requiring less power dissipation in the passive pullups.

The particular NMOS process used to implement the circuits dictates the sizing constraints on the switches in gates with passive pullups. The process used to implement the floating-point adder is the Bell Laboratories 4 micron NMOS process with modified lambda-based design rules (lambda is 2 microns). This process features two types of enhancement-mode switches, a high-threshold device and a low-threshold intrinsic device, as well as a depletion-mode device. Intrinsic devices are used as clocked pullups thus increasing the logic high level voltage of the active-low gate outputs. Since the inputs to an active-high gate are always active-low, there is always one intrinsic threshold drop in the high level voltage at these inputs. Simple Mead and Conway electrical design rules use a conservative 8:1 pullup to pulldown size ratio. SPICE simulations of the well-behaved gate forms indicate that a pullup to pulldown ratio of 6:1 is satisfactory for the active-high gates.

3.6 Some Well-Behaved Self-Timed NMOS Modules

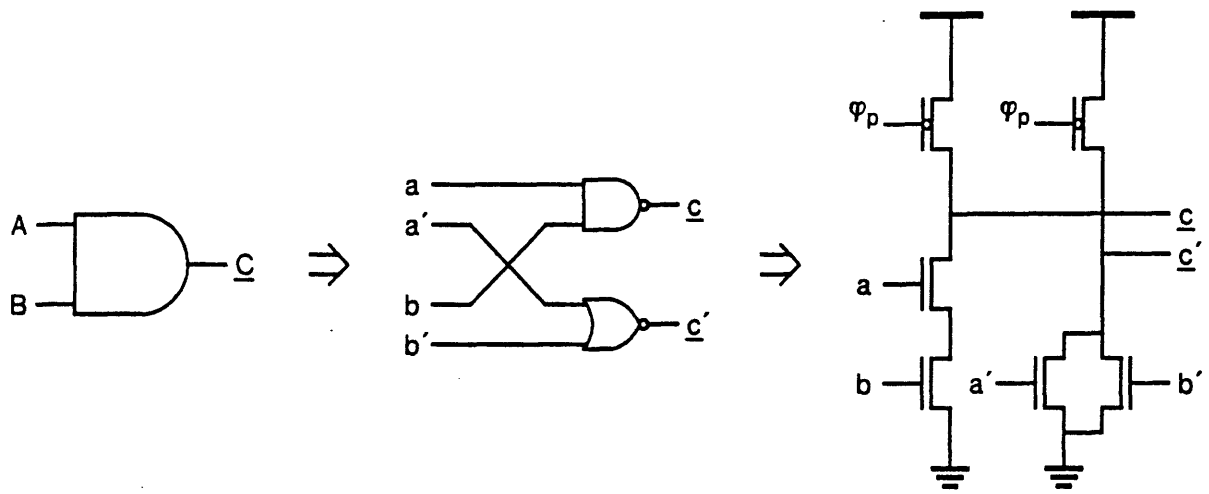


Figure 3-14: NMOS implementation of a dual-rail AND gate

Now we are ready to build NMOS circuit implementations of well-behaved dual-rail networks. Figure 3-14 shows the circuit for the AND gate from figure 3-8. This gate has clocked pullups since it has active-low outputs. This circuit can also be used as an OR gate provided the encodings of input and output pins are inverted and the pullups replaced with passive devices. From the previous section, we know that the gate forms used in the AND gate are well-behaved therefore this implementation of the well-behaved gate level module is well-behaved.

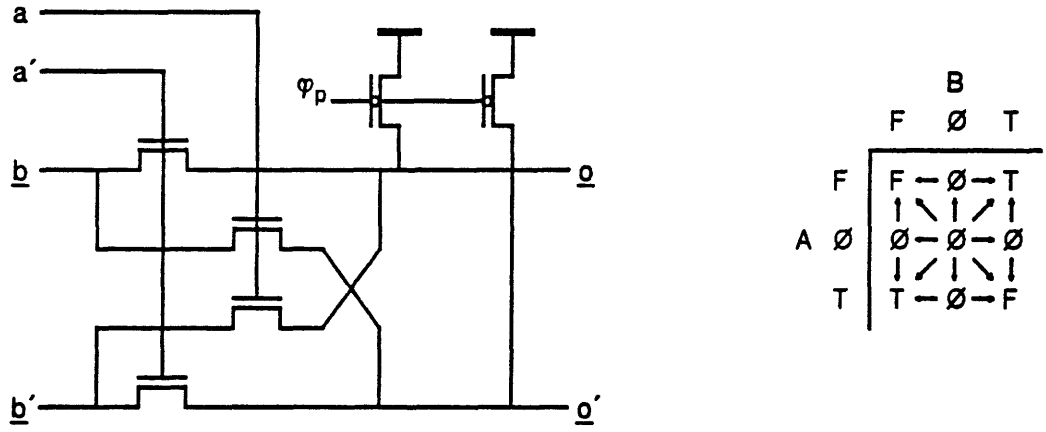


Figure 3-15: NMOS implementation of a dual-rail XOR gate

Figure 3-15 shows how an exclusive-or (XOR) gate is built using precharged pass elements. Note that input A is active high while input B and the output are active low. The gate module is built from two simple well-behaved gate forms. By examining its transition table we can see that it is well-behaved for any sequence of input transitions in contrast to our failed attempt to precharge a conventional XOR gate!

3.7 Verification of Proper Circuit Structure

An important part of the process of building a complex system is verification of proper functionality. When the set of rules that must be followed in the construction process is large it becomes very difficult for human designers to manually verify that the description of the system is well-formed. It is extremely useful to be able to use a computer program to verify that a system is well-formed and well-behaved. This section describes a program used to verify the proper operation of self-timed combinatorial modules as described above.

3.7.1 A Program for Verifying Well-Behaved Networks

The verifier program is actually a form of switch-level logic simulator. Its overall purpose is to verify that a user-described switch network is a well-behaved module.

The program consists of four basic modules:

1. The *loader* module reads a description file of the switch network and builds the internal data structures which describe the network. The description file can be generated from a text description of the network (for initial testing) or by a circuit extraction program from the layout of a circuit (for final testing).
2. The *static evaluator* examines the internal description of the network to make sure that the network is composed entirely of well-behaved switch gate modules and that these modules are interconnected properly (i.e., active-high inputs are always driven by active-high outputs, etc.) The static evaluator also makes sure that there are no input-output loops between gate modules. A side effect of this process is that the network nodes are sorted in order of signal propagation through the network (this is always possible in our loop-free combinatorial design methodology).
3. The *dynamic evaluator* performs a *simulation step* on the network. During a simulation step, all external inputs to the network are set to valid logic levels, then the input state is propagated through the network to find the resulting terminal states of all internal and

output nodes. Since the static evaluator has ordered the network nodes, this process only requires one pass through the node list. The dynamic evaluator checks that information never propagates against the direction of flow determined by the static evaluator in the sorting of the nodes.

4. The *user interface* provides an interactive command interpreter so that the user may control the action of the other three major modules as well as examine the internal information about the structure and state of the network.

The loader is responsible for interpreting the contents of a text file containing a description of a MOS transistor network. The file consists of a set of transistor records. Each record contains the following information:

- Transistor type (enhancement, depletion, intrinsic, or p-channel enhancement).
- Connecting node names (gate, source, and drain nodes).
- Size information (length, width, and area).
- Location (obtained during extraction from layout).

This form of network description file is generated by the NET high-level circuit description language [11] and the EXCL layout extraction program [12] allowing the simulator to verify both high-level network descriptions and the final layouts.

The loader stores the information from the transistor records in a linked list of switch objects in the internal data base. A switch object contains all of the information in its corresponding source file record plus some data fields for linkage with other types of internal data structures.

As the loader constructs the list of switch objects, it also constructs a list of node objects; one node object per distinct node mentioned in a transistor record. The explicit node objects are provided so that the internal algorithms are free to manipulate the network in terms of nodes or switches. The node objects are interlinked with the switch objects so that the network can be freely traversed in any direction starting with any node or switch. The actual linkage works as follows:

- Each node object has three pointers to linked lists of switch objects; one for all switches whose gates connect to this node, one for switch sources, and the last for switch drains.
- Each switch object has pointers to its gate, source, and drain node objects.
- Each switch object also contains a pointer to the next switch whose gate connects to same node as this switch's gate. There are two similar pointers for the source and drain node switch lists.

This structure is based on the node and switch structures used in the RNL switch-level simulator [10NL]. It is useful for many forms of network simulation.

There are two other basic data objects: the *group* and the *class*. The group object is used to group together the set of wire nodes which comprise a logical signal. Each group object points to a node which is a member of a linked ring of nodes in the signal group. The user must declare each signal group; one group object is created per declaration. The group objects are kept in a linked list.

The construction of class objects is the end result of the static evaluation process. From an abstract point of view, a class is a set of nodes and switches which form a well-behaved switch-level physical gate. Since we have defined the well-behaved gate form so that it has a fixed direction of information flow from input to output, we know that a loop-free network of such gates can be sorted so that dynamic evaluation can be performed in a single pass through the list of classes. The task of the static evaluator is to divide the switch network into gate classes and to sort the classes for single-pass evaluation. The class objects are kept in a linked list. Each class object contains two pointers; one to the list of nodes contained in the class, the other to the list of switches in the class. Each switch or node object belongs to one and only one class and contains a pointer to the corresponding class object.

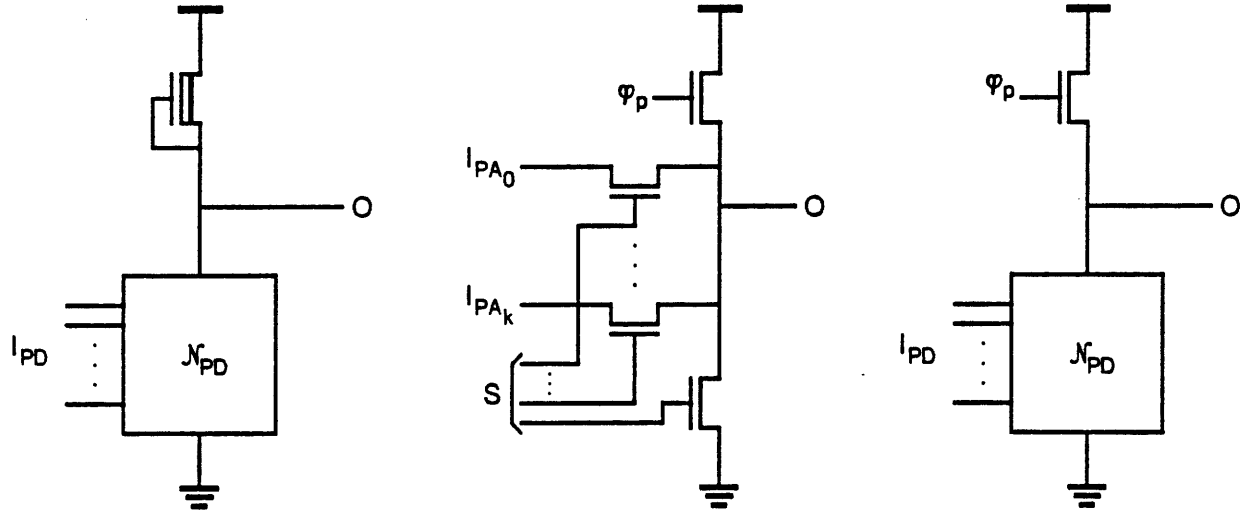


Figure 3-16: Well-behaved NMOS gate forms for simulation

Figure 3-16 shows the basic gate forms recognized by the static evaluator. These follow directly

from the NMOS gate forms from figure 3-13. Note that the active-low gate form with pass inputs is restricted so that each active source has only one simple switch element. The gates of the simple switch elements are driven by the wires of a signal group. This guarantees that no two active sources are ever simultaneously *on* (provided that the gating signal is well-behaved). This simplified form of active-low gate with pass inputs allows the static evaluator to easily distinguish the pass inputs from pass outputs leading to other gate circuits.

Before static evaluation is done, the user must declare all of the network input nodes, output nodes, signal groups, and active-high/low information about the inputs. This information is required by the static evaluator to properly analyze the network. The node and switch data objects each have a flag field containing the user provided information as well as classification information developed by the static evaluator itself.

The static evaluation process is divided into a sequence of steps, the end result of which is a sorted list of class objects. In general, each step is a traversal of one of the internal object lists (node, switch, group, or class). Each step either refines the classification of elements in the network or finds structures which cannot belong to a well-behaved network. When a step finds a questionable structure, it either prints a warning message or an error message. A "warning" message indicates that the structure can be tolerated in a well-behaved network but that the user should examine the network to see why the structure is present. An "error" message indicates that the structure cannot possibly belong to a well-behaved network. In the step descriptions below, the notations "(error)" and "(warning)" are used to indicate the kind of message produced.

The steps are (in sequence):

1. Scan the node list to find obviously incorrect structures. These include: nodes with no connections at all (warning) and nodes which drive gates but have no source, drain, nor input connections (error). Nodes with no connections at all are simply flagged as floating nodes and are subsequently ignored.
2. Scan the switch list to find useless switches. These include: switches with source and drain shorted together (warning) and switches which have source and drain connecting the supply rails together (warning).
3. Classify all nodes either active-high or active-low. This is done by propagating the active-high/low information at the inputs throughout the network. This step reveals errors in signal conformance.
4. Scan node list to find nodes which are not precharged (error).

5. Scan switch list to find switches which connect nodes which are neither both active-high nor both active-low (error). This guarantees that all sets of switch connected nodes are homogeneously active-high or active-low.
6. Scan switch list to classify each switch as either an active source or an inactive source.
7. Verify that all nodes which drive switch gates have both active and inactive sources (error if not).
8. Identify each switch in a pulldown or pullup network as a *primary* source. A switch is a primary source if it is not a pass element.
9. Scan through the switch list to find inactive sources which are not primary sources (i.e. which are pass elements) (error).
10. Scan through the node list to find nodes which have both a primary active source and a primary inactive source. Mark these nodes as gate outputs. If such a node is active-low and has a simple pulldown element, mark all of the node's non-primary sources (i.e. pass elements) which are gated by nodes in the signal group of the pulldown gate node. This mark signifies that the switch is a pass input to an active-low gate. This step finds the outputs of active-high gates and active-low gates with pulldown elements.
11. Rescan the node list finding additional nodes which have no primary active source but have other active sources which can be identified as pass inputs. This step finds active-low gates which have only pass inputs (no pulldown element).
12. Construct a class object for each declared network input node. An input class contains only the input node itself.
13. Scan through the node list: if a node is found which does not belong to any class, construct a new class containing this node, all nodes connected to it through switches, and all of the switches through which connection was traced. Do not trace through a switch which is marked as a pass input. This step and the preceding step yield an unsorted class list.
14. Verify that each class has at least one connection to each of the power supply rails (error if not).
15. Assign a *depth* number to each class. Input classes all have depth 0. The depth number represents the maximum distance of a class from the inputs in terms of gate levels. (The class depth number applies to all nodes and switches in the class.) Sort the classes using the following algorithm (starting with depth 1): scan through the unsorted class list and collect all classes which have all gate and pass input nodes belonging to sorted classes; append the collected classes to the sorted class list and assign them the current depth count; increment the depth count and repeat the process.
16. Verify that all classes are removed from the unsorted class list. If any classes are left,

then there is at least one loop in the inter-class connections. In this case, examine the remaining classes carefully to find each node and switch which is part of a loop.

The dynamic evaluator uses the following algorithm to propagate input wire states through the sorted list of classes:

- Assign a state to each input wire.
- Scan once through class list. For each class we know that all nodes which are inputs to the current class are now stable since their classes have already been evaluated. Examine the open and closed switches within the class to determine which nodes are high and which are low. Assign a node an illegal or "X" state if it attempts to connect a pass input node to a supply rail. Also assign the illegal state to nodes when there is a path through active switches between the supply rails and when charge-sharing occurs between node capacitances with different states. The *on* resistance of an enhancement or intrinsic switch is taken to be zero and the *on* resistance of a depletion pullup is taken to be one.

The primitive dynamic evaluator described above is not directly invoked by the user. The user controls dynamic evaluation via three commands, **precharge**, **compute**, and **scan**. The **precharge** and **compute** commands are the most primitive of these commands. Before they are invoked, the user must specify the terminal state of each of the input signal groups. The *precharge* command performs the following:

- Set all inputs (including the precharge clock) to the inactive state.
- Invoke the dynamic evaluator to find the initial state of the entire network.
- Set the precharge clock to active (*false*) state.
- Invoke the dynamic evaluator again to propagate any additional changes in the network.
- Examine all precharged nodes and report any which are not in a valid inactive state.

The **compute** command is used to find the terminal state of the network after the **precharge** command has initialized the network to a valid state. The **compute** command performs the following:

- Set all inputs to the given terminal states.
- Invoke the dynamic evaluator to propagate the input changes.
- Examine all signal groups and report any which are not in a valid terminal state.

The **scan** command is the most powerful command in the simulator. It is a high-level command capable of running a complete dynamic evaluation of the given network. When invoked, it finds all

input signal which have not already been assigned a terminal state by the user; then it runs one precharge/compute cycle for each possible valid combination of input signal terminal states, printing the terminal states of the inputs and outputs after each cycle. If the user leaves all the inputs initially undefined, then the **scan** command will verify that the circuit is well-behaved for all possible combinations of input states.

Since the static evaluator cannot find all features of the circuit which would cause it not to be well-behaved, the circuit is known to be well-behaved only after a full scan of all input combinations has been made (and no errors found). Although the dynamic evaluator is much faster (being single-pass) than a general-purpose switch-level simulator, the number of input combinations which must be checked increases exponentially in the number of inputs making it impractical to do a full scan of a very large network. We can greatly reduce the amount of time needed to verify a large network by using a modular approach to assembly and testing. If we design the network so that it is composed of a small set of (allegedly) well-behaved modules, then we can verify that the network is well-behaved by first checking that each module is completely well-behaved by itself then checking that all of the module interconnection rules have been obeyed. Together, the static and dynamic evaluators can check the modules for well-behavedness. If the network is composed entirely of verified modules, then only static evaluation is needed to verify that the network is well-behaved since the static evaluator can find all violations of the interconnection rules. Using this technique, the dynamic evaluator only needs to be applied to the entire network to verify that the logical function of the network is correct. This can usually be done using a small set of test cases for input values.

3.7.2 Simulation Examples

Several sample simulation sessions are presented below. They are all based on the well-behaved circuit shown in figure 3-17. The logical gate level representation of the circuit is shown in the lower right corner of the figure. The circuits inside dashed lines are not part of the well-behaved circuit but are used in later examples. Dotted lines are used to outline the switch gate classes.

Figure 3-17 shows the print-out of the first session which is a test of the well-behaved circuit. User typed letters are in bold face, computer typed letters in light face, and comments in italics. In this example, no errors are found and the truth table generated by the **scan** command shows that the logical function $\underline{D} = C \oplus AB$ is obeyed. The set of declaration command lines were contained in a file called "test.csim". This file was automatically taken as command input after the network file "test.sim" was loaded.

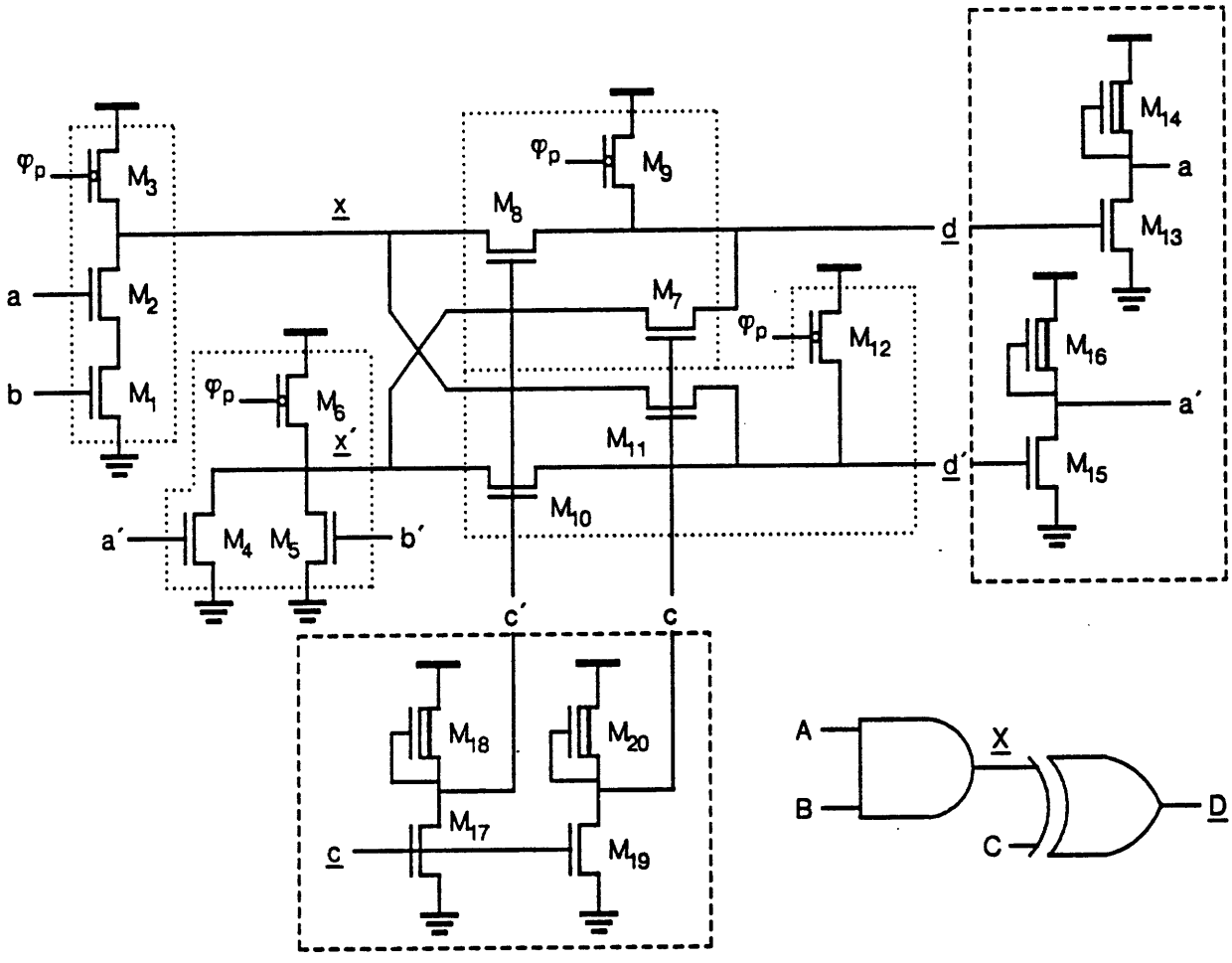


Figure 3-17: Simulator test circuit

Figure 3-18 shows how the simulator reports signal interconnect non-conformance. In this example, the logical input signals to the XOR gate have been reversed so that X connects to the active-high input and C connects to the active-low input. The static evaluator found a conflict at each of the four pass elements in the XOR gate where the user attempted to join the active-low D signal to the active-high C signal illegally. Note that two errors were reported for each switch. The first group of errors was detected during the propagation of the input active-high/low information through the circuit. The second group of errors was detected during examination of all switches after propagation was complete. Both kinds of error are reported since they may occur independently.

```
csim test
Loading "test.sim"...
Network contains: 14 nodes,
        4 intrinsic switches,
        8 enhancement switches,
        0 depletion switches,
        0 p-channel switches,
        0 pullup switches.
(13 lines read)
CSIM>dec PRE VP
CSIM>dec IN A A'
CSIM>dec HI A A'
CSIM>dec GROUP A A'
CSIM>dec IN B B'
CSIM>dec HI B B'
CSIM>dec GROUP B B'
CSIM>dec IN C C'
CSIM>dec HI C C'
CSIM>dec GROUP C C'
CSIM>dec OUT D D'
CSIM>dec GROUP D D'
CSIM>dec GROUP X X'
CSIM>[EOF]
CSIM>verify
Verify succeeded - 11 classes - maximum depth 2.
CSIM>scan
A B C | D
1 1 1 | 0
0 1 1 | 1
1 0 1 | 1
0 0 1 | 1
1 1 0 | 1
0 1 0 | 0
1 0 0 | 0
0 0 0 | 0
CSIM>exit
```

Figure 3-18: Simulation of a well-behaved circuit

Figure 3-19 illustrates the detection of a missing pullup switch. The only change made to the test circuit is to remove M_9.

Figure 3-20 illustrates the loop finding capability of the static evaluator. Transistors $M_{13\text{-}15}$ were added to the basic circuit in figure 3-17 to form a feedback loop from output $\underline{D}$ to input A. On completion of the class sorting algorithm, the static verifier found that there were still classes unsorted. It then examined the unsorted classes carefully to find every node or switch which was a member of an inter-class loop.

The final examples illustrate departure from well-behavedness which the static evaluator fails to

```
csim test1
Loading "test1.sim"...
Network contains: 14 nodes,
        4 intrinsic switches,
        8 enhancement switches,
        0 depletion switches,
        0 p-channel switches,
        0 pullup switches.
(13 lines read)
    Declaration print-out omitted for brevity
CSIM>[EOF]
CSIM>verify
    The node information is: name<flags>state
    flags are: In, input; Out, output; Hi, active-high; Lo, active-low
Error: Active-high/low conflict at node D'<OutLo>=U via switch
  enhancement g:[X<Lo>=U] s:[C<InHi>=U] d:[D'<OutLo>=U]!
Error: Active-high/low conflict at node D<OutLo>=U via switch
  enhancement g:[X<Lo>=U] s:[C'<InHi>=U] d:[D<OutLo>=U]!
Error: Active-high/low conflict at node D'<OutLo>=U via switch
  enhancement g:[X'<Lo>=U] s:[C'<InHi>=U] d:[D'<OutLo>=U]!
Error: Active-high/low conflict at node D<OutLo>=U via switch
  enhancement g:[X'<Lo>=U] s:[C<InHi>=U] d:[D<OutLo>=U]!
Error: Switch
  enhancement g:[X<Lo>=U] s:[C<InHi>=U] d:[D'<OutLo>=U]
    connects non-conformable nodes!
Error: Switch
  enhancement g:[X<Lo>=U] s:[C'<InHi>=U] d:[D<OutLo>=U]
    connects non-conformable nodes!
Error: Switch
  enhancement g:[X'<Lo>=U] s:[C'<InHi>=U] d:[D'<OutLo>=U]
    connects non-conformable nodes!
Error: Switch
  enhancement g:[X'<Lo>=U] s:[C<InHi>=U] d:[D<OutLo>=U]
    connects non-conformable nodes!
Error: Verify failed!
CSIM>exit
```

Figure 3-19: Simulation of a circuit with non-conforming signals

detect. Figure 3-21 demonstrates the detection of "backward" information flow in the circuit during dynamic evaluation. Transistors M_{17-20} in figure 3-17 were added to provide a non-well-behaved signal input to C. Since the static evaluator assumed that the C signal group would be well-behaved, it could not detect that information could flow in both directions through the C gated switches. The dynamic evaluator found that when the $\underline{c}$ input switched to terminal state, the $\underline{D}$ classes placed a static load on the $\underline{X}$ outputs (by shorting them together!)

Figure 3-22 illustrates the detection of a simple functional defect in the test circuit. In this circuit, M_4 and M_5 were placed in series instead of parallel. All of the physical gates are well-behaved but one of the logical gates is not. The dynamic evaluator found that the two gate output signals fail to reach terminal state for some combinations of input terminal states.

```
csim test2
Loading "test2.sim"...
Network contains: 14 nodes,
        3 intrinsic switches,
        8 enhancement switches,
        0 depletion switches,
        0 p-channel switches,
        0 pullup switches.
(13 lines read)
```
declaration print-out omitted for brevity
```
CSIM>[EOF]
CSIM>verify
Error: Class 11 has no inactive source!
Class 11 (depth 2):
  Nodes:
D<OutLo>=U
  Switches:
enhancement g:[C'<InHi>=U] s:[X<Lo>=U] d:[D<OutLo>=U]
enhancement g:[C<InHi>=U] s:[X'<Lo>=U] d:[D<OutLo>=U]
.
Error: Verify failed!
CSIM>exit
```

Figure 3-20: Simulation of a circuit with missing pullup

```
csim test3
Loading "test3.sim"...
Network contains: 14 nodes,
        4 intrinsic switches,
        10 enhancement switches,
        0 depletion switches,
        0 p-channel switches,
        2 pullup switches.
(17 lines read)
    declaration print-out omitted for brevity
CSIM>[EOF]
CSIM>verify
Error: Loop(s) detected between classes!
Loop members:
  Class 11 node D'<OutLo>=U.
  Class 10 node D<OutLo>=U.
  Class 9 node X'<Lo>=U.
  Class 8 node A'<Hi>=U.
  Class 7 node X<Lo>=U.
  Class 7 node $1<Lo>=U.
  Class 6 node A<Hi>=U.
  Class 8 switch
    enhancement g:[D'<OutLo>=U] s:[A'<Hi>=U] d:[Gnd<InNor>=L(L)].
  Class 6 switch
    enhancement g:[D<OutLo>=U] s:[A<Hi>=U] d:[Gnd<InNor>=L(L)].
  Class 11 switch
    enhancement g:[C<InHi>=U] s:[X<Lo>=U] d:[D'<OutLo>=U].
  Class 10 switch
    enhancement g:[C<InHi>=U] s:[X'<Lo>=U] d:[D<OutLo>=U].
  Class 11 switch
    enhancement g:[C'<InHi>=U] s:[X'<Lo>=U] d:[D'<OutLo>=U].
  Class 10 switch
    enhancement g:[C'<InHi>=U] s:[X<Lo>=U] d:[D<OutLo>=U].
  Class 9 switch
    enhancement g:[A'<Hi>=U] s:[X'<Lo>=U] d:[Gnd<InNor>=L(L)].
  Class 7 switch
    enhancement g:[B<InHi>=U] s:[$1<Lo>=U] d:[X<Lo>=U].
  Class 7 switch
    enhancement g:[A<Hi>=U] s:[$1<Lo>=U] d:[Gnd<InNor>=L(L)].
Error: Verify failed!
CSIM>exit
```

Figure 3-21: Simulation of a circuit with loops

```
csim test4
Loading "test4.sim"...
Network contains: 15 nodes,
        4 intrinsic switches,
        10 enhancement switches,
        0 depletion switches,
        0 p-channel switches,
        2 pullup switches.
(17 lines read)
CSIM>dec PRE VP
CSIM>dec IN A A'
CSIM>dec HI A A'
CSIM>dec GROUP A A'
CSIM>dec IN B B'
CSIM>dec HI B B'
CSIM>dec GROUP B B'
CSIM>dec IN C
CSIM>dec LO C
CSIM>dec GROUP C
CSIM>dec GROUP C C'
CSIM>dec OUT D D'
CSIM>dec GROUP D D'
CSIM>dec GROUP X X'
CSIM>[EOF]
CSIM>verify
Verify succeeded - 12 classes - maximum depth 2.
CSIM>set hi a b
CSIM>scan
C | D
Error: Transition propagates backward to node X<Lo>=L(U) via switch
  enhancement g:[C'<Hi>=H(U)] s:[X<Lo>=L(U)] d:[D<OutLo>=H(U)]!
Error: Transition propagates backward to node X'<Lo>=H(U) via switch
  enhancement g:[C'<Hi>=H(U)] s:[X'<Lo>=H(U)] d:[D'<OutLo>=H(U)]!
Error: Group {X<Lo>=x(U) X'<Lo>=x(U)}
    has illegal signal values (2/2)!
Error: Group {D<OutLo>=x(U) D'<OutLo>=x(U)}
    has illegal signal values (2/2)!
Error: Group {C<Hi>=H(U) C'<Hi>=H(U)}
    has too many active signals (2/2)!
0  | x
CSIM>exit
```

Figure 3-22: Simulation of a circuit with "backward" signal propagation

```
csim test5
Loading "test5.sim"...
Network contains: 15 nodes,
        4 intrinsic switches,
        8 enhancement switches,
        0 depletion switches,
        0 p-channel switches,
        0 pullup switches.
(13 lines read)
    declaration print-out omitted for brevity
CSIM>[EOF]
CSIM>verify
Verify succeeded - 11 classes - maximum depth 2.
CSIM>scan
A B C | D
1 1 1 | 0
Error: Group {X<Lo>=H X'<Lo>=H} is undefined!
Error: Group {D<OutLo>=H D'<OutLo>=H} is undefined!
0 1 1 | U
Error: Group {X<Lo>=H X'<Lo>=H} is undefined!
Error: Group {D<OutLo>=H D'<OutLo>=H} is undefined!
1 0 1 | U
0 0 1 | 1
1 1 0 | 1
Error: Group {X<Lo>=H X'<Lo>=H} is undefined!
Error: Group {D<OutLo>=H D'<OutLo>=H} is undefined!
0 1 0 | U
Error: Group {X<Lo>=H X'<Lo>=H} is undefined!
Error: Group {D<OutLo>=H D'<OutLo>=H} is undefined!
1 0 0 | U
0 0 0 | 0
CSIM>exit
```

Figure 3-23: Simulation of a circuit with a functional defect

CHAPTER FOUR

Implementation of the Adder Module

In this chapter we unite the topics discussed so far in the design of a procedure for building a floating-point adder with arbitrary exponent and fraction size.

The first step in this process is to determine the exact requirements for the function blocks from which the adder is to be built. We start from the approximate algorithm in figure 2-1 and derive the exact functional specification for each major block following the data flow through the unit.

4.1 Functional Specification of Adder Logic

Figure 4-1 shows the detailed structure of the floating-point addition algorithm. The inputs are the operands A and B each of which consists of an N_e-bit exponent, A_e, B_e; a sign bit, A_s, B_s; and an N_f-bit fraction, A_f, B_f.

The detailed sequence of operations is as follows:

1. If A_e is non-zero then append a 1 above the MSB of A_f otherwise append a 0. Do the same for B_e and B_f. This operation recovers the "hidden" bit in the normalized numbers. The fractions now have $N_f + 1$ bits each.
2. If B_e is larger than A_e then swap corresponding bits of each operand otherwise pass the operands through unchanged. This yields a new pair of operands A' and B' where A'_e is never less than B'_e.
3. Subtract B'_e from A'_e to get a positive offset D_e.
4. Shift B'_f right by D_e bits to get B''_f. Note that B''_f has $N_f + 3$ bits (two more than B'_f). The extra bits are the two least significant bits of B''_f. The most significant of these two bits is called the *guard* bit, The LSB of B''_f is called the "sticky" bit and is the logical inclusive-or of all bits of B'_f which get shifted to the right of the guard bit. The "sticky" bit is required to allow the arithmetic to behave as if carried out to infinite precision before rounding or truncation.

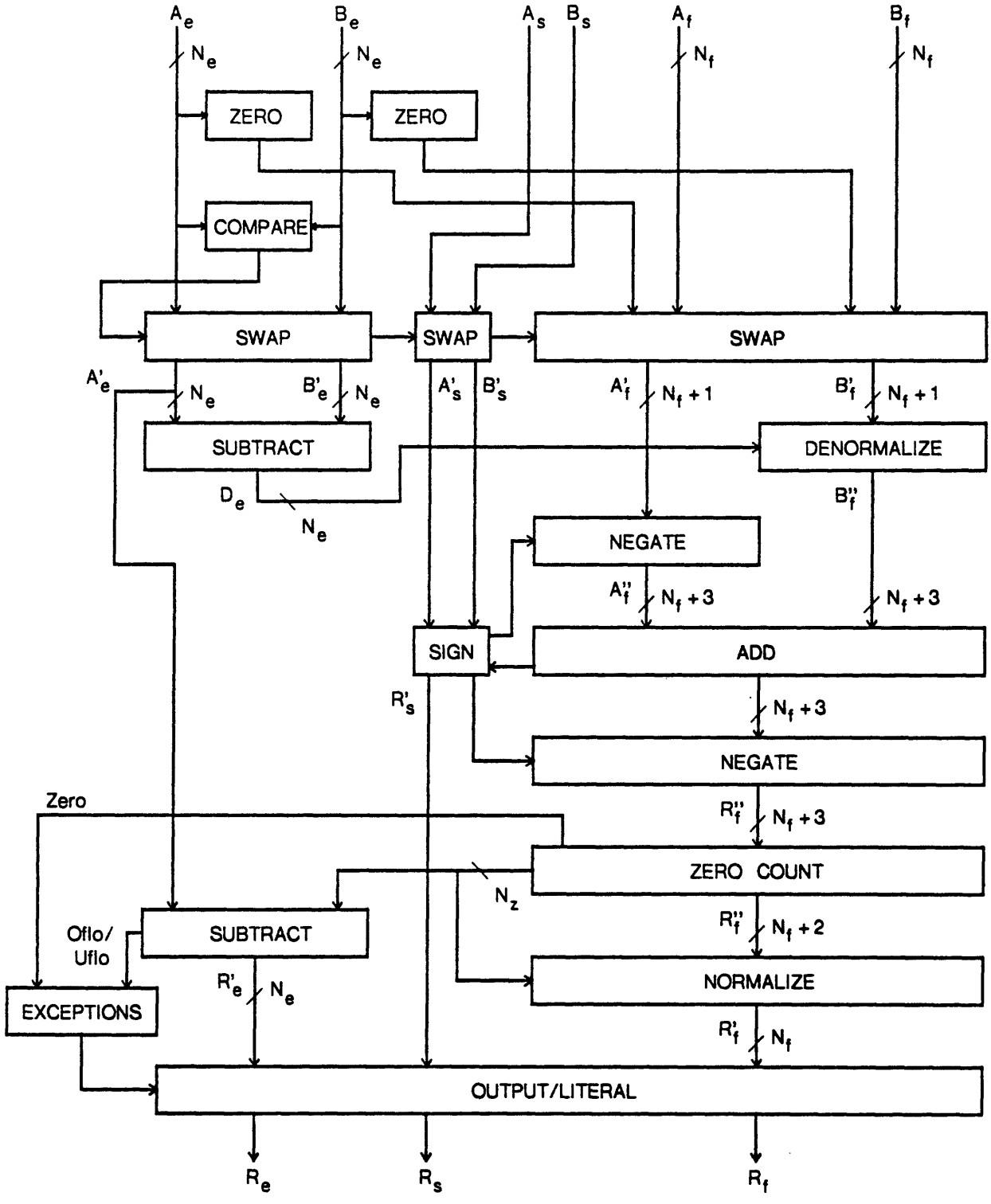


Figure 4-1: Floating-point addition algorithm

5. Append two 0 bits to A'_f, then add or subtract A'_f and B''_f. The raw result has $N_f + 4$ bits to allow for overflow from the MSB.
6. If the raw result is negative then take its absolute value to obtain the result magnitude R''_f. For unbiased truncation, the "sticky" bit need only participate in the addition/subtraction itself. Thus R''_f has $N_f + 3$ bits, including the overflow bit in the most significant position.
7. The value of R''_f is in the range $0 \leq R''_f < 4$. Shift R''_f (left or right) so that the value of R'_f is in the range $1 \leq R'_f < 2$. Add the number of positions shifted right (which may be negative) to A'_e to obtain the result exponent R'_e. The actual number of bits required to encode the shift count is not necessarily equal to N_e. The total number of bits required is the smallest integer N_z such that $2^{N_z} > N_f + 3$.
8. If R''_f is zero, R'_e is zero, or the exponent adjustment underflowed then produce a zero result. If the exponent adjustment overflowed, then produce a result which has exponent and fraction set to all ones (maximum magnitude) but retains the sign of the actual result. If there are no exceptional conditions then pass the values R'_e, R'_s, and R'_f through as the final result R_e, R_s, R_f.

4.2 Guidelines for Procedural Design

Now that we have a precisely specified algorithm for doing floating-point addition, we need to map this algorithm into a VLSI layout. This mapping process is complicated by the need for selectable data bus widths for both the fraction and exponent computations. To make the task manageable, we need some general guidelines for doing this kind of mapping.

First we need to establish some general goals for this task. The task itself is to write a program for building an arithmetic unit given such parameters as exponent width, fraction width, and round-off mode. Since the program will be able to generate a large number of different layouts, we need to design the program so that we can verify that it works correctly in all cases without actually having to build each possible layout.

A set of general guidelines for achieving the goals above is:

1. Minimize the number of kinds of basic cell needed.
2. Minimize the dependence of cell size on the design parameters. Ideally, only a cell which drives a set of data bus cells in parallel needs to have adjustable size. This minimizes the need to recheck the correctness of the cell as the parameters change.
3. Minimize the dependence of function block size and topology on the design parameters.

The general goal is for the function block to be composed of a set of N identical cells (for a bus of width N) where the size of the cell is independent of the parameter, N. In this case we can prove correctness for any value of N using induction on two or three test cases.

4. Minimize the complexity of interconnect between function blocks.
5. Design function blocks which have common data busses so that they can be directly abutted, i.e., equalize cell pitch. This allows a correctness check by abutting a small subset of cells from the parent function blocks.
6. Most function blocks require control signals which are fed in common to each data cell in the block. A cell used for driving the control inputs to a function block which needs to have adjustable driver size should have variable width and *fixed* height (assuming that the block is oriented with bits along a horizontal line.) This makes the correctness of the connection between the block and the driver independent of the block's width.
7. Data bit cells for function blocks should be organized so that data input/output pins are along the top and bottom edges and control pins are along the side edge (the orientation can be rotated 90 degrees if appropriate.) Power and ground pins can be oriented with either the control or data signals. This cell organization must be consistent among all cells in an abutted set of function blocks.

We start decomposing the adder flow graph in figure 4-1 into layout by observing that the graph naturally falls into two major sections: elements which have N_f-bit data paths (fraction section) and elements which have only N_e-bit data paths (exponent section.) The function blocks which connect to both sections (e.g. denormalizer, normalizer) naturally belong to the fraction section with their exponent busses considered as control signals.

Figure 4-2 shows a rough floor plan of the layout. The layout is now divided into the exponent section whose size depends only on N_e and the fraction section whose size depends primarily on N_f. This topology will allow most of the interconnections between function blocks to be made by directly abutting the blocks.

The next step is to decompose the function blocks into a small number of basic cells.

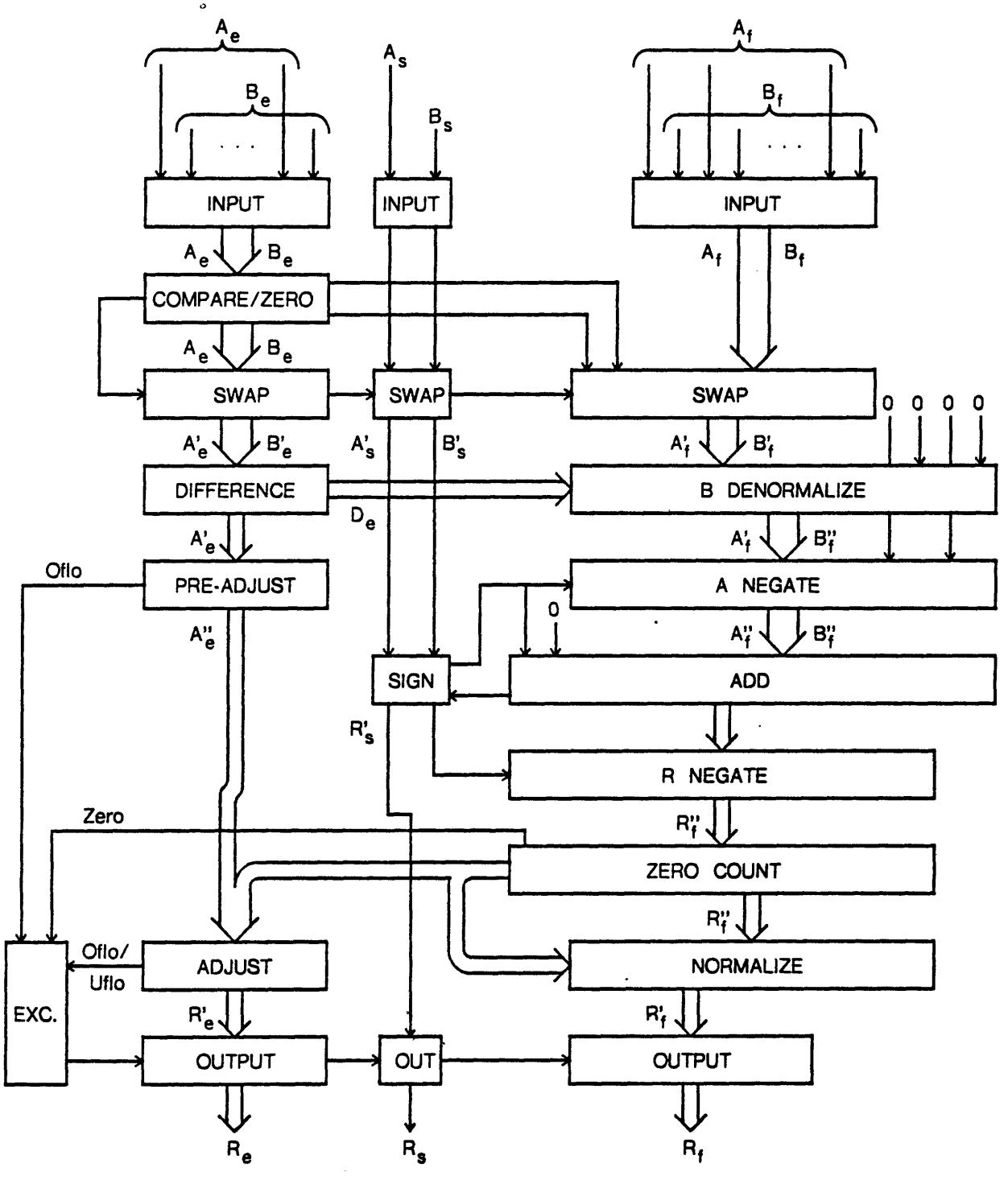


Figure 4-2: Initial floor plan of adder layout

4.3 Specification of Function Block Forms

Now we must specify the basic topology for each of the different kinds of function blocks. Figure 4-2 shows the adder floor plan described below. As we work out the overall structure of each block we need to keep in mind that the relative ordering of the bits (i.e. MSB -> LSB or LSB -> MSB) of the exponent and fraction paths is yet to be determined.

The *input buffer* is straightforward to implement. If we pair together like signal lines of the two input operands then we find that the input buffer should be built using a basic cell which converts two conventional logic signals into two dual-rail logic signals (two pairs of wires). An N bit input buffer is constructed by simply abutting N input cells.

The *exponent comparator* is more difficult to build using a single replicated cell type. It is a basic example of a function block which requires communication between bit positions and is sensitive to the ordering (significance) of the input bit vector. For this kind of function block we would like to use a basic cell which only needs to communicate with its nearest neighbors and is usable in any bit position. We can specify the function of this cell by using a form of induction: Assume that we can build a module that compares two N-bit words A and B and produces a result signal with states {A<B, A>B, A = B}. To build an N + 1-bit comparator, we need a cell which performs the following function in the most significant position of the enlarged comparator: Examine the MSB input data; if bit B is greater than bit A then signal A<B otherwise if A is greater then signal A>B otherwise duplicate the output of the original N-bit comparator to get the comparison result.

For an N-bit comparator we use N of the cells above where the output of one cell is connected to the cascade input of the next more significant bit. If the cascade input of the least significant bit cell is connected to indicate equality then the LSB cell constitutes a 1-bit comparator. By induction on the definition of the cell behavior, we can see that the output from the MSB cell is the comparison result for the entire word.

The exponent zero detect function can be performed by a NOR gate. We will incorporate two of these gates into the exponent comparator as will be seen when the actual circuitry is shown in a later section.

The *operand swap* is similar in form to the input buffer. An N-bit swap unit can be built using N identical cells each of which swaps corresponding bits in the two operands (four wires). The swap block also requires a driver cell which drives the select inputs of each of the swap cells in parallel.

The *exponent difference* block is the first kind of adder unit encountered. There are many different common forms for adder circuits. For this design we prefer regular linear structures rather than tree-like structures. Since the exponent path has only a few bits we can justify using a simple full adder cell in each bit position without the topological complications of a carry lookahead circuit. We can build an N-bit adder using N full adder cells connecting the carry output of one cell to the carry input of the next more significant cell. To subtract the two exponents, we complement the smaller operand (by swapping its dual-rail wire pair) and set the carry input of the least significant bit to 1 thus performing a 2's complement subtraction (with guaranteed positive result).

Designing the *denormalization* block poses a new kind of problem: A combinatorial implementation of this block is bound to vary in both dimensions; width depending on fraction size and height depending on exponent size. The best topological form we could hope for is to build the denormalizer using a N_e by N_f array of identical cells. The so called *barrel shifter* is an example of a shifting block which can be built from an array of identical cells. Unfortunately, it would require 2^{N_e} rows of cells since each row only shifts by one bit position; it would also require a decoder to convert from the N_e-bit shift count to the 2^{N_e} row control lines. By sacrificing some regularity we can build a modified form of barrel shifter where each row shifts by a number of bits which is a power of two. If we provide one row per bit of the exponent difference and match that row's shift amount to the binary significance of its controlling bit then we eliminate the need for a decoder and reduce the height to linear dependence on N_e. We can build this kind of shifter from an array of identical switch cells each of which selects either an unshifted or shifted bit from the previous row. N_f of these cells can be abutted horizontally to form a single row but the vertical connection will vary with row significance and fraction width. This is the first block where we will need the services of a routing procedure to interconnect elements.

We now arrive at a key element of the adder unit: the *fraction adder/subtractor*. In order to simplify the construction of the *result negation* block which follows this block we will use 1's complement arithmetic. This allows negation to be done by bitwise complementing (wire-swapping) of the operand or result. To build the adder/subtractor with result negation we take the basic full adder cell and add two negation cells; one for negating one of the operands, the other for negating the result. Since the B operand to the adder has been delayed by passing through the denormalizer, we will want to perform the input negation on the A operand to avoid causing further delays.

To get proper 1's complement arithmetic we must connect the carry output from the MSB of the

adder to the carry input of the LSB. This is in violation of the interconnect rules we derived in the previous chapter! However, as we will show later, it is still possible to design the 1's complement adder so that the system as a whole is well-behaved.

Using 1's complement arithmetic, the *result negation* block is simple. The negation cell is a multiplexer which selects whether or not to swap the wires of its input bit.

The *normalization* block can be divided into two self-contained sections: *leading zero count* and *normalize shift*. The leading zero count section produces a binary coded count of the number of leading zeros in the result. This result is fed to the exponent adjust block and to the normalize shift block. The shift block shifts the result left by the specified amount causing the most significant 1 in the result to appear in the MSB position.

Following our tendency toward using linear structures we can build the zero count unit using a row of cells each of which performs the following function: if all of the more significant bits are 0 then either a) if the current bit is a 1 then output the position of this bit (the MSB has position zero) or b) if the current bit is 0 then signal the next less significant bit cell that only leading zeros have been seen so far. Each of these cells is identical except for the encoding used to indicate the significance position of the cell. The cascade output from the LSB serves as a zero result detect signal for exception handling. The complete form of these cells is discussed in a later section.

The normalization shifter can be built using the same form as the denormalization shifter. The only major difference is in the direction of shift.

The *exponent adjust* block is basically a subtractor. It can be implemented using the same form as the exponent difference block above. Note that since the normalizer must be able to accommodate an overflow of one bit in the adder, a leading zero (and left shift) count of zero corresponds to an exponent adjustment of +1. To compensate for this offset we will need an *exponent pre-adjust* block to add 1 to the exponent. For the pre-adjust block, we need an adder block with one operand constant. There is no speed penalty for the pre-adjustment since this is done in parallel with several lengthy fraction operations.

The *output/literal* block is another simple block where there is no communication between adjacent cells. The basic cell converts a dual-rail result bit to a single conventional logic wire. On overflow or underflow the cell outputs a constant 1 or 0 respectively.

The only remaining function units are two simple random-logic blocks: the *sign handling* block which directs the fraction negation blocks and computes the true sign of the result and the *exception handling* block which detects overflow, underflow, and zero result and controls the output block.

4.4 Global Interconnect Topology in the Adder

Note that the interconnections between two major sections in figure 4-2 are only shown roughly. The next step of the design process is to decide how to order bit significance in the various blocks to make interconnection as easy as possible.

Obviously, the horizontal ordering of bits in the fraction section must be the same for all blocks so that they may be abutted vertically. Similarly, all the exponent blocks must share the same horizontal ordering. In all, there are five orderings to be chosen: exponent section, fraction section, denormalizer count input, zero count output, and normalizer count input. Note that the sign bit circuitry naturally falls next to the MSB of the fraction section since it is only at the MSB of the fraction adder that the sign bit interacts closely with other circuitry.

The orderings can be resolved as follows:

- The major constraint on the ordering of the fraction section is that the zero operand detect signals from the exponent section must provide the hidden MSB bit. Thus the fraction MSB should face the exponent section.
- Since the exponent difference block will produce the LSB result first, the rows of the denormalizer should be oriented so that the LSB row is uppermost so that fraction data can start to propagate through it as soon as possible.
- Since the exponent difference should be wired to the denormalizer as simply as possible, the exponent section should be oriented with LSB inward.
- Since the wiring between the exponent section and the normalizer sections cannot be done on a single layer there is no clear choice of orientation. The only obvious choice is to make the orientation of the normalizer opposite to the orientation of the zero count output. We choose to make the LSB row of the normalizing shifter uppermost making it similar to the denormalizing shifter. This means that the zero count LSB is lower most.

4.5 Basic Adder Cells

Now that we have specified the form of the function blocks down to the cell level, we need to devise a minimal set of basic cells from which these functions can be easily assembled.

As we design these basic cells we must keep in mind that data bit cells must have the same pitch so that function blocks can be directly abutted. To begin designing the individual cells, we need to make a good guess at the optimal standard pitch. A good way to start the process is to do trial layouts of the most complex cell. This cell usually sets the minimum possible pitch. The rest of the cells should be designed to match the pitch of this *guide* cell. The full adder cell is the most complex of the basic cells and was used as the guide cell in the original design process. The initial guide pitch was 50 lambda units (100 microns). It turned out that all the other cells could be built to fit in this width thus it was the correct choice.

Since most of the data bit cells interact closely with each other we need to set a standard for the orientation of power supply, control, and data wiring. Since our NMOS technology has only two global wiring layers, metal and polysilicon, we need to select the routing style carefully. We note that generally the vertical wiring runs in the data path are much shorter than the horizontal wiring runs distributing parallel control signals to the data bit cells. This difference gets more pronounced as the data path widths are increased. Since poly has a much higher resistivity than metal we want to make the longer runs in metal. Because metal is the only low resistivity conductor available, we must run all power supply lines in metal. Hence each data cell will be designed with power and control lines running horizontally in continuous strips of metal and data lines running vertically between cells in polysilicon.

It is interesting to consider what effects the availability of two metal layers would have on the choice of wiring configuration. In a two layer metal process, the first layer of metal has the same characteristics as the metal layer in a single-level process. The second layer of metal generally has a larger minimum pitch than the first, hence lower wiring density. One way to take advantage of the second layer would be to use it for the low-density vertical bus wiring instead of the poly layer. This would eliminate space consuming poly wiring from many of the basic cells to achieve a smaller horizontal cell pitch. Cell height could be reduced by running the power connections in second metal along with the vertical bus wires.

4.5.1 Input Buffer Cell

The input buffer cell is composed of two identical sections, one for each input bit. Each section converts the high-true conventional logic input into a pair of active-low dual-rail wires. Figure 4-3 shows the gate and transistor level circuits for the A operand section of the input buffer. The B operand circuits are identical.

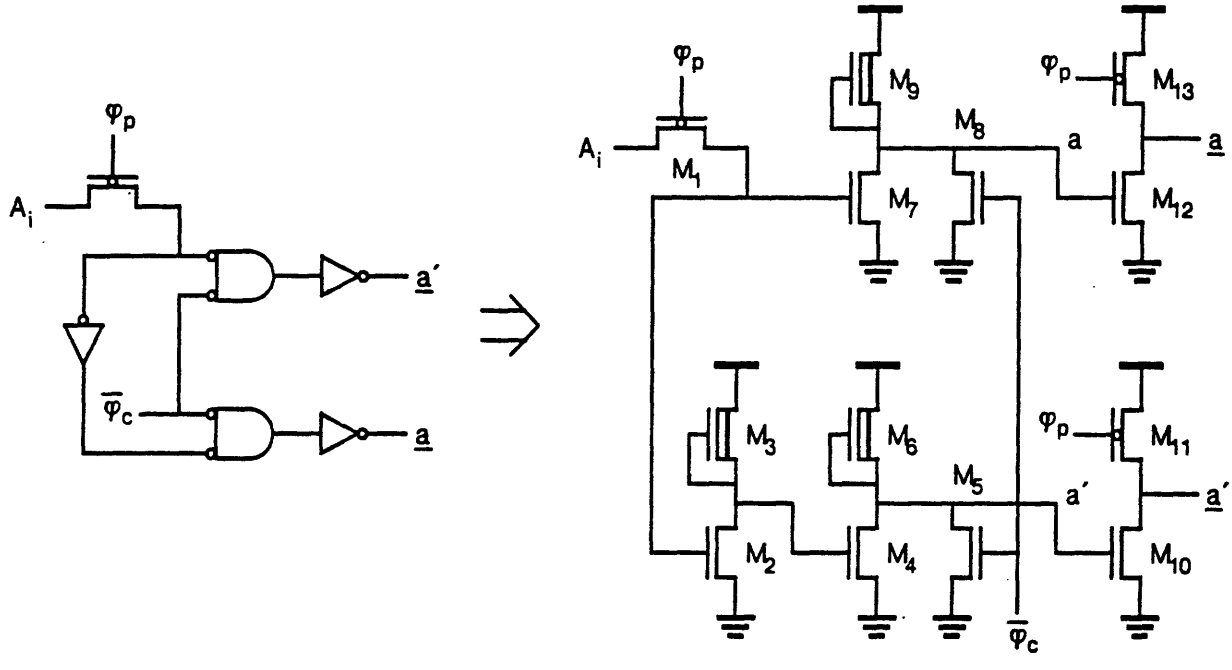


Figure 4-3: Input buffer cell circuit

Operation of the cell is simple:

- During initialization, the precharge and inverse compute clocks are both *high*. This forces $\underline{a}$, $\underline{a}'$ *high* and a, a' *low*. The input A_i is passed through M_1.
- When the precharge clock goes *low*, A_i is isolated from the input section. The outputs remain *high*.
- When the inverse compute clock goes *low*, either a or a' goes *high* depending on the value of A_i stored on the gates of M_2 and M_7. The selected output then goes *low*.

Obviously this circuit is well-behaved with respect to the clock inputs.

Figure 4-4 shows the layout of the input buffer cell. The data inputs are on the top edge, outputs

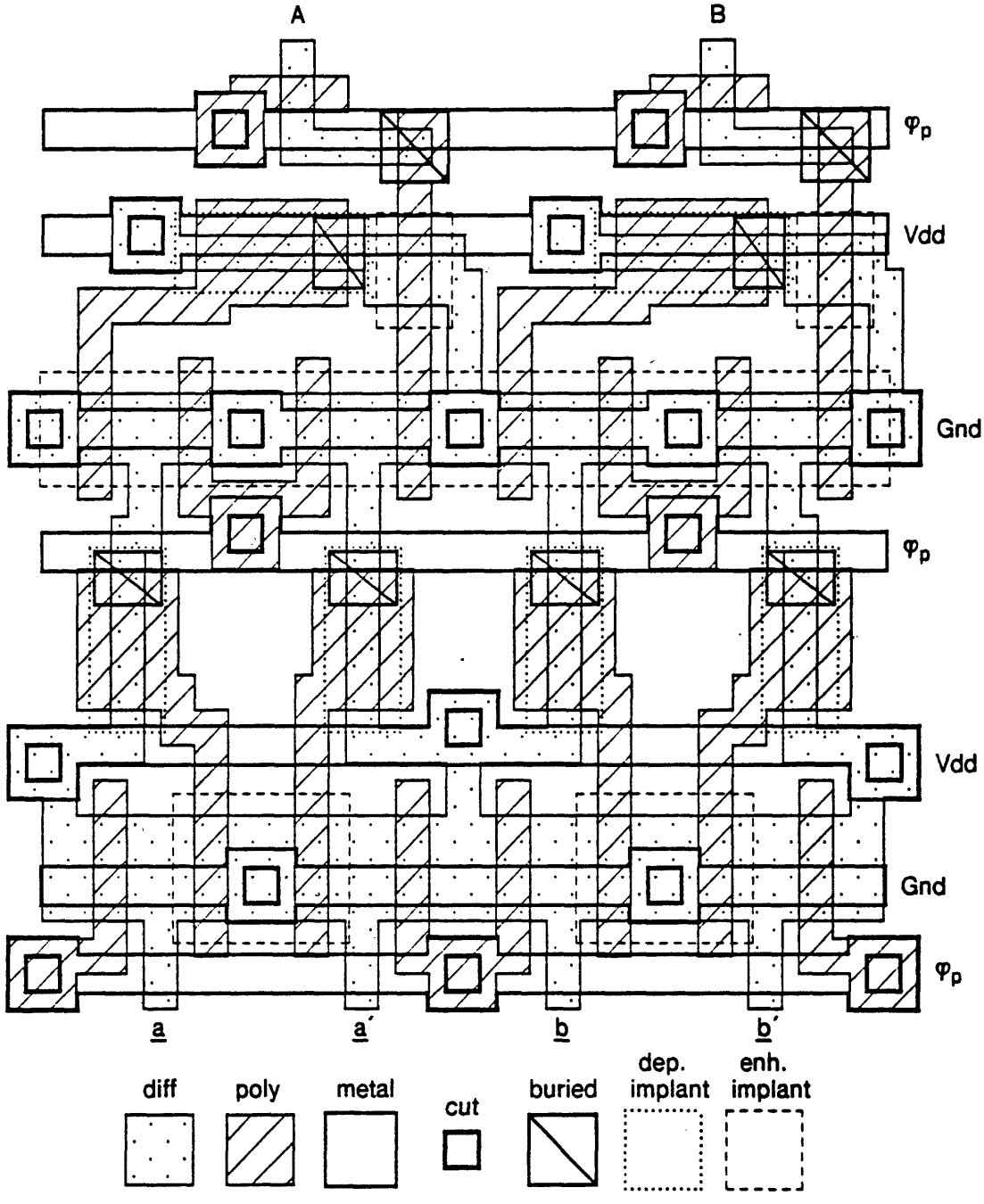


Figure 4-4: Input buffer cell layout

on the bottom edge, and control signals along the sides. Each cell is functionally complete and obeys the geometric design rules. A block of cells is built by replicating the basic cell with a 50λ pitch. Since the actual cell width is 54λ, the cells must overlap by 4λ. The other data bit cells follow this form closely.

The names and symbols for the available layers in the Bell NMOS process are shown at the bottom of figure 4-4. The value of λ is 2 microns in this process. The layers and geometric design rules for this process follow Mead and Conway style with the addition of buried contacts and an enhancement ion implant to distinguish high-threshold enhancement devices from intrinsic ones. The only notable variation from Mead and Conway rules is that metal width and spacing are both 2λ minimum rather than 3λ.

A special feature of this buffer cell layout is that the critical drive transistors have been oriented so that their sizes may be adjusted by stretching (or shrinking) the entire cell vertically while maintaining design rule correctness. Thus the drive capability can be varied without affecting the horizontal pitch or the form of the interface with the next block in the data path.

4.5.2 Exponent Compare Cell

Figure 4-5 shows the exponent comparator cell circuit. This cell has three sections: the comparator proper and two zero-detect NOR gate segments.

The comparator operates roughly like the comparator cell described in a previous section but its topology has been simplified as much as possible. In particular, this comparator only needs to decide whether or not to swap the input operands hence its result signal needs only two terminal states. We treat the case of $A_e = B_e$ the same as $A_e > B_e$. The result wire $\underline{s}_o$ becomes active if $A_e < B_e$; its complement, $\underline{s}_o$ ′becomes active if $A_e \geq B_e$. The initial condition of $\underline{s}_i$ inactive and $\underline{s}_i$′active is applied to the input of the LSB cell in a comparator block. This optimization eliminates the need for a separate wire for detecting $A_e = B_e$.

An isolated cell of this type is not obviously well-behaved. If fact, an individual cell of an N-bit comparator block is not perfectly well-behaved when considered as a module. However, we can show that the N-bit block is a well-behaved module. This is shown using induction: first we show that a 1-bit block is well-behaved, then we show that if an N-bit block is well-behaved, we can always add one more cell to get a N + 1-bit well-behaved block.

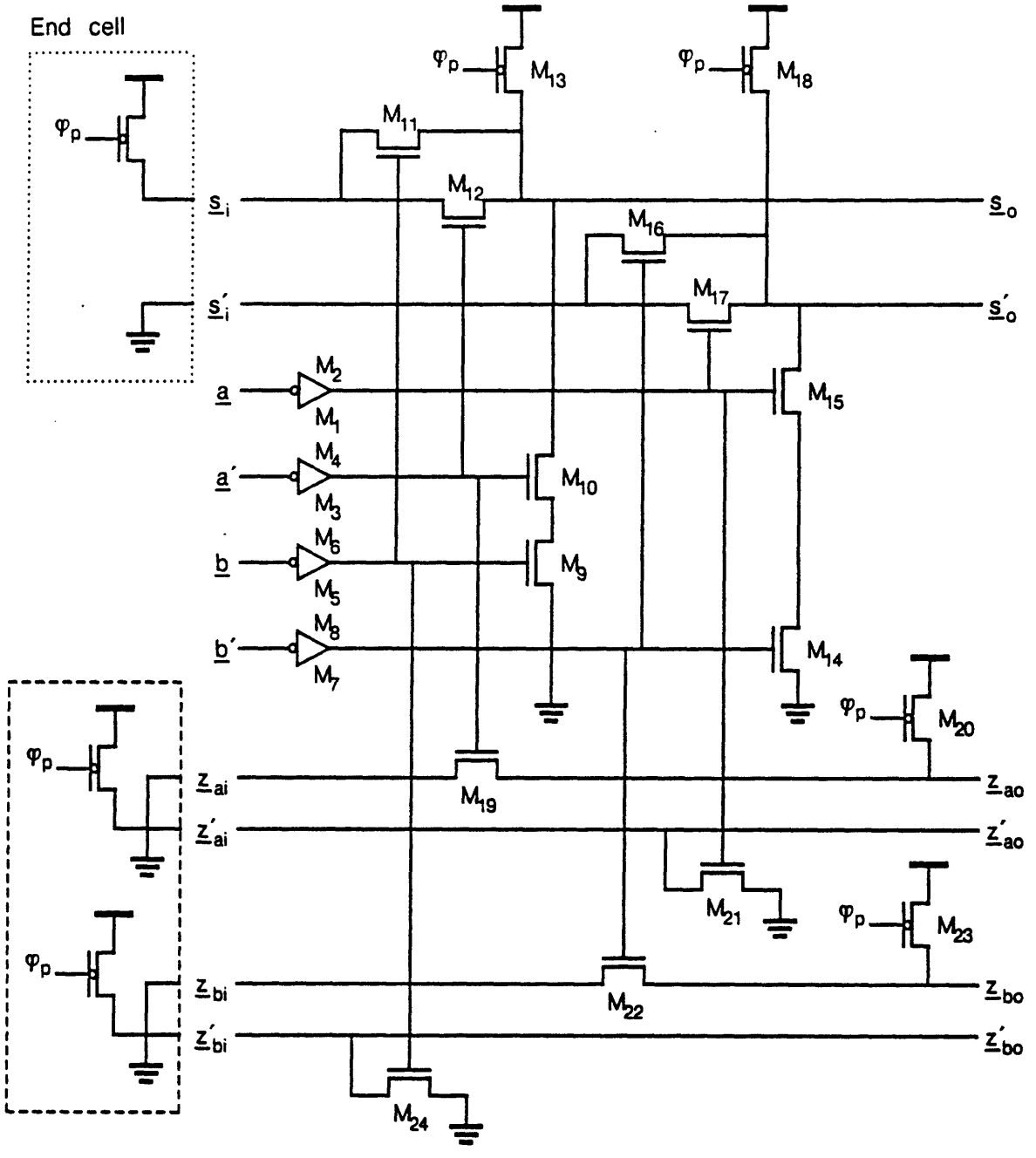


Figure 4-5: Comparator cell circuit

We first consider the 1-bit case. The $\underline{S}_i$ input signal is driven by the initial condition network shown inside dotted lines. For the purposes of proving well-behavedness we will consider the $\underline{S}_o$ signal from the MSB of the comparator block as the only output of the module; the other $\underline{S}$ signals are simply internal module signals. During precharging, the $\underline{A}$, $\underline{B}$, and $\underline{S}_o$ signals are set inactive (all wires *high*). After all the inputs have reached terminal state there are three possible conditions:

1. If $\underline{A} = 0$ ($\underline{a}$ *'low*) and $\underline{B} = 1$ ($\underline{b}$ *low*) then M_9 through M_{12} will be *on* and $\underline{s}_o$ will be pulled *low*. M_{14} through M_{17} will remain *off* and $\underline{s}_o$ *'*will stay *high*. The state of $\underline{S}_o$ will be 1 indicating $\underline{A} < \underline{B}$.
2. If $\underline{A} = 1$ ($\underline{a}$ *low*) and $\underline{B} = 0$ ($\underline{b}$ *'low*) then M_{14} through M_{17} will be *on* and $\underline{s}_o$ *'*will be pulled *low*. M_9 through M_{12} will remain *off* and $\underline{s}_o$ will stay *high*. The state of $\underline{S}_o$ will be 0 indicating $\underline{A} \geq \underline{B}$.
3. If $\underline{A}$ and $\underline{B}$ are equal then one switch in each pulldown network will be *off* and one switch in each pass network will be *on* thus the pulldowns will be off and $\underline{S}_i$ will get passed through to $\underline{S}_o$. The state of $\underline{S}_o$ will be 0 as for the previous case.

To build an N + 1-bit comparator, we connect the $\underline{S}$ output from a well-behaved N-bit comparator to the $\underline{S}_i$ input of a comparator cell. This cell becomes the new MSB cell whose output $\underline{S}_i$ is the N + 1-bit comparison output. To see that the N + 1-bit block is well-behaved, we consider the operation of the new MSB cell. During precharging, all inputs and outputs of the cell are set inactive (all wires *low*). After all of the $\underline{A}$ and $\underline{B}$ inputs to the entire block have reached terminal state we have three cases to consider in the MSB cell:

1. If the inputs $\underline{A}$ and $\underline{B}$ to the MSB cell are equal then, by the same reasoning as for the 1-bit case, the state of $\underline{S}_o$ will be the same as the state of $\underline{S}_i$.
2. If $\underline{A} = 0$ and $\underline{B} = 1$ then $M_{9\text{-}12}$ will all be *on* and $M_{14\text{-}17}$ will be *off*. This means that $\underline{s}_o$ *'*will remain *high* and $\underline{s}_o$ will be pulled *low*. Since M_{11} and M_{12} are *on*, $\underline{s}_i$ will get pulled *low* regardless of the terminal state of $\underline{S}_i$. This can cause the signal $\underline{S}_i$ to switch to an invalid state! Since there is no resistive path to the positive source inside the N-bit comparator driving $\underline{S}_i$ we know that $\underline{s}_o$ cannot fail to be pulled *low* regardless of the terminal state of $\underline{S}_i$. Hence the output $\underline{S}_o$ is well-behaved for this case even though $\underline{S}_i$ may not be.
3. Similarly, if $\underline{A} = 1$ and $\underline{B} = 0$ then $M_{14\text{-}17}$ are *on*, $M_{9\text{-}12}$ are *off*, $\underline{s}_o$ stays *high*, and $\underline{s}_o$ *'*goes *low* regardless of the terminal state of $\underline{S}_i$.

The well-behavedness of the NOR gate sections of the comparator is simple to verify. For an N-bit comparator block, the circuit inside the dashed box is connected to the LSB of the block. During precharging, all of the $\underline{Z}_o$ wires are precharged *high*. After all the $\underline{A}$ inputs to the block have

reached terminal state, either all the inputs are 0 or at least one input is 1. In the former case, the *low* level at the LSB $\underline{z}_{ai}$ propagates through each M_{19} and reaches the MSB $\underline{z}_{ao}$; $\underline{z}_{ao}$ ′remaining *high*. In the latter case, at least one M_{21} switches *on* and pulls $\underline{z}_{ao}$ ′*low*. The corresponding M_{19} stays *off* and prevents $\underline{z}_{ao}$ from going *low*. The $\underline{B}$ section is identical to the $\underline{A}$ section.

Figure 4-6 shows the layout of the exponent compare cell. The input operands are carried completely through the cell to connect to the input of the swap block. The comparison result flows from right to left following the orientation of the exponent data path from LSB to MSB. The zero detect result flows from left to right to make that result available to the fraction section. Unlike the input buffer cell, this cell does not require adjustable transistor sizing hence there are no constraints on transistor orientation.

4.5.3 Control Buffer Cell

The control buffer cell is used to drive dual-rail control lines which must drive a set of data bit cells in parallel. It consists of two identical sections which together convert a pair of active-low wires to a pair of active-high wires with much greater drive capability. Figure 4-7 shows the transistor level circuit for the $\underline{c}$,*c* control buffer section. The $\underline{c}$ ′,*c* ′section is identical.

This buffer circuit uses a new form of well-behaved switch-level gate. The output section $M_{4,5}$ is a well-behaved source follower buffer which is used to greatly reduce the load capacitance seen by the inverter stage, $M_{1,3}$; the worst-case effective load capacitance being the gate capacitance of M_5 rather than the full output load at *c*. Since M_4 and M_5 are both high-conductance devices, M_2 is added to the input stage to make sure that M_5 is *off* when M_4 *on*. This eliminates static power dissipation in the output stage and offers increased speed-power performance over, say, a super-buffer stage.

There is a performance trade-off involved in the selection of this particular buffer structure. Since M_5 is an enhancement device, it introduces a gate threshold drop into the output high level. This reduces the effective *on* conductance of switches driven by the output signal, thus increasing the total delay time of the circuit (as a function of the output high voltage level). We could use a bootstrapping buffer circuit to supply the gate of M_5 with enough voltage to eliminate the output threshold drop, but such a circuit would be very sensitive to variations in process parameters and would have a lower yield than the simpler circuit. In the NMOS process we are using, we can use an intrinsic device for M_5. SPICE simulations of the buffer and driven circuits indicate that the threshold drop of the intrinsic device does not introduce significant delay.

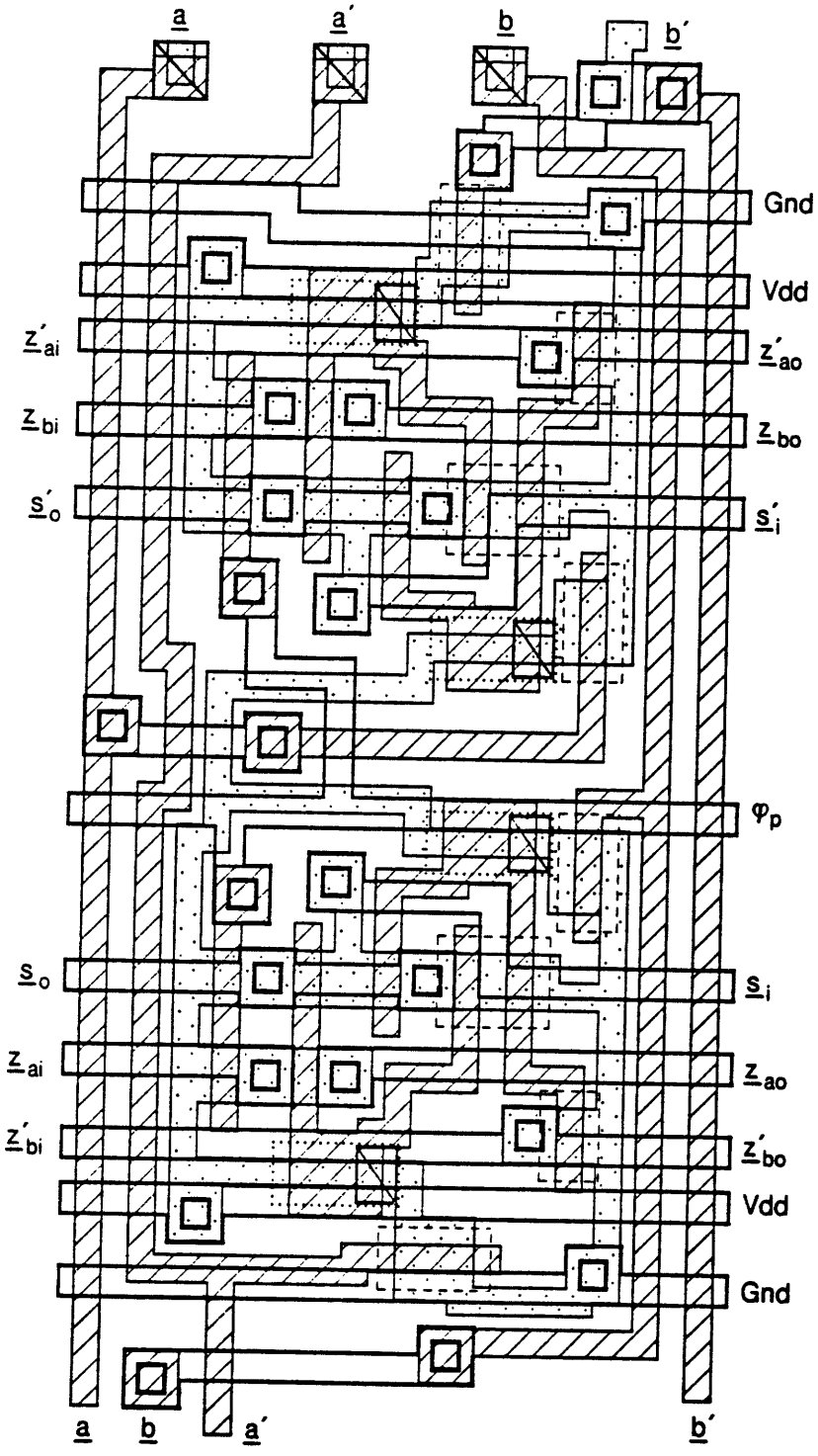


Figure 4-6: Exponent comparator cell layout

Figure 4-8 shows the layout of the control buffer. Like the input buffer, the critical drive transistors are oriented so that only one dimension of the buffer need change as the drive capability is changed.

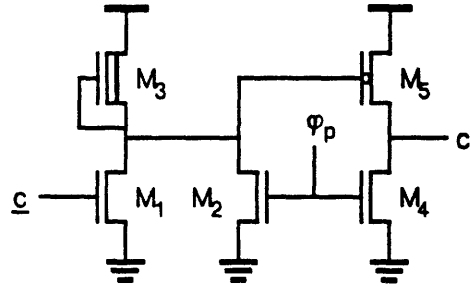


Figure 4-7: Control buffer cell circuit

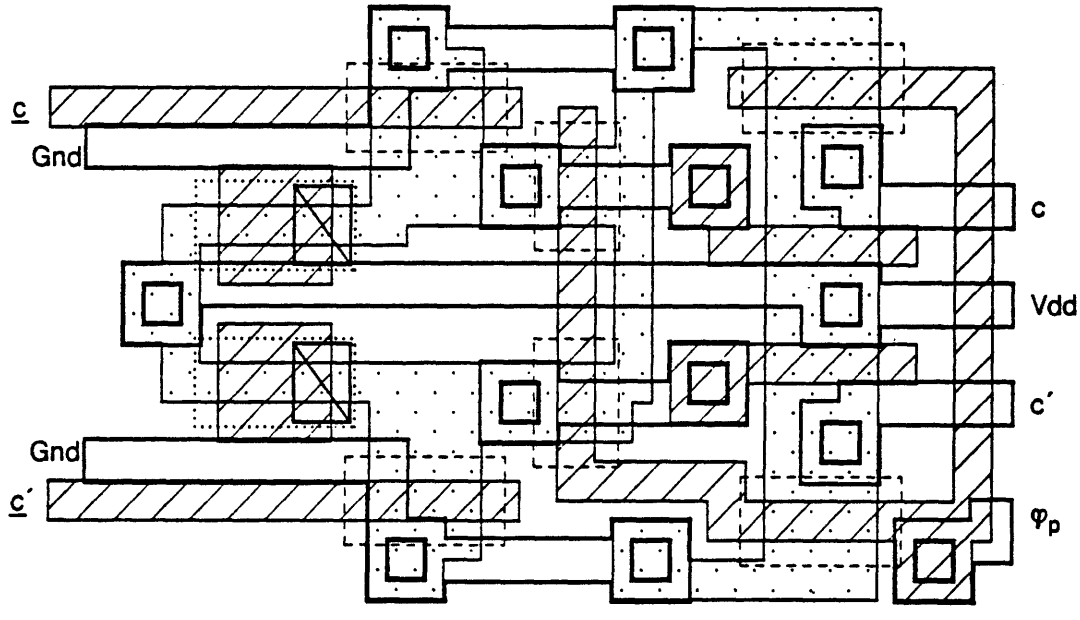
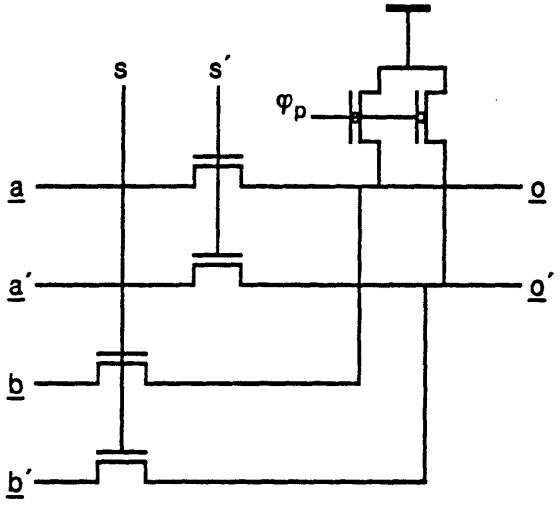


Figure 4-8: Control buffer cell layout

4.5.4 Multiplexer Cell

The multiplexer cell will be used as the basis for several of the function block cells. It selects one of two logical signals to pass through to the output. Figure 4-9 shows the transistor level circuit and logical truth table for this cell. The select control signal is dual-rail active high and is supplied by a control buffer cell. The data inputs and output are all dual-rail active-low signals. Note that if $\underline{b}$ and $\underline{b}'$ are connected to $\underline{a}'$ and $\underline{a}$ respectively then this block performs the logical XOR function, $\underline{O} = \underline{A} \oplus S$.



S	A	B	O
Ø	Ø	Ø	Ø
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Figure 4-9: Multiplexer cell circuit

Figure 4-10 shows the layout of the basic multiplexer cell. Since this cell is always part of a data path with two operand busses, the extra vertical poly wires are present to carry the unaffected bus through the cell.

4.5.5 OR cell

The OR cell is used in the denormalizer section to generate the "sticky" bit. Figure 4-11 shows the OR cell circuit. The circuit within the dotted outline is used to set the initial condition at the input of the first cell of an OR block. The operation of the circuit is similar to that of the zero-detect circuit in the exponent comparator cell.

Figure 4-12 shows the layout of the OR cell. This cell is designed to abut directly to the bottom edge of the multiplexer cell used in the denormalizer section.

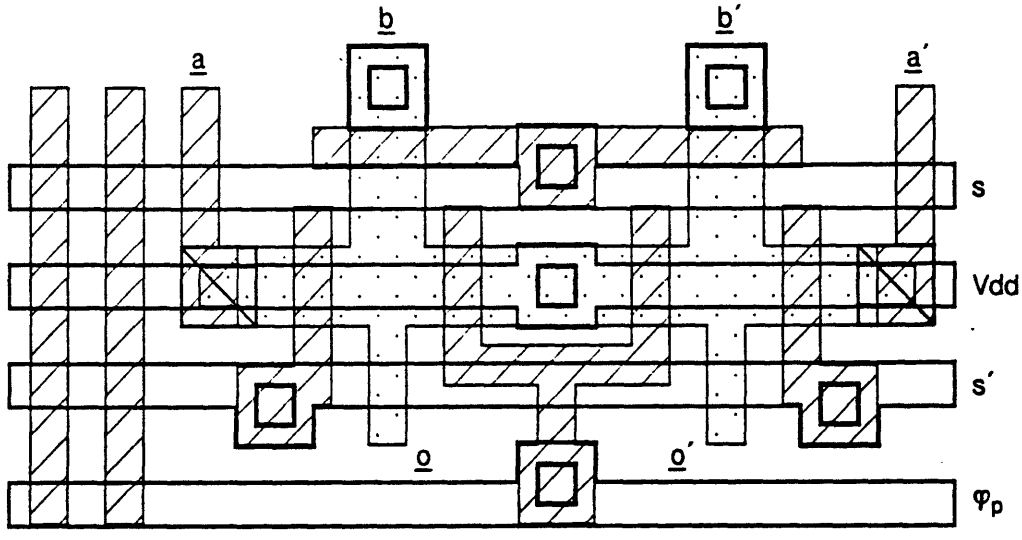


Figure 4-10: Multiplexer cell layout

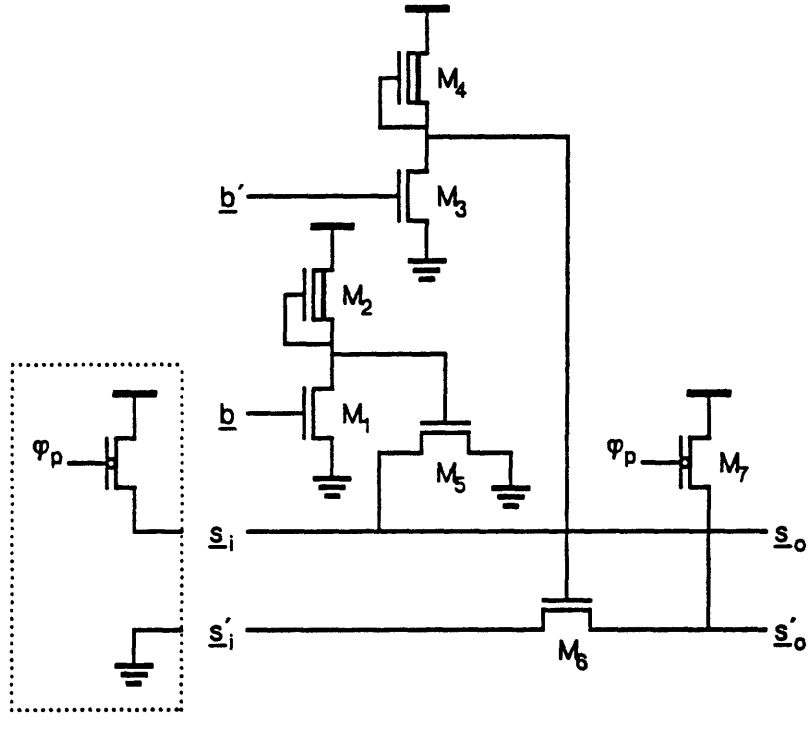


Figure 4-11: Denormalizer OR cell circuit

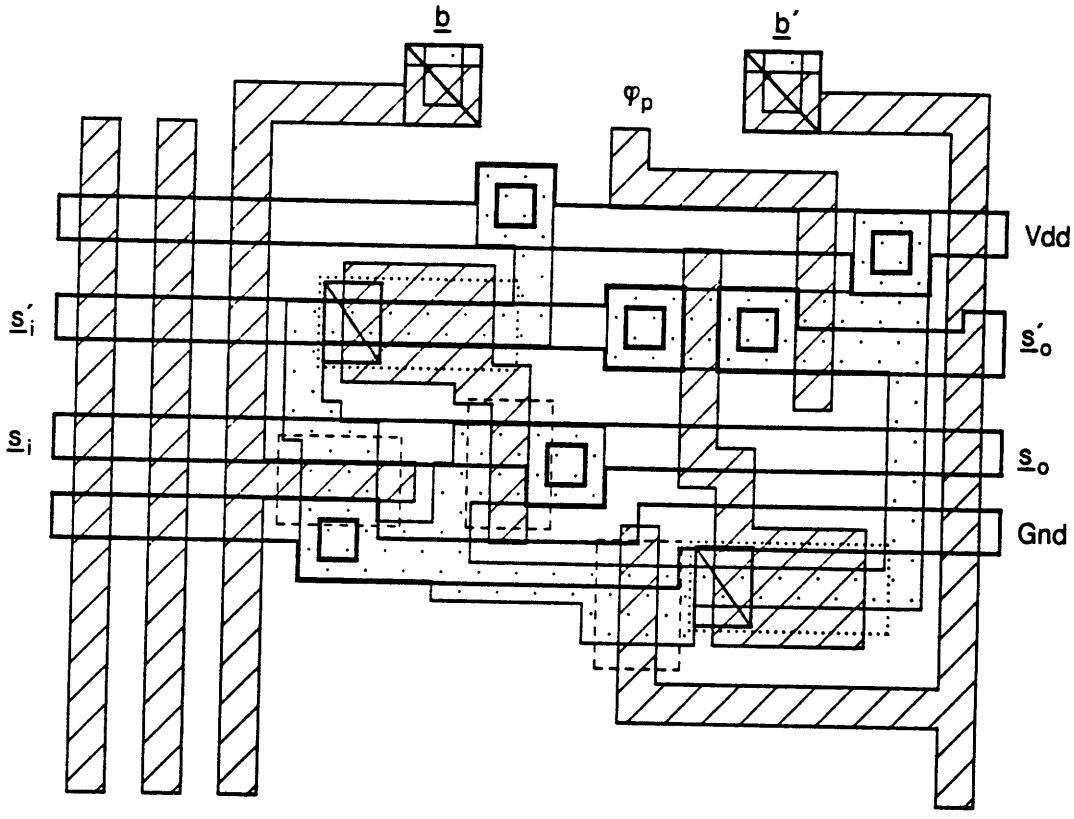
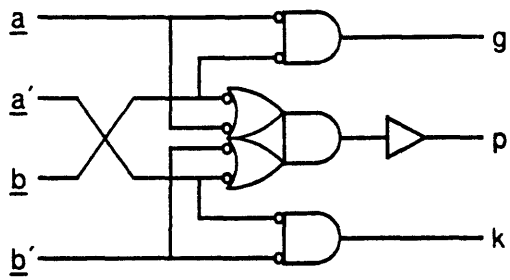


Figure 4-12: Denormalizer OR cell layout

4.5.6 Full Adder Cell

The basic full adder cell design will be used in all of the adder-type function blocks. It consists of three distinct modules: input decoder, carry chain, and sum generator. Since this is an element of a carry propagate adder, the carry chain module has been optimized for minimum delay at the expense of delay in the other sections.

Figure 4-13 shows the gate level circuit and logical truth table for the input section. The transistor level circuit is shown in figure 4-14. This section converts the two active-low dual-rail operand signals A and B into a single active-high triple-rail signal, X. The wires of this triple-rail signal are the familiar generate, propagate, and kill (*g*, *p*, and *k* respectively) used in several adder forms. The *g* wire is active only if a carry should be generated by this cell. The *k* wire is active only if a carry should not be generated. The *p* wire is active only if the carry input should be passed to the carry



A	B	X
Ø	Ø	Ø
0	0	k
0	1	p
1	0	p
1	1	g

Figure 4-13: Adder cell input circuit: gate level

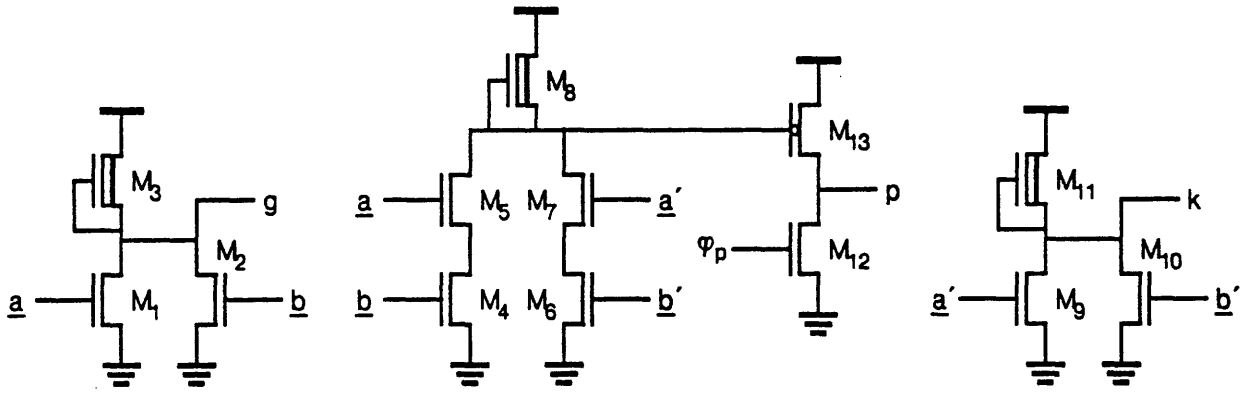
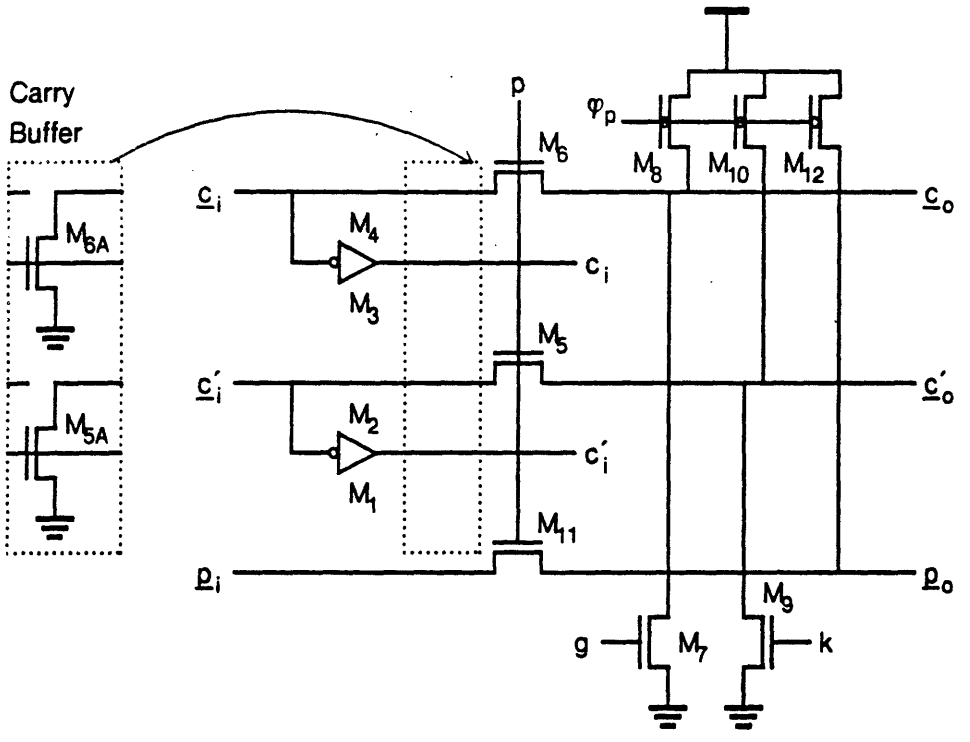


Figure 4-14: Adder cell input circuit: transistor level

output. Note the source-follower buffer $M_{12,13}$ used in the propagate wire. This buffer is a form of well-behaved gate with clocked pulldown. The buffer greatly reduces the load capacitance seen by the XOR gate used to drive the propagate wire.

Figure 4-15 shows the carry circuit and logical truth table. The carry chain section provides a high-speed path from the carry input to the carry output. The carry signals are both dual-rail active low. If *p* is active then the carry input is connected directly to the carry output. If *g* is active then the carry 1 output is set active. If *k* is active then the carry 0 output is set active. This section also includes an encoding inverter, $M_{1\text{-}4}$, used to convert the active-low carry input to an active-high signal for use in the sum section.

As the leading edge of a waveform travels through a series of pass transistors such as the Manchester carry chain, the transition time becomes slower and slower. This causes the propagation



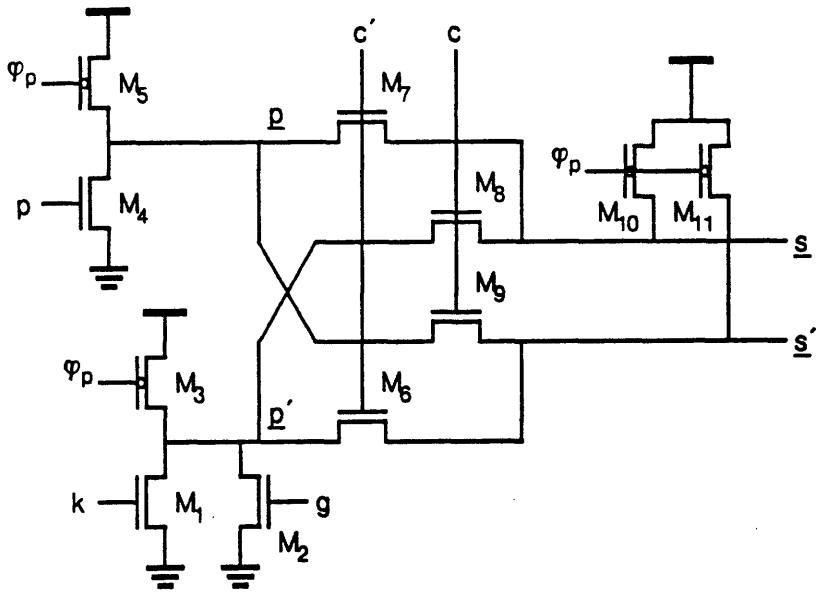
X	$\underline{C}_i$	$\underline{C}_o$
Ø	Ø	Ø
k	0	0
k	1	0
p	0	0
p	1	1
g	0	1
g	1	1

Figure 4-15: Adder cell carry circuit

delay through the entire chain to increase worse than O(N), the actual performance being roughly $O(N^2)$. To counteract this effect we use the carry inverter to provide for periodic buffering of the pass transistor chain. This is done by replacing the wiring inside the dotted box with transistors M_{5A} and M_{6A} shown in the inset. The resulting double inversion sharpens the waveform edge and permits expansion of the adder with O(N) delay performance. This form of carry speed-up circuit fits our procedural design criteria well. Since there is very little difference between the topology of the buffered and unbuffered carry sections, we can lay out both versions within the same area. In fact, the actual layout was done so that all circuitry except the outlined region is identical.

The carry section also includes transistors M_{11} and M_{12} which are used to generate a global propagate signal over all the bits in an adder block. This is done by grounding the $\underline{p}_i$ input of the LSB bit cell. Only if all of the bit cells propagate will the $\underline{p}_o$ wire of the MSB bit cell become active. The purpose of this signal will be discussed later.

Figure 4-16 shows the transistor level circuit and truth table for the sum section. The sum section computes the sum bit using the active-high carry input signal C_i and the triple-rail signal X



X	P	C_i	S
Ø	Ø	Ø	Ø
k	0	0	0
k	0	1	1
p	1	0	1
p	1	1	0
g	0	0	0
g	0	1	1

Figure 4-16: Adder cell sum circuit

from the input section. The *p* wire is inverted and the *g* and *k* wires are NORed together to obtain a dual-rail active-low $\underline{P}$ signal where $\underline{P} = \underline{A} \oplus \underline{B}$. The $\underline{P}$ signal is then XORed with C_i signal to obtain the active-low dual-rail sum bit $\underline{S}$.

The full adder cell is a well-behaved module by itself and can be cascaded to form well-behaved N-bit adder blocks. In the fraction adder section we need to use feedback from the MSB carry output to the LSB carry input for 1's complement arithmetic. This feedback is forbidden by the interconnection rules given for well-behaved modules. However, we can get well-behaved operation from the adder block by using the global propagate output wire from the most significant adder bit. We can show how this works by considering the possible behavior of the looping carry chain under all possible conditions after the operand inputs have reached terminal state. If there is at least one cell which does a carry generate or kill then a valid carry signal will eventually reach the MSB carry output since the cells to the left of that cell constitute a well-behaved logic network. Similarly, the cells to the right of the original cell constitute a well-behaved network hence the LSB carry input will eventually propagate to the carry input of the original cell. Since the original cell was not doing a propagate operation, its carry output is *not* a function of its carry input thus the carry signal *never* propagates through more than N-1 cells and the single-transition rule is obeyed. The only remaining condition is when all N cells do a carry propagate. In this case none of the carry or sum outputs become defined.

This condition is detected by the global propagate output from the MSB cell. In 1's complement arithmetic, this corresponds to a zero result so we can feed the global propagate output to the exception handling unit as an additional zero result detection input.

Figure 4-17 shows the layout of the complete full adder cell. Note the that the carry feedback wiring is provided for within the cell itself. This is much more efficient than routing the wires around the entire adder block. Carry chain buffering is selected by replacing the two metal straps in the carry input section with the section of layout within the dotted outline.

There are two major variant forms of the basic adder cell. These are both *half-adder* cell forms. These cells are used where the B operand bit is a constant (either 1 or 0). These cells are derived directly from the full adder cell by applying the constant B input and removing as many transistors as possible while retaining the proper logical function and well-behavedness. A total of nine transistors (seven in the input section) are removed in each case. The B = 0 half adder is used in the fraction adder MSB position. The B = 1 cell is used in the exponent adjust blocks.

4.5.7 Zero Count Cell

The basic zero count cell is divided into two sections: a header section which is identical for each cell in a block and the count output section which contains the position count for the cell.

Figure 4-18 shows the circuit for zero count cell. This circuit is not a complete well-behaved module by itself; the zero count block is well-behaved only when considered as a complete unit. The $\underline{z}_i$ wire is an active-low input from the next more significant cell which is active when all of the more significant result bits are 0. The $\underline{R}$ input ($\underline{r}$, $\underline{r}$) is the active-low dual-rail result bit. The $\underline{z}_o$ wire connects to the $\underline{z}_i$ input of the next less significant bit in the block. The active-high *o* wire connects to the enable input of the count output section. The output section consists of a set of pulldowns, one for each dual-rail active-low count output signal, $\underline{A}_i$. Each pulldown selects whether its count output signal will be set to 0 or 1 when the output section is enabled, the pulldown pattern being fixed for each cell. The active-low count output signal wires are bussed through all the zero count cells in parallel with one clocked pullup for each wire for precharging.

The operation during the compute period is as follows: if $\underline{R}$ is 0 ($\underline{r}$ *'low*) then pass $\underline{z}_i$ through M_1 to the $\underline{z}_o$ output otherwise if $\underline{R}$ is 1 ($\underline{r}$ *low*) then send $\underline{z}_i$ inverted to the output enable wire *o*. As with the carry chain in the adder cell, we provide for periodic buffering of the $\underline{z}$ chain by replacing the dotted outlined wiring with the M_{1A} circuit.

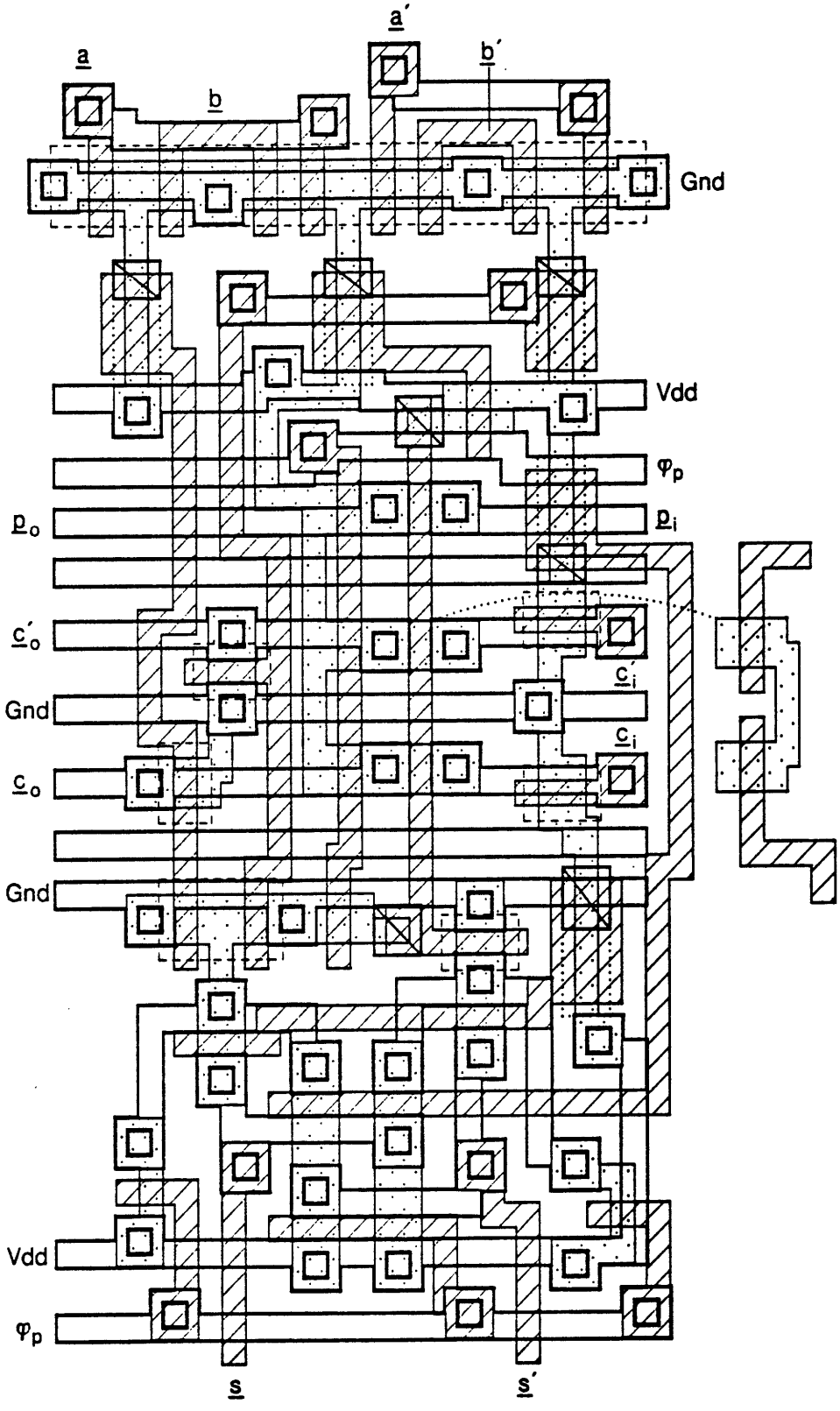


Figure 4-17: Full adder cell layout

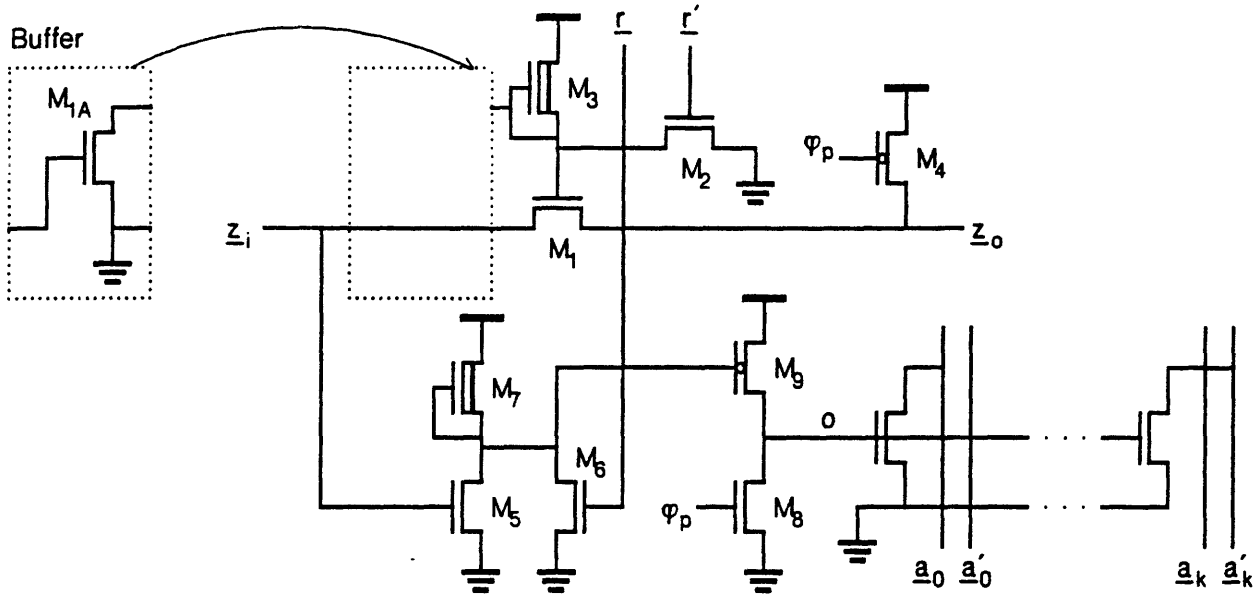


Figure 4-18: Zero count cell circuit

The operation of a cascaded set of N zero count cells is as follows:

- During precharging, all of the $\underline{z}_{o_k}$ wires are set *high* and all of the output enable wires o_k are set *low*. The $\underline{z}_i$ wire of the MSB cell is tied *low*.
- During the compute period, the *low* level on $\underline{z}_{i_k}$ is propagated rightward through all cells with leading zeros. The most significant cell with a 1 halts the propagation of the $\underline{z}$ signal and asserts its output enable wire. The output section of that cell then asserts its location on the set of N_z count output lines.
- If there is no 1 in the result then the $\underline{z}_o$ output of the LSB cell will become active.

The entire zero count block can be considered to be well-behaved since, in the terminal state, either the count output signals are defined or the result zero output is asserted. The result zero output feeds into the exception handling circuit which will direct the output buffer to produce a zero output should the zero count block signal a zero result.

Figure 4-19 shows the layout of a zero count cell with three count output signals. It is composed of three kinds of basic cell: an input header, three output wiring cells, and three output pulldown cells. As with the full adder, zero detect chain buffering is selected by replacing a wiring cell with a cell containing an extra pulldown.

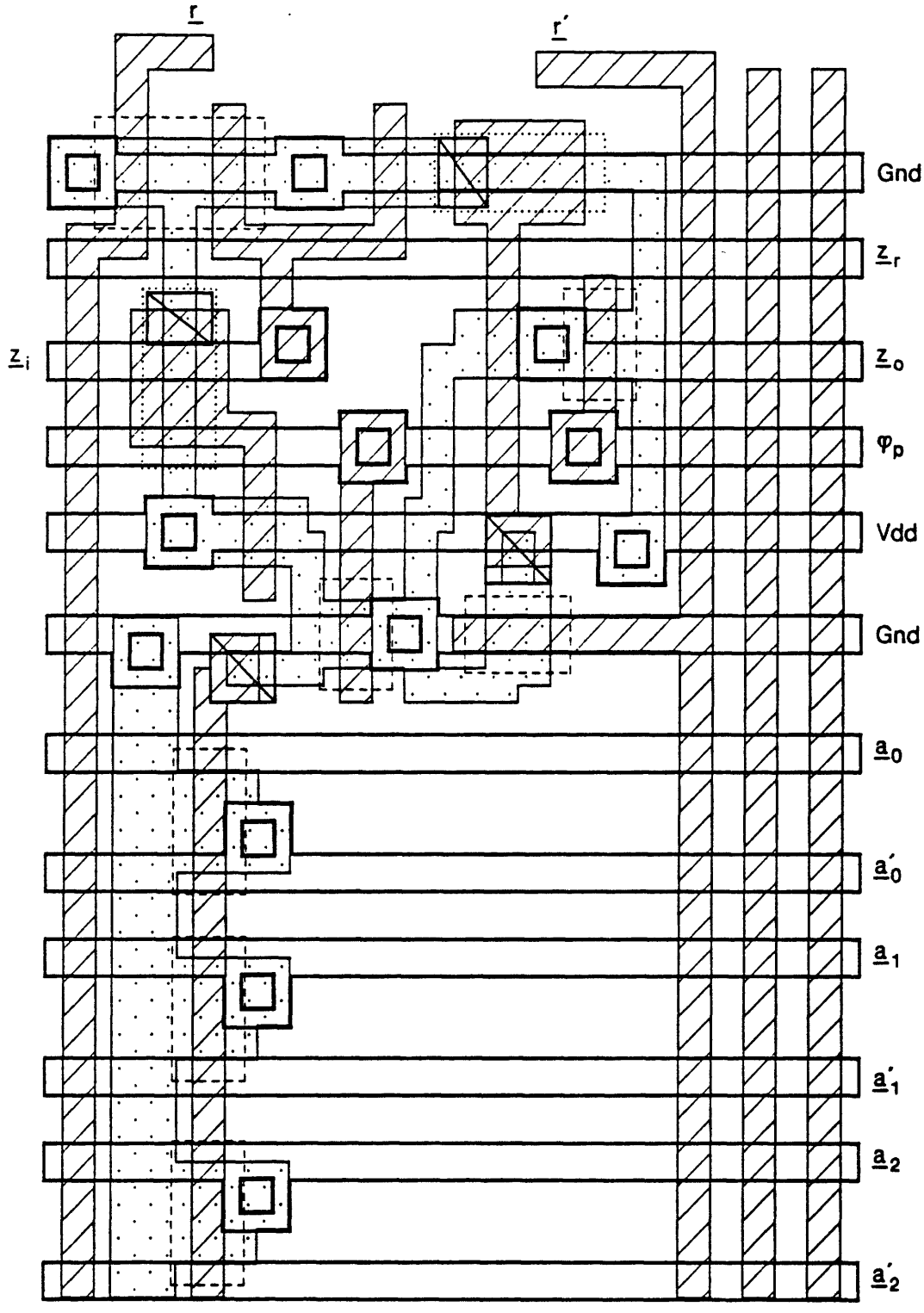
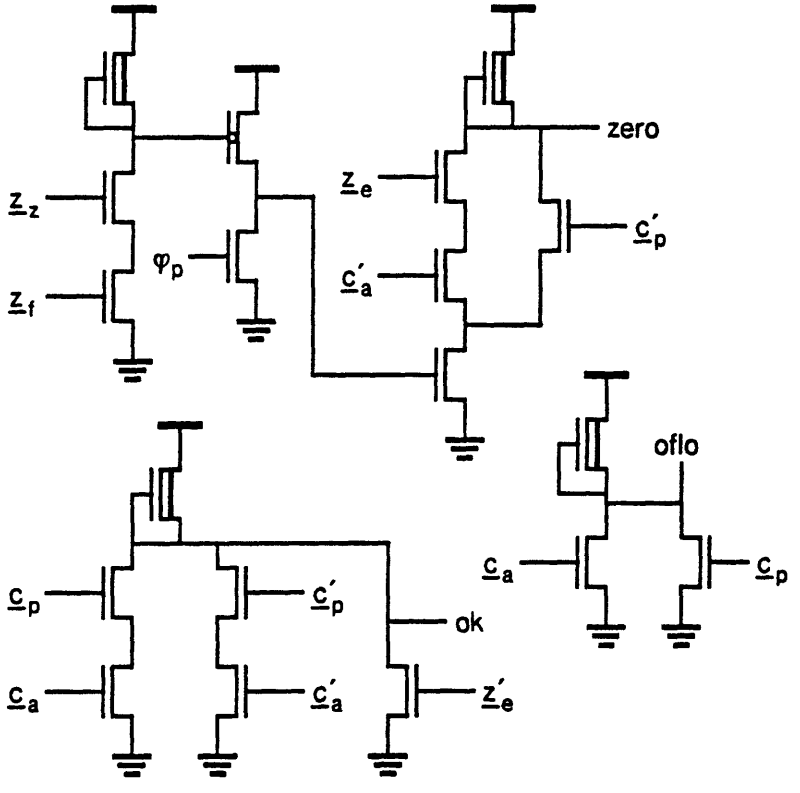


Figure 4-19: Zero count cell layout

4.5.8 Exception Handling

The exception handling block contains the logic which detects "error" conditions and other states of the adder unit which require special consideration. The three basic conditions sensed by this block are *overflow*, *underflow/zero result*, and *normal result*. Figure 4-20 shows the transistor level circuit and truth table for the exception handling block. The inputs to the block are the zero result wires from the fraction adder and zero count block ($\underline{z}_f$ and $\underline{z}_z$ respectively), the exponent pre-adjust carry signal ($\underline{C}_p$), the exponent adjust carry signal ($\underline{C}_a$), and the exponent zero signal ($\underline{Z}_e$). All of the inputs are active low. The block produces a triple-rail active-high status signal ST which is composed of the three wires *ok*, *zero*, and *oflo*.



$\underline{Z}_f$	$\underline{Z}_z$	$\underline{C}_p$	$\underline{C}_a$	$\underline{Z}_c$	ST
Ø	Ø	Ø	Ø	Ø	Ø
T	Ø	-	Ø	Ø	zero
Ø	T	-	Ø	Ø	zero
Ø	Ø	0	0	0	zero
Ø	Ø	0	0	1	zero
Ø	Ø	0	1	0	ok
Ø	Ø	0	1	1	zero
Ø	Ø	1	0	0	ok
Ø	Ø	1	0	1	-
Ø	Ø	1	1	0	-
Ø	Ø	1	1	1	oflo

Figure 4-20: Exception handling circuit

As in the previous truth tables for well-behaved modules, the first line of the truth table lists the output state with inputs in the initial state and the remaining lines list the output terminal state for all possible input terminal states. Note that the some of the input terminal states are null. This is a result of the behavior of the fraction adder and zero count blocks when they have detected a zero result

condition (i.e. they fail to produce non-null terminal states on their "normal" output signals). Since the exception handler produces a valid non-null state on its output for all possible input terminal states, we can say that it is well-behaved with respect to the input signals to the adder unit.

The ST output is connected to the control input of the output buffer block. The output block guarantees that the output signals of the adder unit are well behaved with respect to the adder unit inputs.

4.5.9 Sign Handling

The sign handling cell contains all of the logic for controlling the two fraction negation blocks and computing the result sign. As such there is only one of these cells in the entire adder unit. It is designed to be placed to the left of the MSB cell of the fraction adder. Figure 4-21 shows the transistor level circuit for the sign cell.

The input and output signals are all dual-rail active low. All of the sign bit signals are encoded so that a 1 state represents a negative value. Inputs $\underline{A}_i$ and $\underline{B}_i$ are the sign bits of the input operands from the output of the swap block. Input $\underline{C}$ is the carry output from the fraction adder. Output $\underline{B}_o$ is a buffered version of $\underline{B}_i$. Output $\underline{A}_o$ is the sign bit of the A operand to the fraction adder block. It is connected to the control input of the A_f negate block and to the A input of the fraction adder MSB cell (the B input to the adder MSB is always 0.) Output $\underline{R}$ is the sign of the raw result from the fraction adder. It is connected to the control input of the R negate block.

The logic equations for the output signals are $\underline{B}_o = \underline{B}_i$, $\underline{A}_o = \underline{A}_i \oplus \underline{B}_i$, and $\underline{R} = \underline{A}_o \cdot \overline{\underline{C}}$. The true sign of the result is computed by passing the buffered $\underline{B}_o$ signal through one bit cell in the result negate block, thus performing an exclusive-or operation on the raw sign and the original B operand sign.

4.5.10 Output Buffer Cell

The output buffer cell circuit is shown in figure 4-22 along with its truth table. It converts the raw active-low dual-rail result $\underline{R}$ into an active-high dual-rail signal R. The three control inputs *ok*, *oflo*, and *zero* are driven by the ST signal from the exception handling block. As in the exception handling block, the truth table for the output buffer contains null input terminal states. Since the exception handler forces the output buffer to produce a zero result whenever it is possible for the $\underline{R}$ input to fail to reach a non-null terminal state, the output buffer and the adder unit itself are always well behaved with respect to the adder unit inputs.

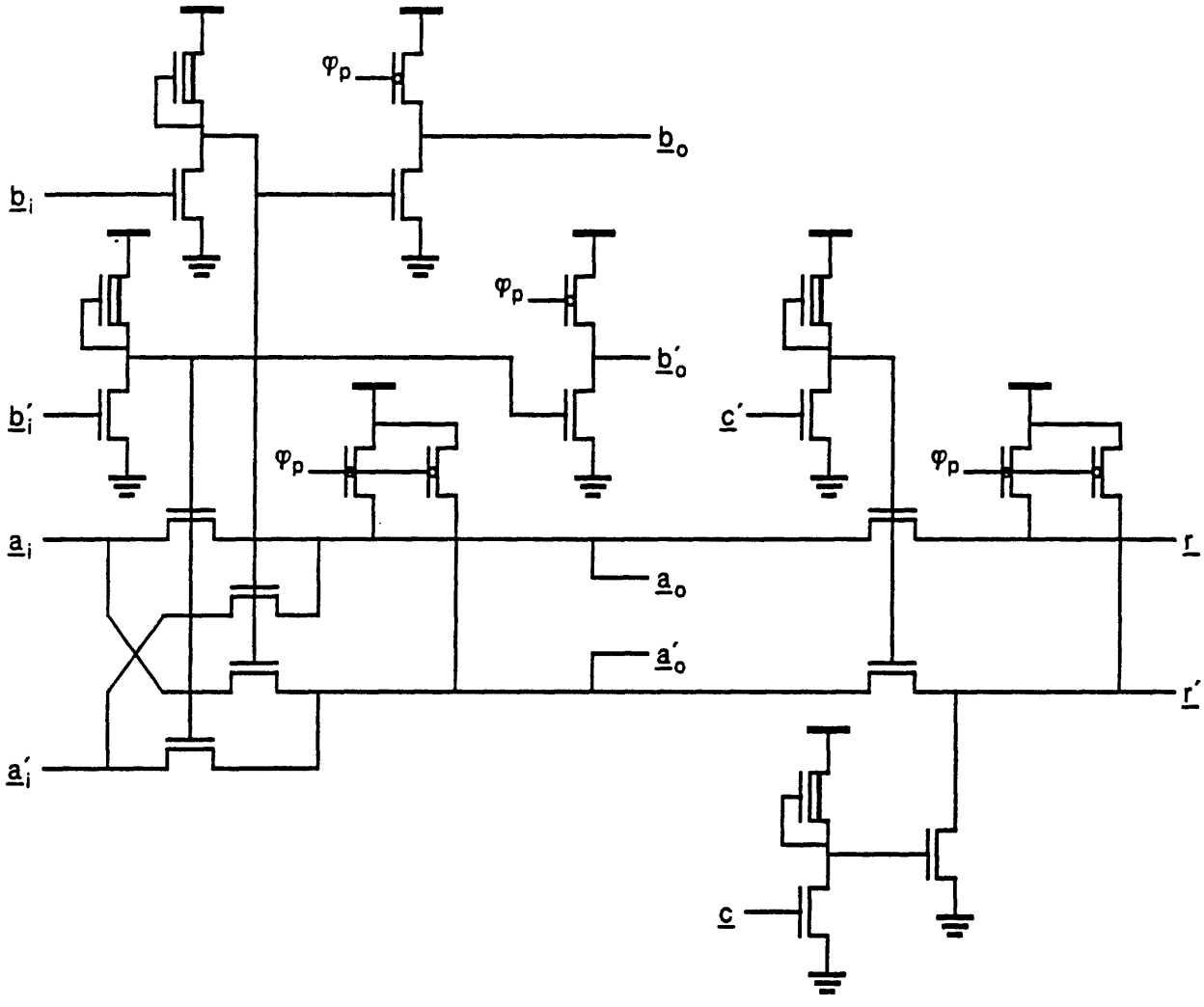


Figure 4-21: Sign handling cell circuit

To permit the raw result sign to be passed to the output on overflow, the output buffer cell for the sign bit has M_2 connected in parallel with M_1.

The wires $\underline{c}_i$ and $\underline{c}_o$ are used to generate a *completion signal*, C, for the adder unit. This is a single-rail active-low signal which has the single terminal state, *true*. It only reaches terminal state after all of the other adder outputs have reached terminal state, indicating that the addition is complete. The $\underline{c}$ wires of all the output cells are connected in series with the first $\underline{c}_i$ wire grounded. During precharging, all of the $\underline{c}_o$ wires are precharged *high*. When all of the result output bits have become defined, the initial ground level will be passed through to the $\underline{c}_o$ wire of the last stage.

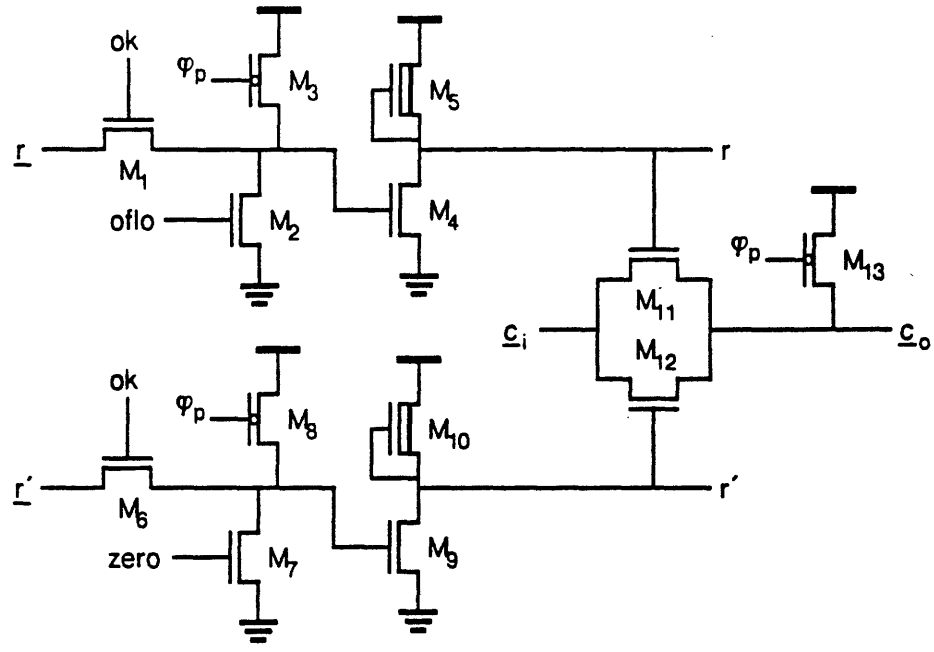


Figure 4-22: Output buffer cell circuit

Figure 4-23 shows the layout of the output buffer cell.

The completion detect stage is only needed for a system using asynchronous clocking. The completion signal would be used to restart the precharge/compute cycle to obtain minimum average cycle time. When the adder unit is used as a module in a synchronous system, the completion detect signal would only serve as a diagnostic feature to indicate whether or not the adder was able to complete its computation within the allotted cycle time.

In a synchronous system, the output stages can simplified to reduce the total area of the adder unit. Since we will have set the clock period to allow for the worst case delay through the adder, we have no need to be able to tell when the computation has ended; we assume that the adder will always reach terminal state before the end of the compute period. In this case we only need one output wire for each output signal since the other wire only supplies redundant information about the terminal state. We can derive a "stripped down" version of the adder circuit by removing all circuitry used *only* to compute the redundant information. If we eliminate the *r'* wire from each output bit cell, then we can delete $M_{6\text{-}10}$ in each output cell as well as the circuits which generate the *zero* wire in the

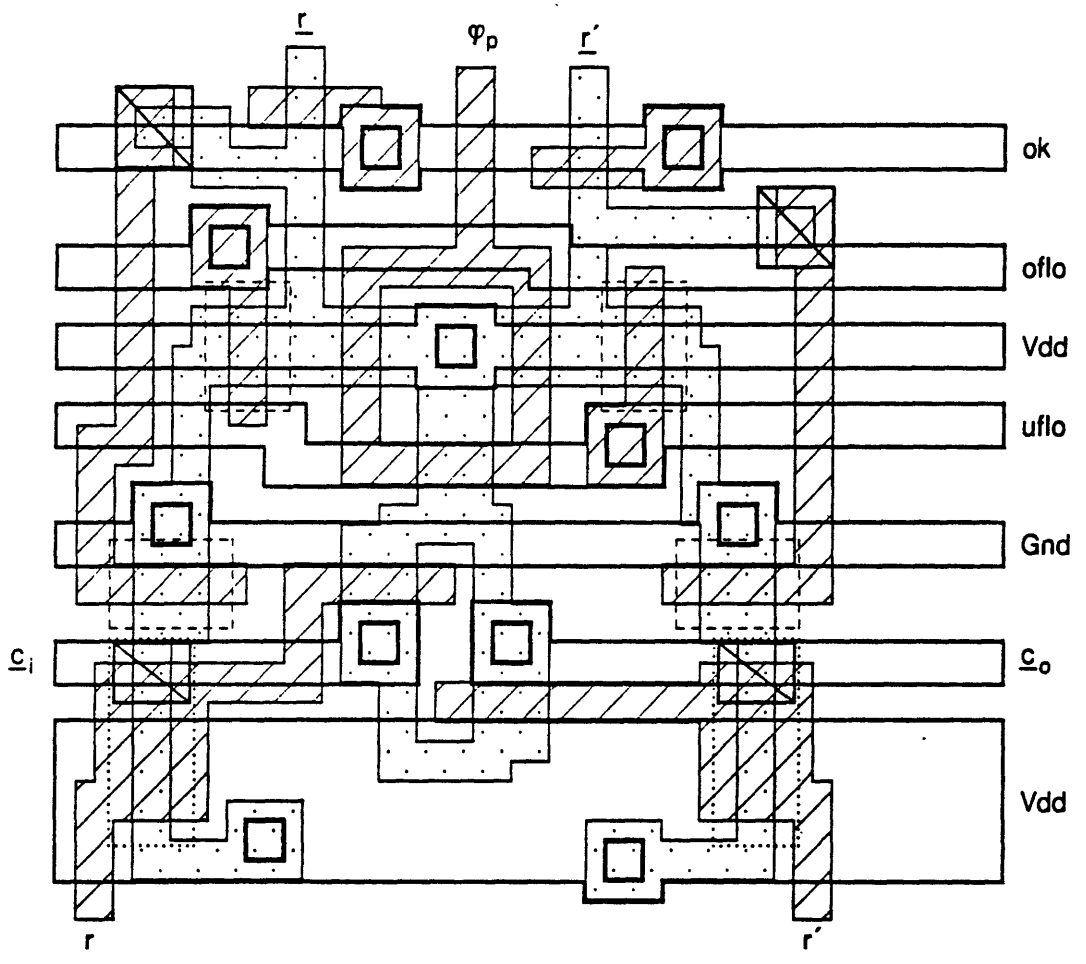


Figure 4-23: Output buffer cell layout

exception handling block. This process can be carried further to eliminate the $\underline{r}'$ paths from the normalizing shifter and the zero-detect circuitry from the adder and zero-count blocks.

4.6 Building Function Blocks From Simple Cells

Now that we have a set of basic cells, we need to specify an algorithm for constructing a complete floating-point adder using these basic cells. The decomposition of the original algorithm proceeded in a top-down sequence from algorithm to block diagram to cell specification. The construction process will proceed in the inverse order from basic cells to function blocks to complete layout.

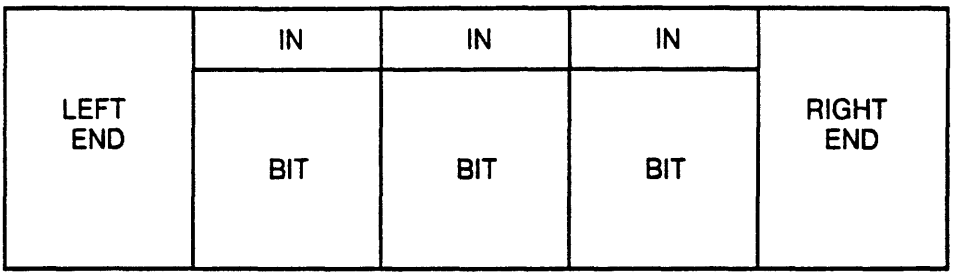


Figure 4-24: Generalized N-bit function block layout

The first step in the construction process is to specify procedures for building each of the different function blocks. Figure 4-24 shows the general form of a variable width function block. This form applies, with minor variations, to all the data path blocks in the adder unit. The generalized layout of an N-bit function block consists of a set of N identical *bit cells*, two *end cells*, and a set of N identical *interconnect cells*. The bit cell is one of the basic cells described in the previous section. The end cells provide power and ground connections to the bit cell vector as well as buffering for horizontal control inputs, if needed. Often the end cells provide constant initial conditions for cascade inputs to the bit cell vector (e.g., a constant input to the LSB carry input of an adder block.) The interconnect cells are optional and are used to align the input wiring of the bit cells with the output wiring of the bit cells of the function block above.

One of the advantages of using this kind of general form is that it makes verification of adherence to geometric design rules (design rule checking) very easy. Since a 2-bit block contains each possible kind of cell interface we only need to design rule check this one version of the block. The only additional check needed for an N-bit block is to make sure that the bit cell pitch is correct so that the cell to cell interface remains the same as the block width varies. The cells are permitted to overlap at their boundaries. This causes no problems in applying the verification rules provided that the overlap is confined to the boundary regions (i.e. one kind of cell cannot completely overlap its neighbor). What is important is that there must be a fixed number of kinds of cell overlap independent of the size of the block.

The interface between two function blocks can also be verified by abutting a 2-bit example of each block vertically and checking the resulting layout. If the 2-bit interface is correct, each block is correct for any N by itself, and the bit cell pitch is the same for both blocks, then the interface will be correct for any N.

In the discussion of the function blocks below, we make the following assumptions about the function block form:

- All of the function blocks have left and right end cells.
- Ground wiring is supplied by the left end cells in the fraction section and by the right end cells in the exponent section (the inside ends). The positive supply wiring is supplied by the outside end cells in each section.

Only deviations from these assumptions will be mentioned.

4.6.1 Input Buffer

The input buffer is divided into two parts: an N_f-bit block for the fraction section and an $N_e + 1$-bit block for the exponent section. There are two procedures, one for each block. Both procedures use the basic input buffer as the bit cell but differ in the form of the end cells used. The end cell of the fraction block includes a version of the input buffer cell with the normal-to-dual-rail conversion section removed to receive the "hidden bit" data from the exponent section. Thus the fraction block has an output bus width of $N_f + 1$ bits. The exponent block includes the sign bit next to the LSB of the exponent proper. This makes room for the "hidden bit" wiring from the exponent comparator to the fraction MSB input.

4.6.2 Exponent Comparator

The exponent comparator is built from N_e comparator cells and two end cells. The left end cell feeds the comparison output to the swap block below. The right end cell provides precharging for the zero detect gates, the initial condition for the comparison input, and wiring for the sign bit and zero detect output.

4.6.3 Operand Swap

Like the input latch, the operand swap unit is divided into two parts: an $N_e + 1$-bit block for the exponent section which includes the sign bit and an $N_f + 1$-bit block for the fraction section. The swap cell is built from two multiplexer cells. One multiplexer swaps the $\underline{a}$ and $\underline{b}$ wires while the other swaps the $\underline{a}'$ and $\underline{b}'$ wires. The left end cell of the exponent swap block contains a control buffer which amplifies the comparison output from the comparator section above. The right exponent end cell connects together the control lines of the two multiplexer sections in the exponent block cells. The left end cell of the fraction block contains two control buffers driven in parallel by the control wires from the exponent block.

4.6.4 Exponent Difference

The exponent difference block is built from N_e modified full adder cells and two end cells. The modified cell is functionally the same as the basic full adder but contains two extra vertical wires to carry the A operand through the cell for later use in the exponent section. The right end cell applies the constant carry input of 1. Since the bit cell could not be made to fit the 50 lambda pitch constraint, this block cannot be directly abutted to its neighboring blocks. A simple river routing procedure is used to generate the interconnect. This problem is discussed more fully in the section on fraction and exponent placement and interconnection.

4.6.5 B_f Denormalize

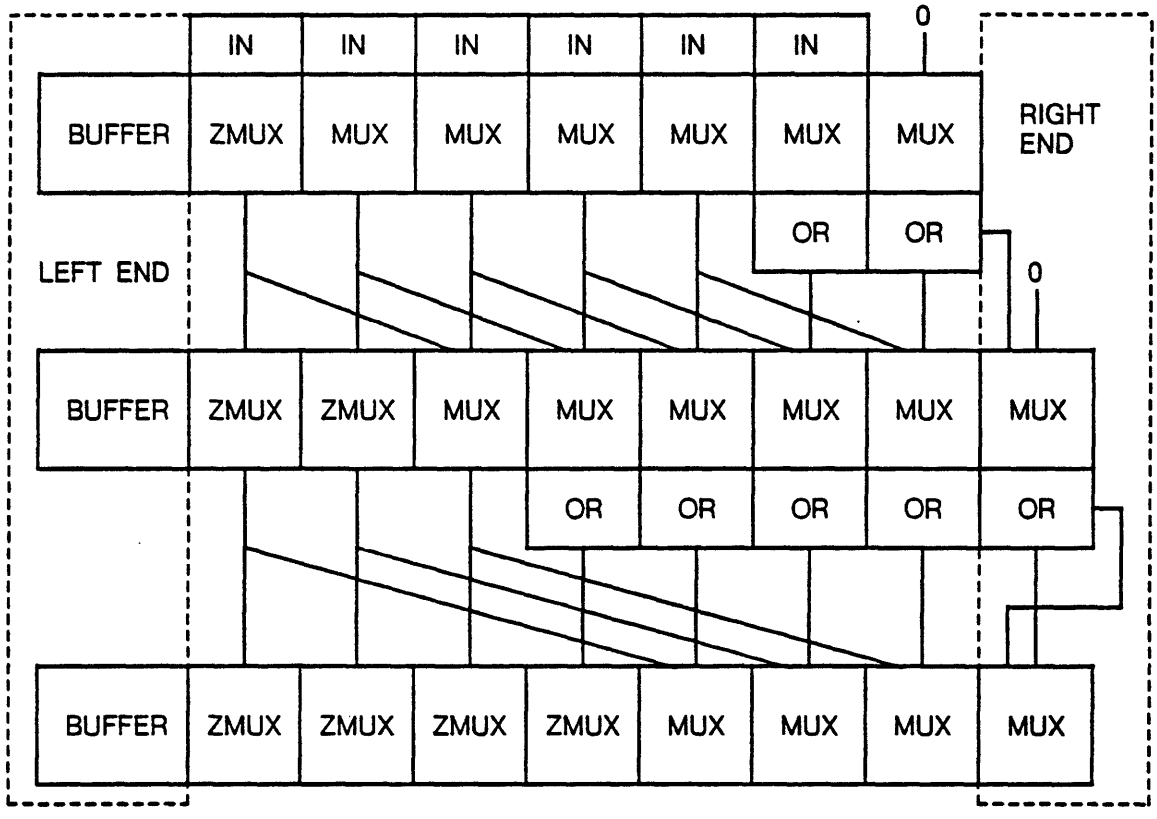


Figure 4-25: Floor plan of denormalizing shifter

The denormalizing shifter block is one of the most complex to construct. This is a result of the dependence of its topology on both N_e and N_f. Figure 4-25 shows the cell organization in a

denormalizer block for $N_e = 3$ and $N_f = 5$ (cell overlaps are not shown). To simplify testing as much as possible, the interface with the function blocks above and below follows the simple function block model; the varying topology is hidden inside the block away from the external interfaces.

The block always has N_e rows of multiplexer cells. A fixed rule is used to generate each row of cells (the special case on the first row is discussed later). If n_r is the row number where the first row is numbered 0, then the following rule holds: from left to right, each row is composed of one control driver for the select signal, 2^{n_r} ZMUX cells which output the constant 0 when directed to shift, and $N_f + 3\text{-}2^{n_r}$ basic multiplexer (MUX) cells.

The interconnect between rows n_{r-1} and n_r is built using the following rules: the unshifted operand is carried between rows on vertical poly wire; the high-order $N_f + 2\text{-}2^{n_r}$ operand bits are carried through diagonal metal wiring to obtain a shift offset of 2^{n_r}; a set of $2^{n_r} + 1$ OR cells is attached to the output of the upper row to compute the "sticky" bit. The output of the OR chain is connected to the shifted input of the LSB MUX cell so that the "sticky" bit position receives the inclusive-or of all of the bits which get shifted to the right of the guard bit position.

The first row only has $N_f + 2$ multiplexer cells. The $N_f + 1$ IN cells connect the block to the $N_f + 1$-bit swap block above. The IN cells actually overlap to provide the 1-bit shift wiring for the first row of MUX cells. The LSB MUX cell in the first row receives a constant 0 to serve as the guard bit. The "sticky" bit MUX is omitted here since both of its inputs are always zero.

The end cells are designed to be usable with any size of shift block. The left end cell only contains power wiring and is stretched to fit the vertical height of the finished unit. The right end cell contains wiring to provide the constant inputs to the guard MUX cells as well as power wiring. The right end is stretched to fit in two steps, first to align it to the first and second rows, next to align its lower edge to the lower edge of the complete block.

4.6.6 A_f Negate

The A_f operand negate block is built from $N_f + 3$ multiplexer cells connected as exclusive-or gates. The left end cell contains a control buffer which amplifies the A negate control signal from the sign handling block.

4.6.7 Adder

The adder block is built from $N_f + 3$ full adder cells with a half adder cell as MSB to hold the A sign bit and result overflow bit. The left end cell contains the sign handling block which also connects the feedback wiring for the carry chain. The right end cell contains the other end of the feedback connection and the initial condition for the global propagate wire.

4.6.8 Exponent Pre-adjust

The exponent pre-adjust block is built from N_e half adder cells. The left end cell contains wiring to connect the carry output to the exception handling block.

4.6.9 Result Negate

The result negate block is built from $N_f + 4$ multiplexer cells in exclusive-or connection. The left end cell contains a buffer which amplifies the control signal from the sign handling block. The leftmost bit cell of the block is used by the sign logic to compute the true sign of the result. The other $N_f + 3$ bits are connected to the $N_f + 3$ high order bits of the adder output (the "sticky" bit does not participate in the negation.)

4.6.10 Leading Zero Count

The leading zero count block is built from $N_f + 3$ zero count cells with appropriate count output encoding in each cell. If $N_z \leq N_e$ then the encoding for a cell is simply the bit position of the cell (MSB = 0).

If $N_z > N_e$ then it is possible for the number of leading zeros to be greater than the largest possible exponent value (which would cause underflow regardless of the actual exponent value). In this case we only need to build a 2^{N_e}-bit zero count block connected to the high order bits of the result. If the most significant 1 in the result is below the LSB of the truncated count block, then the block signals a zero result which is the appropriate action for underflow. The uncounted lower order bit positions receive "dummy" zero count cells.

Figure 4-26 shows the cell organization in a zero count block for $N_f = 5$ and $N_e = 3$. Each bit cell is composed of a set of a header cell (HEADER), N_z output wiring cells (OUT), and N_z output pulldown cells (0 and 1). As in the denormalizer block, the end cells are stretched to fit the height of the complete count cell.

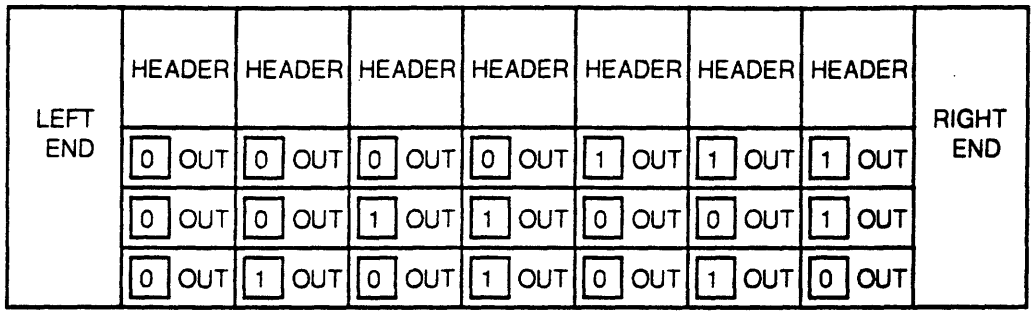


Figure 4-26: Floor plan of a zero count block

The left end cell provides some of the interconnect wiring for the count output. The right end cell only provides the connection to feed back the zero detect output to the left end cell. The set of clocked pullups for the count output wires is located in the output section of the LSB count cell in order to make use of what would otherwise be wasted space.

4.6.11 Result Normalize

The result normalize shift block is similar to the B_f denormalize block. It is built using N_z rows of $N_f + 2$ multiplexer cells. Note that there is no need to provide a column of shifters for the MSB of the raw result since we know that the final shifter output for that column is always a 1. In short, we omit the circuitry for the "hidden" bit.

Figure 4-27 shows the cell organization in a shift block for $N_z = 3$ and $N_f = 5$. The algorithm used to construct this block is the same as the denormalize shift block constructor except that the shift direction is reflected and the "sticky" bit circuits are omitted.

4.6.12 Exponent Adjust

The exponent adjust section is built from N_e adder cells. If $N_e > N_z$ then only the least significant N_z of the bit cells are full adders. The other $N_e - N_z$ bit cells are half adder cells. The left end cell contains the exception handling block.

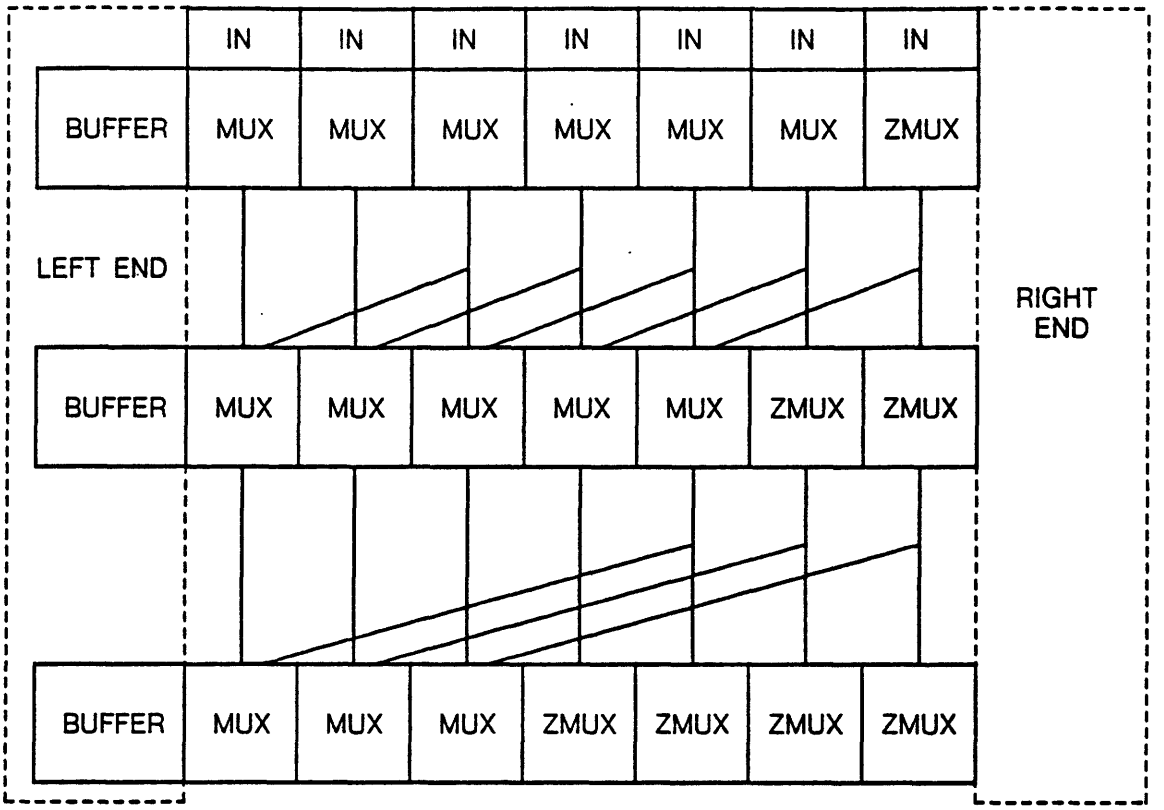


Figure 4-27: Floor plan of normalizing shifter

4.6.13 Output Buffer

The output buffer is built in two parts: an N_e-bit section for the result exponent and sign and an N_f-bit section for the result fraction. The left end cell of the exponent block contains a buffer for the control signal from the exponent handling block in the adjust block above. The right end cell of the exponent block contains wiring to connect the control inputs across to the left end cell of the fraction block. The left end cell of the fraction section contains the modified output buffer for the sign bit.

4.7 Assembling the Adder Procedurally

So far, our design methodology has produced a set of fixed-size cells (with a few "stretchable" cells) which must be assembled in orderly configurations according to the design parameters. This suggests the following approach to the design of the constructor program: build a library of building-block cells using an interactive graphic layout editor; then write a program to assemble the library cells into the complete adder unit, given the design parameters as input. This technique allows the designer to enter the basic cell definitions easily (encoding a detailed cell layout directly into a procedural layout description language is a time-consuming and error-prone process.) The constructor program itself only needs to manipulate the layout in an abstract fashion making it easier to write and debug.

The cell library includes all of the end cells, interconnect cells, bit cells, and special-purpose cells needed to build an adder of any size. The design of these cells was carried out by building a prototype layout of a 5-bit adder unit (2-bit exponent, 2-bit fraction, and sign bit). Figures 4-28 and 4-29 show simplified block diagrams of the prototype fraction and exponent sections of the adder, respectively (cell overlaps have been removed for clarity).

The prototype layout was created using an interactive graphic layout editor. Since almost all of the possible kinds of cell interaction are present in the prototype, design rule checking of the prototype during construction uncovered most design rule errors.

To enable the constructor program to assemble the library cells using the same amount of overlap used in the prototype, a set of *alignment points* was placed in each cell. For example, each data bit cell contains at least four alignment points; left, right, input, and output. The left and right points are used to align bit cells to each other as well as to the end cells. The input and output points are used to align the interconnect cells above and below. The alignment point technique makes it easy to visually verify that the connection between cells has been properly specified both in the prototype and in the constructed layout.

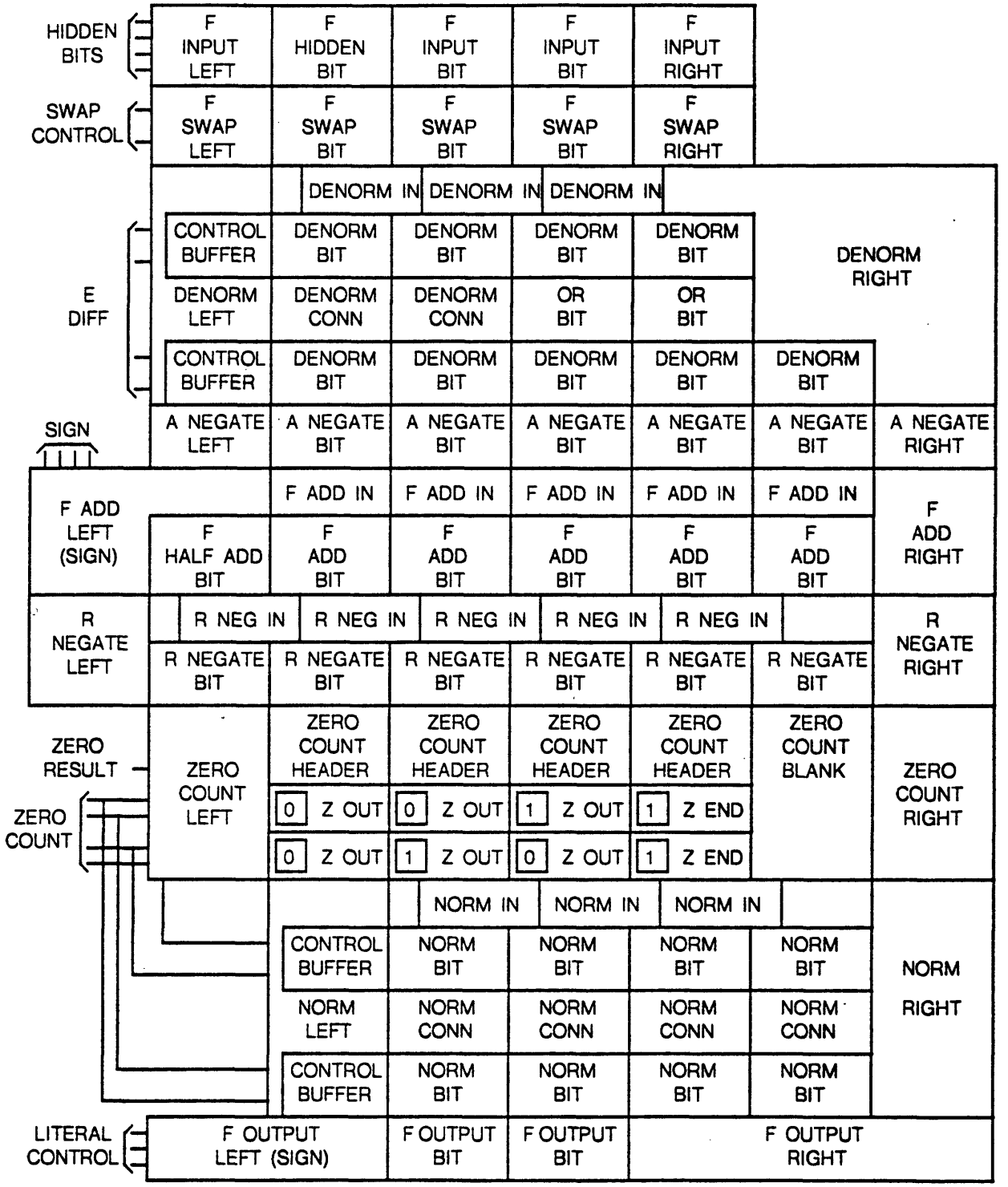


Figure 4-28: Prototype fraction section floor plan

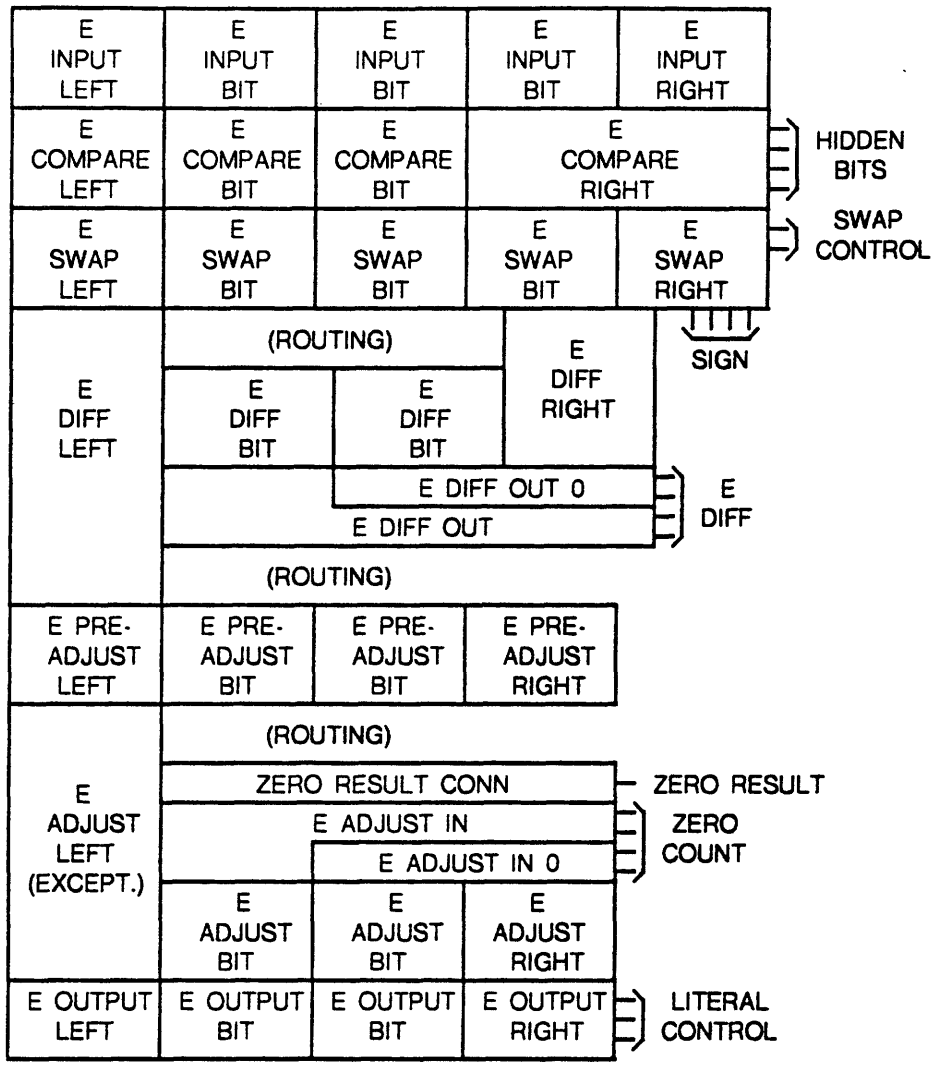


Figure 4-29: Prototype exponent section floor plan

4.7.1 Primitive Functions and Objects in the Constructor Program

The constructor program consists of a set of primitive functions for creating and manipulating layout objects along with specific code for constructing the desired functional unit (in this case, a floating-point adder). The primitive functions fall into several categories:

- Functions for converting between the internal data-base format and the standard external layout description file format used by the interactive editor and other programs.
- Functions for creating and positioning primitive objects.
- Functions for modifying existing objects.

The standard layout description file format is the one supported by the layout editor HPEDIT. This format supports a subset of the CIF standard format [10]. A description file consists of a set of *cell definitions* (CIF symbols). Each cell definition (or cell, for short) contains a set of records describing the objects contained in the cell. The following objects are supported:

- The *mask box* is the basic unit of fabrication mask material. The box is a rectangle with horizontal and vertical edges (only Manhattan geometry is supported). It is described by the location of two diagonally opposite corners (lower left and upper right) along with a mask layer name.
- The *label point* is used for alignment of cells and labelling of circuit nodes. It is described by a location, a name, and a mask layer.
- The *cell instance* (or simply *instance*) is a reference to a cell definition, similar to the CIF symbol call. It is described by the name of the cell it references, a transformation, and a name. The transformation consists of a rotation/reflection followed by a displacement vector. The rotation/reflection is one of the eight possible using Manhattan geometry (e.g., rotate 0, 90, 180, or 270 degrees, with or without reflection.) The instance represents a copy of the contents of the referenced cell, with the transformation applied to each object in the referenced cell. As in CIF, recursive instance references are disallowed.

This format can be used to hierarchically describe any Manhattan geometry layout.

All of the primitive objects in the layout file are supported directly by the constructor program. In addition, the program provides a *cell vector* (or simply *vector*) object for describing a one-dimensional array of cell instances in a compact form. A vector is described by a cell reference, a transformation to be applied to the cell before replication, a name, a replication displacement, a number of replications, and a starting index. The first three parameters are the same as for a simple instance.

The displacement is the offset between neighboring instances in the vector. This offset is arbitrary allowing any amount of overlap (or separation) between cells. Individual instances in the vector are referred to by index number. The starting index is the index of the first instance; the index number of the last instance is given by (start + length-1). This object useful for representing rows of data bit cells.

The primitive library reading function converts a layout file directly into a set of internal data structures. The list of cell definitions in the file becomes a linked list of cell objects in the internal structure. Each cell object has a set of pointers to linked lists of objects contained in the cell; one list for each kind of object. Each instance or vector object contains a pointer to the cell definition which it references. Figure 4-30 illustrates the basic data structures. The cell list is kept ordered so that vector and instance references always point to a cell further towards the root of the list. This simplifies recursion detection and allows easy conversion between internal and external formats since cell "forward references" are illegal in the external format.

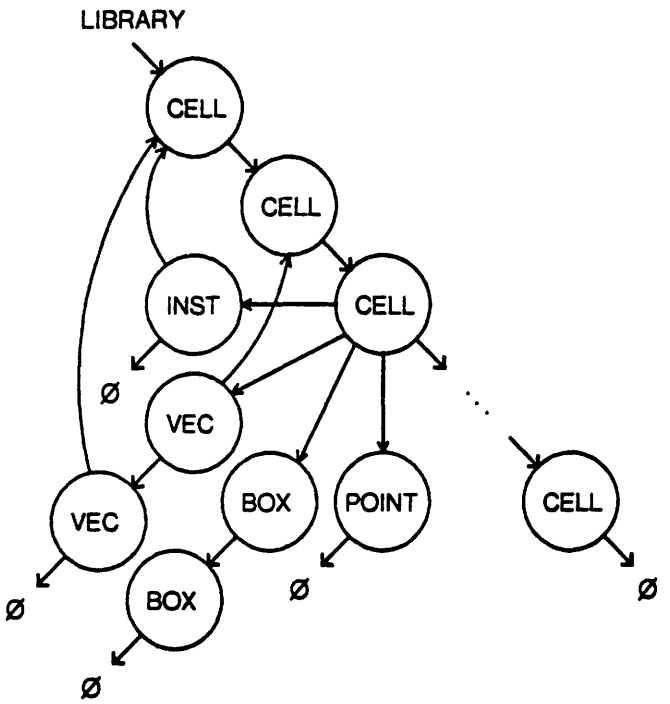


Figure 4-30: Cell Library Data Structures

The actual data objects which represent the primitive layout objects contain the same

information given in the layout file with the addition of linkage pointers (described above) and an object type field. The type field is not required to determine the object type in the data structures described above (though it is used for verifying the integrity of these structures). The primary purpose of the type field is to support data abstraction in the application dependent code. The primitive functions and the application dependent functions are contained in separate code modules with different environments. Code in the primitive environment is strongly typed (as in the data structure) while code in the application environment is relatively weakly typed. Primitive functions manipulate the various data objects directly. Application functions are only permitted to manipulate one abstract object type which is a union of the primitive object types. A primitive function called by the application code checks the type field of each object passed to it to dispatch to the proper section of code for that type of object (or to produce an error message if the function is undefined or illegal for that object type). This abstraction forces the application code to use only the primitive functions to manipulate the data structures thus insuring (insofar as the primitive code is error-free) that data structure integrity is maintained. The strong type checking in the primitive code module decreases the chance of errors while the application programmer is spared the nuisance of keeping track of all the data types.

The data abstraction at the application level also provides an opportunity to use of special abstract data objects which are never explicitly "seen" by the application code. Two such special objects are used within the primitive environment. These are described below along with the primitive functions which use them.

Several terms which are used throughout the discussion below need to be defined:

- A *point* or *location* is a simple coordinate pair object. A *label point* layout object is always referred to as a label point.
- The *bounding box* of an object is the (imaginary) box just large enough to enclose the object.
- A *feature* of an object is a descriptor of a location on the object. A feature descriptor can specify one of several points on the bounding box of an object (i.e., corners, center, or midpoints on the edges) or (if the object is an instance, cell, or vector) it can be the *path name* of a label point in the object.
- A *hierarchical object* is an object which can contain other objects, i.e., a cell, instance, or vector.
- A *path name* is name which uniquely identifies an object within the structure of a given

hierarchical object. A simple path name is the name of an instance, vector, or label point contained within the given object. Path names can be concatenated together to trace down through the hierarchical structure. For example, to identify a point named "foo" within an instance "bar" inside the given object, the path name "bar.foo" is used. To identify a point named "foo" inside the fourth instance in vector "bar", the path name "bar.4.foo" is used. For a path name to be valid, there must be exactly one object which can be identified by that name inside the given hierarchical object.

The following primitive functions are provided for use in the application environment:

- Create a new cell definition given a name. A cell of the same name must *not* already exist in the library. This new cell becomes the *current cell*; objects created by primitive functions are always added to the contents of the current cell.
- End the definition of the current cell. This function is always paired with the cell creator function. Cell creation can be nested.
- Find a cell definition given its name. This is used in creating an instance or vector of a library cell.
- Create a box given two diagonally opposite corner points and a mask layer. The new box is added to the current cell.
- Create a label point given a location, a mask layer, and a name.
- Create an instance of a cell given the cell, a reflection/rotation, a name (for the instance), an *alignment feature* inside the instance, and a location to which the given feature is to be aligned. The alignment feature describes a point within the cell definition to which the instance refers. It can be one of the corners of the bounding box of the cell or it can be the name of an label point inside the cell. The label point alignment is the most commonly used form.
- Create a vector of cell instances given the cell, a reflection/rotation, a name for the vector, an alignment feature inside the cell, a location at which the alignment point of the first instance in the vector is to be positioned, a length, a starting index, and a pair of *replication features* in the cell. The two replication features are used to align an instance in the vector to its neighbors.
- Return the location of a feature in a given object. The feature can be either a bounding box feature or the path name of a point in which case the location of the point is returned. This is used to find alignment locations.
- Move an object given a start point and an end point.
- Find an object inside a given hierarchical object given a path name for the target object. This is used to get a handle on an object for later (perhaps repetitive) reference. If the

path name refers to an object which appears at the top level of the hierarchical object then the function just returns a pointer to that object. If the target object is deeper in the hierarchical structure, then the function cannot simply return a pointer to it. For example, if the function finds an alignment label point within an instance and returns a pointer to the label point object in the cell definition for that instance, then the instance transformation information will be lost and the location of the returned object will not, in general, be the same as the location of the path named label point! In such cases, the function actually returns a special type of object called a *closure*. A closure object is nearly identical to an instance object. The closure contains a pointer to the found object (which need not be a cell definition) along with the transformation required to keep the object in the proper position and orientation. This transformation is the composition of the transformations encountered while tracing down through the hierarchical structure to find the target object.

In addition to the simple functions above, there are two powerful functions which require more elaborate description. These are the stretch function and the router function. The stretch function is used to adjust the size of stretchable cells such as bus buffers and end cells of variable height blocks. It is a powerful, general-purpose function. Its parameters are: a handle on the object to be stretched, a pair of points on a *fracture line* which divides the target object into two sections, and a pair of points describing the stretch displacement. The only restrictions on the parameters are that the stretch displacement must be either horizontal or vertical and must not be parallel to the fracture line. The fracture line can have any orientation, though it is usually perpendicular to the displacement.

The operation of the stretch function is simple: the function determines on which side of the fracture line the displacement points lie, then moves all objects on that side of the fracture line by the displacement amount; boxes with two corners on each side get stretched. The powerful feature of this function is that it operates properly in the presence of hierarchical structure in the stretched object; if the fracture line divides an instance within the target object, then the following steps are taken:

- A duplicate is made of the contents of that instance's cell definition.
- The stretch parameters are mapped into the duplicate cell's coordinate system using the inverse of the instance's transformation.
- The stretch function is recursively applied to the duplicate cell.
- Finally, the instance is modified to reference the duplicate cell.

This allows the stretch function to stretch all boxes in the target object as if there were no hierarchical structure without affecting unrelated instances of the cells appearing the structure of the target cell!

The obvious disadvantage of this method is that can result in the generation of large number of duplicate cells. To ameliorate this problem, the stretch function makes use of a special data structure, the *stretch* object. When a duplicate cell is created and stretched, the function appends a new stretch object to a linked list of these objects attached to the parent cell. This object contains a pointer to the stretched duplicate and a canonic representation of the stretch parameters used. When the function is called upon to stretch a cell, it first checks that cell's stretch object list to see if the required stretch has already been done and, if so, immediately returns a pointer to the existing stretched duplicate. This minimizes the number of duplicate cells required.

The last primitive function is actually an application level function in that it only uses the abstract primitive functions. This is the simple router function. It was originally implemented as an application function to do the routing inside the shift block but it turned out to be useful for all of the routing tasks which could not done with library cells. The function of the router is straightforward; it does a minimum-height single-layer routing between contact regions on two parallel edges of a rectangular channel. It is given two lists of contact points (one for each channel edge), a mask layer, the minimum mask width, the minimum mask spacing, a source object, a destination object, and a pair of points on a fracture line between the source and destination. The contact points are expected to lie along edges in the source and destination objects. The source and destination object should be initially positioned at the closest permissible separation. The router then constructs a cell containing the wiring to connect the two edges in the minimum possible distance. If the channel is too large for the wiring, then the router extends the wiring to fit. If the channel is too small, then the router uses the stretch primitive with the given fracture line to stretch the destination object to allow the wiring cell to fit. The stretch feature is provided so that existing connections between the source and destination objects are preserved.

The functions described above constitute the core of the primitive functions. A typical usage of these functions is to write an application function for constructing a new cell built out of library cells. The canonical form of such a function is as follows:

- The application function receives a set of parameters controlling the construction of the desired cell, say, the number of bits in a data bus.
- By convention, the name of the new function should be constructed from a base name, which is unique for each constructor function, plus a copy of each of the control parameters. For example, a cell of type "foo" with data path width of 6 could be named "foo-6". The function first builds the name of the desired cell, then checks to see if a cell

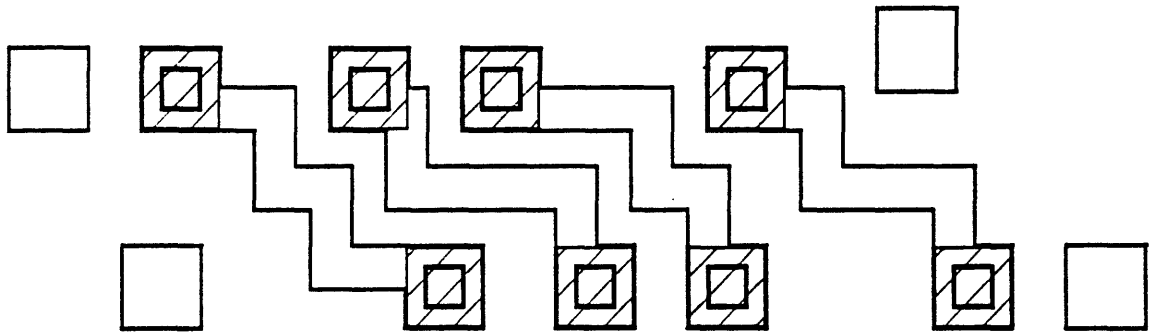


Figure 4-31: Example of Routing Primitive

with that name already exists before it attempts to construct a new one. This keeps the function from building a duplicate of an existing cell. Of course, the the constructor function must be a *function* in the formal sense; any given set of parameters in the range of the function must map to exactly one cell layout in the domain of the function.

- If the cell needs to be built, then the cell creator primitive is called with the new name. This suspends the creation of the current cell and makes the new cell the new current cell.
- The constructor then uses the primitive functions to find and assemble library cell in the proper configuration. The function can also call other constructor functions since the cell creation process is recursive.
- The end cell definition primitive is called after all objects have been added to the new cell. This restores the previous current cell (if any) to active status.
- Finally, the constructor returns a pointer to the new cell to its caller.

All of the algorithms described in the previous section are implemented using the form described above. The overall form of the adder constructor is described below.

4.7.2 Application Functions for the Adder

The application code in the adder constructor program consists of: a top-level function which receives the user specified parameters; a cell constructor for the complete adder cell; a cell constructor for the fraction section; a cell constructor for the exponent section; and a set of cell constructors which implement the individual function block construction algorithms described in earlier sections.

The top-level function builds an adder layout using the following process:

- Obtain the design parameters from the user (exponent width, fraction width).
- Load the library of basic cells into the internal database.
- Call the adder constructor function with the user given parameters.
- Write out the completed layout to permanent storage.

The functions for building and assembling the function blocks follow closely the algorithms described in previous sections. The only functions left to specify are the three top-level constructors. The fraction constructor is straightforward and assembles the fraction section from top to bottom, calling each block constructor in turn to assemble the next lower function block. The adder constructor simply calls the fraction constructor and exponent constructor, then abuts instances of the resulting cells using alignment points in the cells to build the complete adder unit. The exponent constructor is similar in form to the fraction constructor except that it also contains the code for generating the interconnect between the sections.

Since the exponent constructor needs to find alignment points in the fraction section, it obtains a handle on the fraction cell (it assumes that the fraction cell has already been built). It then orients its function blocks and wiring relative to the alignment information obtained via the handle. The fraction cell itself does not actually appear in the exponent section. It is only used as a reference for constructing the exponent section.

Since the input blocks (input buffer, exponent comparator, swap blocks) do not vary in height, the hidden bit wiring and swap control connections are contained in the appropriate function block end cells. The two sections are initially positioned by abutting the input blocks. The routing function is then used to perform the exponent difference to denormalize and zero-count to exponent adjust wiring. If there is not enough room for the router to operate, then the two sections are moved apart and the existing wiring stretched. The wiring between the two sections of the output buffer is done by

stretching wiring in the right end cell of the exponent output section to connect to the left end cell in the fraction output section.

The verification of the proper operation of the constructor program was carried out by examining four key adder configurations during the design process ([N_e, N_f] pairs): [3,4], [3,10], [4,4], and [4,10]. Together, these four examples exercised all of the special case handling code in the constructor program. Figure 4-32 shows the layout of an adder with 4-bit exponent and 10-bit fraction.

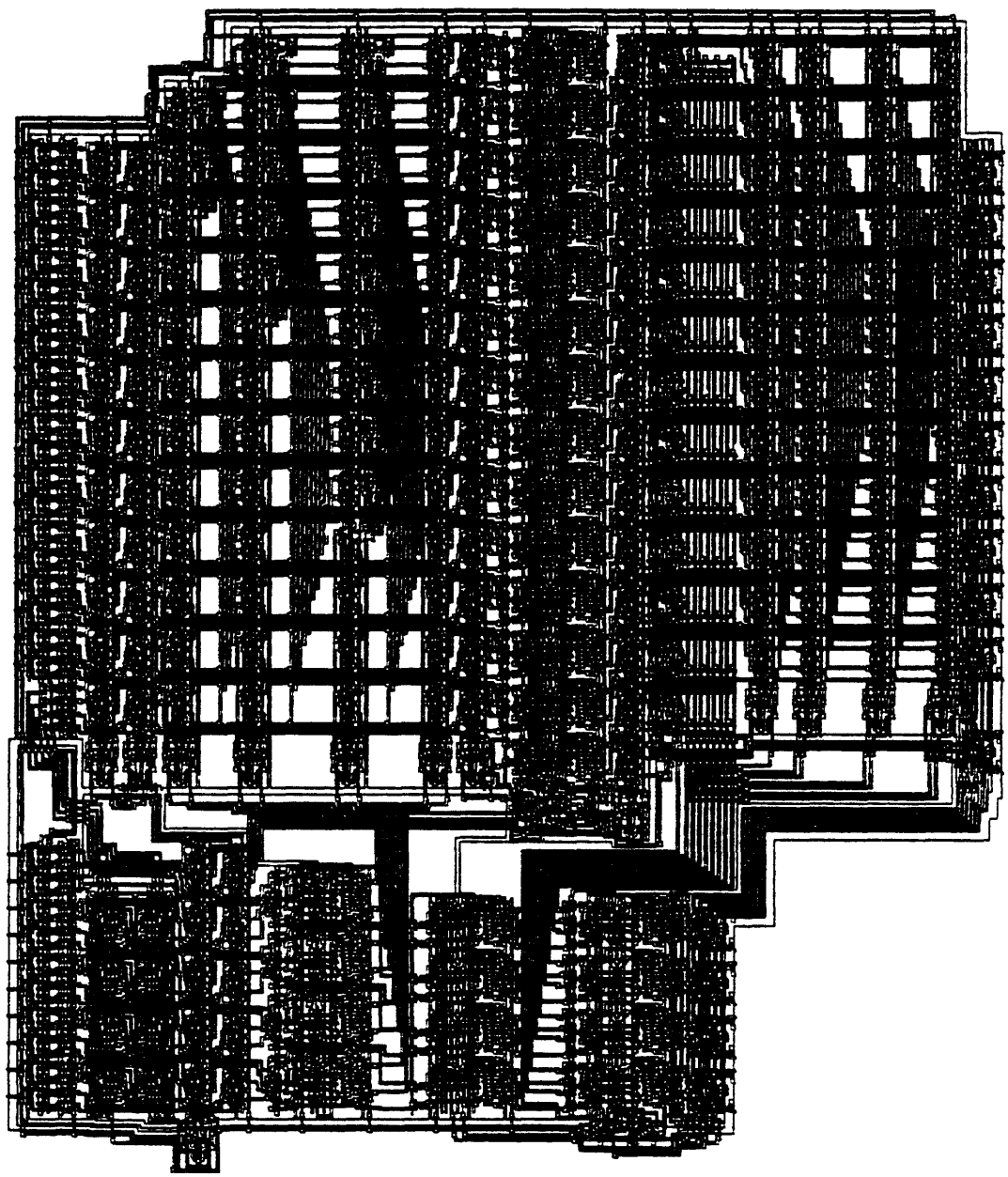


Figure 4-32: Layout of 15-bit floating point adder

CHAPTER FIVE

Implementation of the Multiplier

In this chapter we carry out the high-level design of a floating-point multiplier unit using the same methodologies applied to the adder in the previous chapter. As before, the first step in this process is to determine the exact requirements for the multiplier function blocks.

5.1 Functional Specification of Multiplier Logic

Figure 5-1 shows the detailed structure of the floating-point multiplication algorithm. The inputs are the operands A and B each of which consists of an N_e-bit exponent, A_e, B_e; a sign bit, A_s, B_s; and an N_f-bit fraction, A_f, B_f.

The detailed sequence of operations is as follows:

1. If either operand is zero then immediately output a zero result.
2. Append a 1 above the MSB's of A_f and B_f. This operation recovers the "hidden" bit in the normalized numbers. The fractions now have $N_f + 1$ bits each.
3. Multiply the two $N_f + 1$ bit fractions together to get the $2N_f + 2$ bit unsigned product R''_f (only the most significant $N_f + 2$ bits of R''_f are actually used).
4. Add A_e to B_e to obtain the raw result exponent R''_e.
5. Examine the MSB of R''_f. If this bit is a 1 then select the N_f next most significant bits of R''_f as the normalized result R'_f. If this bit is a 0 then the next most significant bit of R''_f must be a 1 (since the raw result is in the range 1.0 to 4.0). In this case, drop the two most significant bits of R''_f and select the N_f most significant bits remaining as R'_f.
6. If the MSB of R''_f is 1 then increment R''_e to obtain the result exponent R'_e.
7. Compute the result sign as $R'_s = A_s \oplus B_s$.
8. If the result exponent is zero or the exponent addition underflowed, then output a zero result. If the exponent addition or the normalization adjustment overflowed, then output the overflow constant retaining the sign of the result. If there are no exceptional conditions then pass the values R'_e, R'_s, and R'_f through as the final result R_e, R_s, R_f.

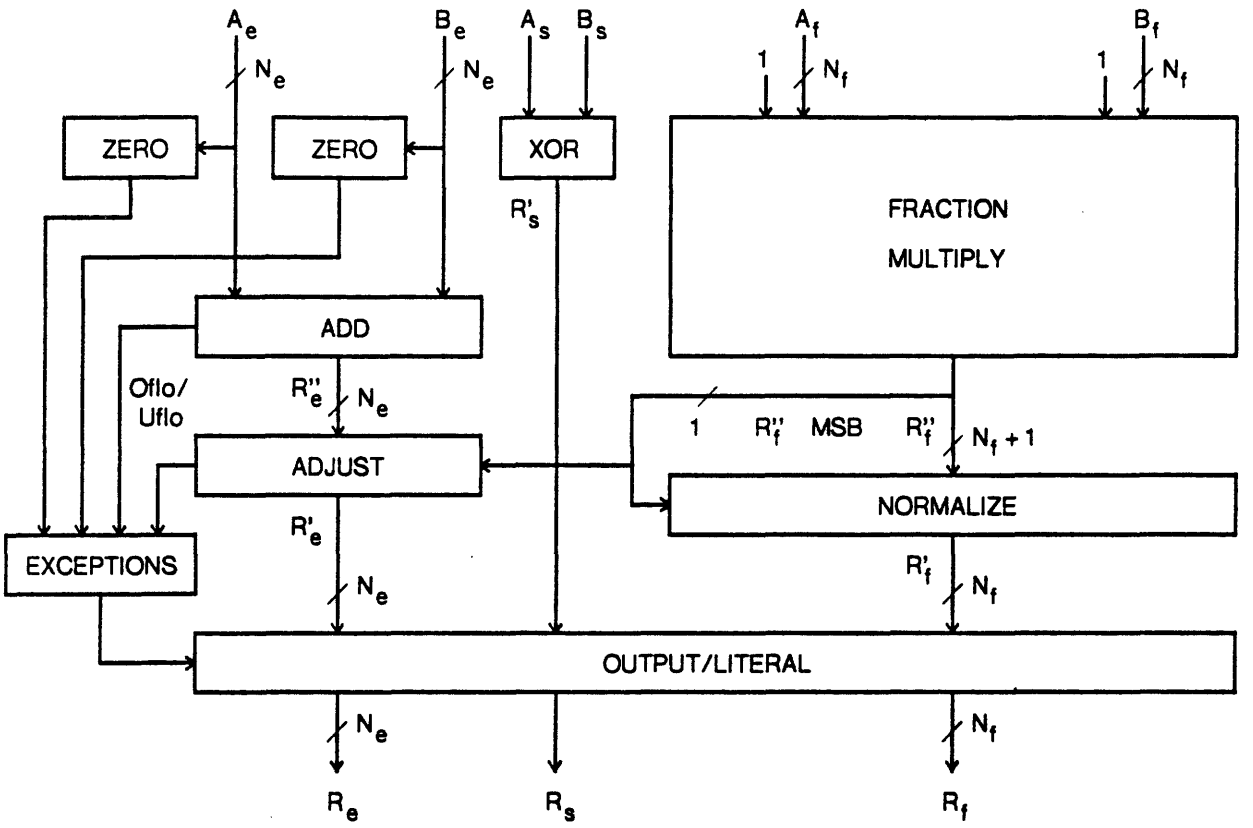


Figure 5-1: Floating-point multiplication algorithm

5.2 Specification of Function Block Forms

Figure 5-2 shows the multiplier floor plan described below. All but one of the function blocks in the multiplier are similar to blocks we have already seen in the adder. The unfamiliar kind of block is the fraction multiplier which dominates the floor plan of the multiplier unit. We will consider this block first.

The fraction multiplier is fundamentally an integer multiplier. Considerable study has been made of the problem of constructing high-speed integer multipliers. For the purposes of this discussion we assume that a multiplier generation procedure is available. Particularly well-suited for this purpose is the Baltus multiplier generator [13] which constructs a variable-size fraction multiplier based on a fixed-size prototype designed by Evans [14].

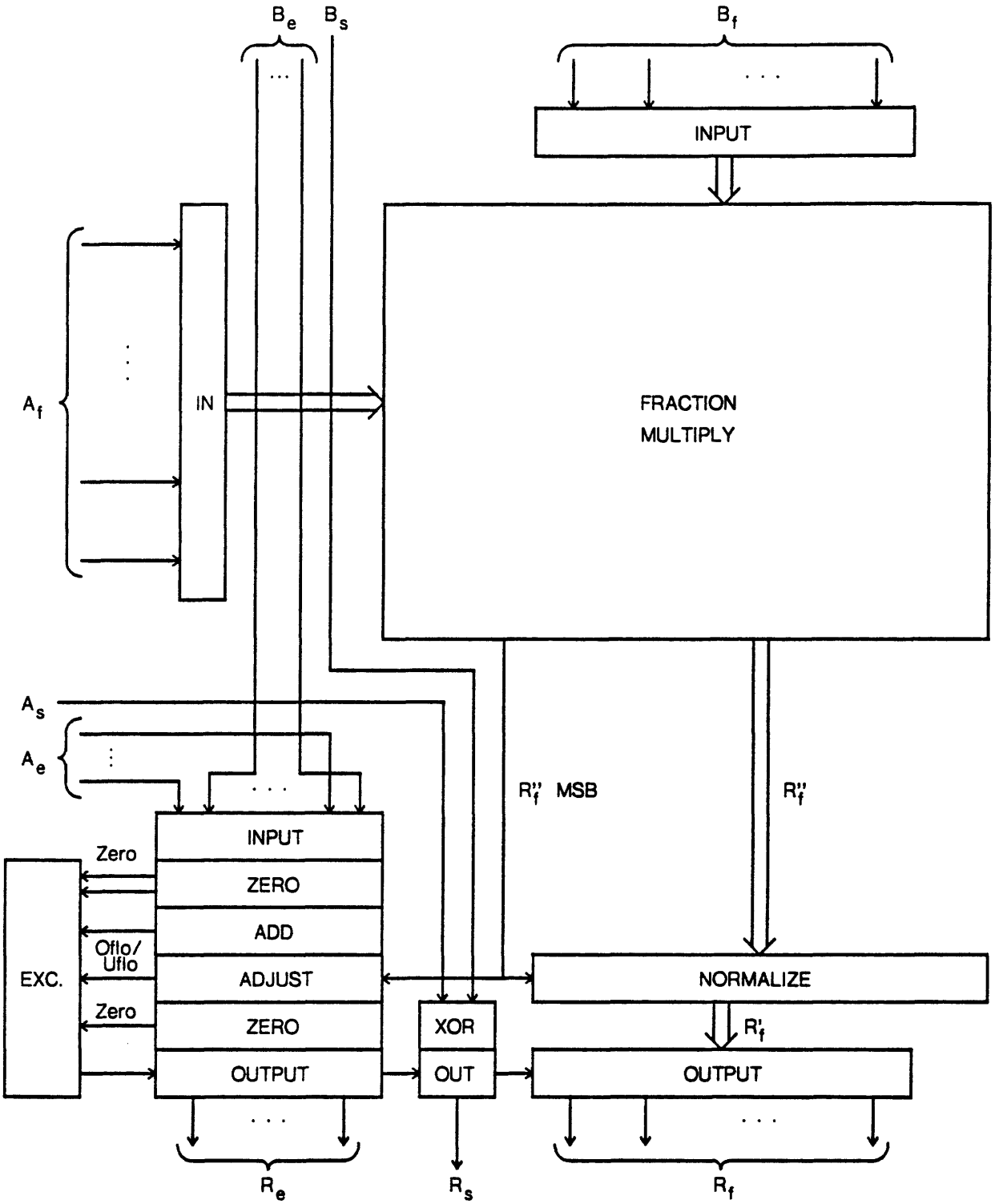


Figure 5-2: Initial floor plan of multiplier layout

This generator constructs a multiplier array using modified Booth recoding to minimize the size of the internal adder cell array and obtain 2's complement operation. Since our floating-point multiplier uses sign-magnitude representation, the boundary conditions on the multiplier array in the Baltus generator must be modified to accomodate unsigned operands. In order to discuss the modifications, we need to review the recoding technique used in this multiplier.

5.3 Modified Booth Recoding

In order to compute the product of two N-bit integers, we could use a set of N rows of N-bit adders where each row computes the partial product of one of the A operand bits with the B operand and adds this to the total product. Thus each row performs the operation $P_{i+1} = P_i + B \cdot A_i \cdot 2^i$. If we consider the rows i and i + 1 together, they perform the operation of adding 0, 1, 2, or 3 times the B operand to the total product depending on the states of A_i and A_{i+1} (multiplying the partial product by the binary significance of row i). We can reduce the complexity of the array by observing that the operations required of a row pair can be modifed to eliminate the need for adding 3 times the B operand. The modified Booth recoding technique changes the required set of operations for a row pair from {+0X, +1X, +2X, +3X} to {-2X, -1X, +0X, +1X, +2X}. Multiplication by 2 can be done by shifting and subtraction can be done by complementing the B operand and setting the carry input of the current row to 1 (using 2's complement arithmetic). Thus the row pair can be replaced by a single N + 1-bit adder row with a multiplexer on its B input to select the proper value for each operation. The overall size of the array is then reduced to N/2 rows of N + 1-bit adders with input multiplexers.

Two logical signals are used to control the operation of a row. These are S, the operation sign, and F, the operation factor which have possible terminal states of {ADD, SUB} and {0X, 1X, 2X} respectively. Table 5-1 shows how the row control signals are derived from the states of bits A_{i-1}, A_i, and A_{i+1} (for row pair [i,i + 1]). This particular *recoding* of the A operand bits is known as the modified Booth recoding. In addition to reducing the size of the array, this recoding results in both operands and product being correct when interpreted as signed 2's complement values. Rubenfield has presented a formal proof of this technique [15].

A_{i+1}	A_i	A_{i-1}	S	F
0	0	0	ADD	0X
0	0	1	ADD	1X
0	1	0	ADD	1X
0	1	1	ADD	2X
1	0	0	SUB	2X
1	0	1	SUB	1X
1	1	0	SUB	1X
1	1	1	SUB	0X

Table 5-1: Modified Booth recoding

5.4 Designing the Fraction Multiplier Prototype

The 2's complement operation of Booth recoded multipliers is normally a benefit; in our case, however, we want unsigned arithmetic for the fraction multiplier. This is obtained by adding an explicit zero (positive) sign bit to each operand. In addition, the "hidden" fraction MSB bit must be included in each operand. Given N_f-bit fraction operands, the raw size of the multiplier array is then $(N_f + 2)/2$ rows of $N_f + 2$ bit cells. Figure 5-3 shows the complete logical structure of a protoype 6 by 2 bit multiplier. This size was selected to show all types of cell interaction; normally, the number of bits in the A and B operands would be equal.

Figure 5-4 shows the floor plan of the layout cells corresponding to the logical diagram in figure 5-3. The logical function of each cell is described below.

The fundamental cell in the array is the adder/multiplexer cell (ARRAY BIT). This is composed of three modules: The 0/1/2X module is a multiplexer controlled by the F_i signal which selects either the constant zero, the left input (B operand), or the right input (2•B operand). The 0/1/2X module's output is conected to the data input of the + /- module which selects whether or not to complement the data under control of the S_i signal. The CSA (carry-save adder) module is a full adder.

Surrounding the core of array bit cells is a set of boundary cells. There are five categories of boundary cell:

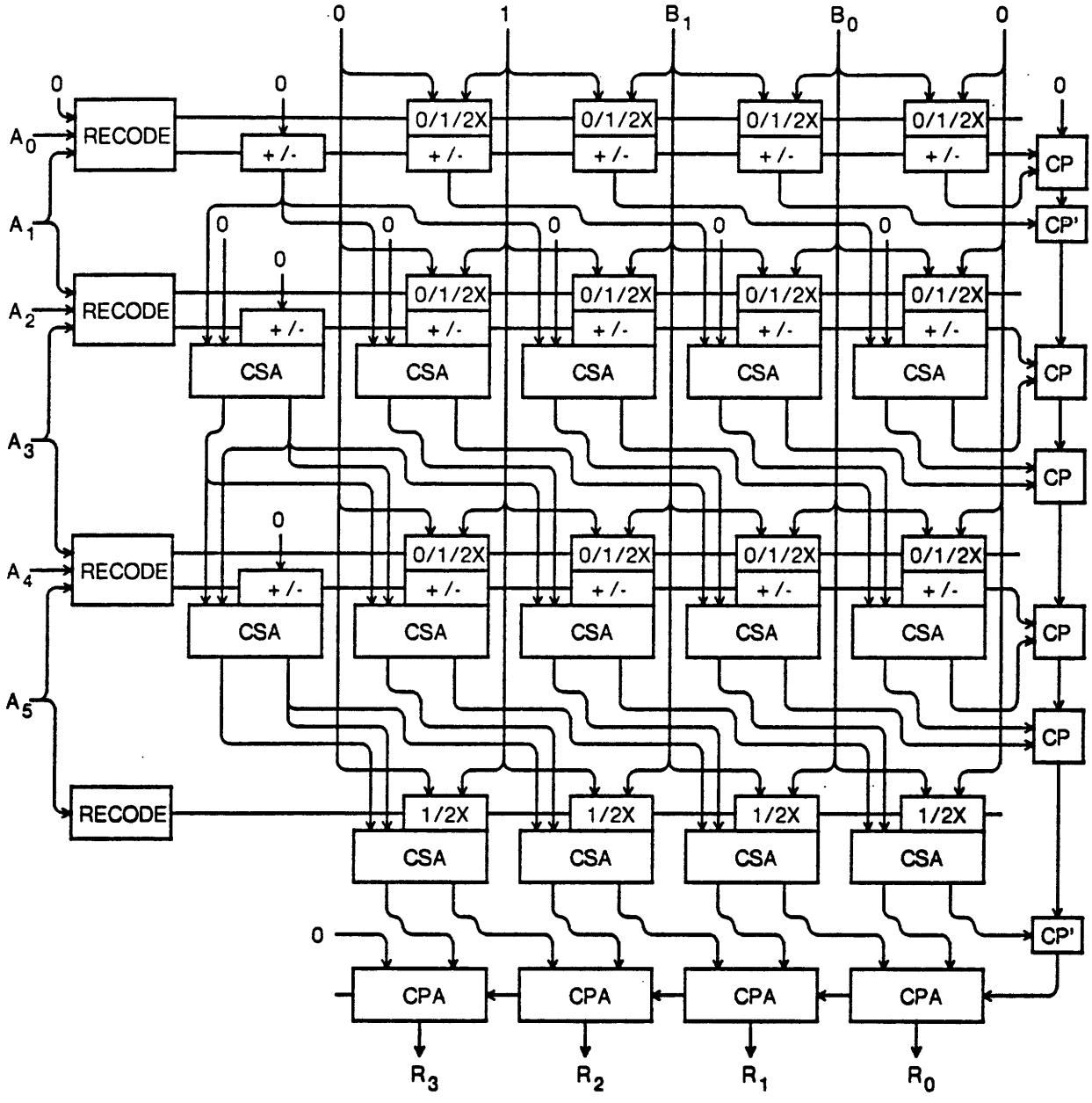


Figure 5-3: Logical diagram of prototype fraction multiplier

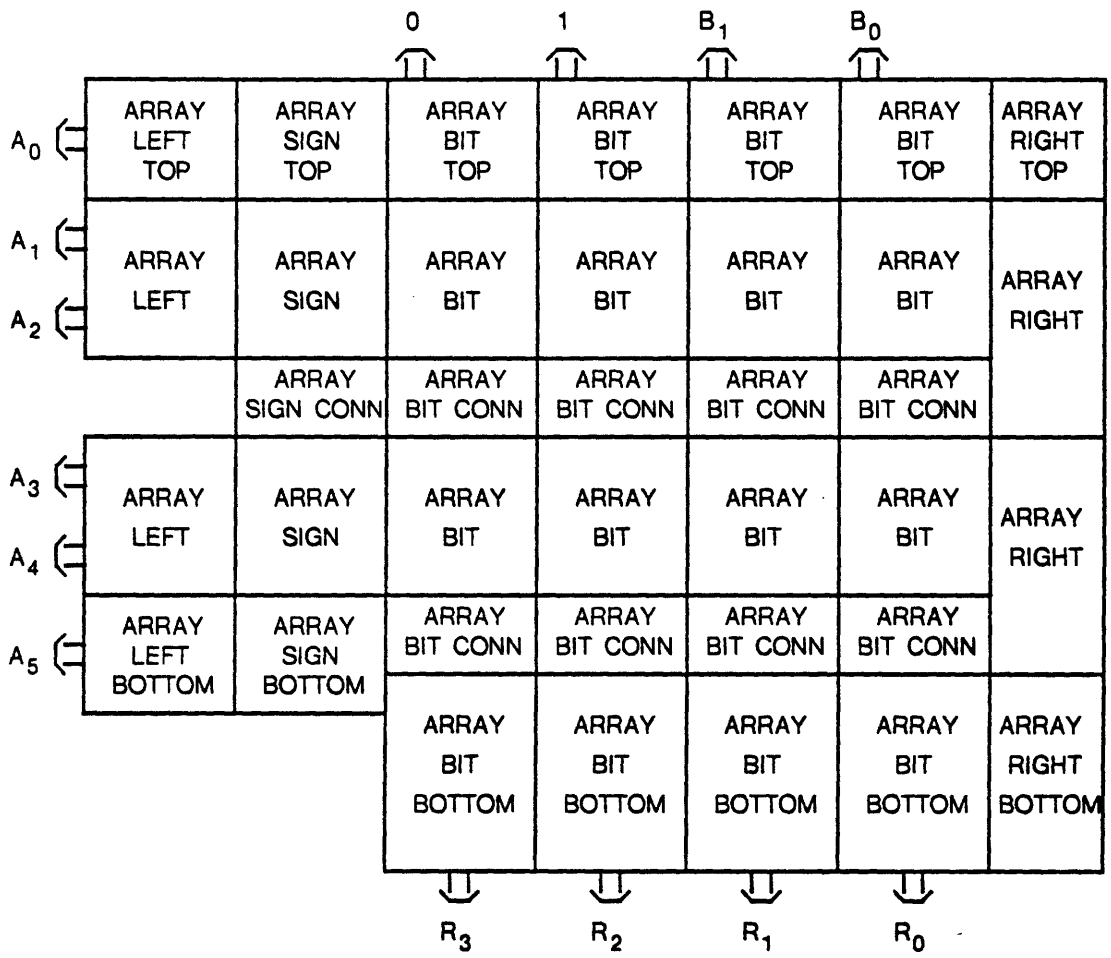


Figure 5-4: Floor plan of prototype fraction multiplier

- The top row of the array requires no CSA modules since there is not yet any accumulated product hence the top row cells contain only the multiplexer modules.
- The leftmost column of bit cells (the sign bit column) has a constant zero B input (since the sign of B is always positive) hence the 0/1/2X module is omitted from these cells. The lowermost cell is omitted since its output is always zero.
- The leftmost column of cells generates the set of S and F recoded signals from the A input operand. The two most significant bits of the A operand are always 01 as in the B operand. The lowermost RECODE module (ARRAY LEFT BOTTOM) is simplified to take this into account. Since this module only implements the second and fourth rows of table 5-1, the multiplexers in the bottom row of the array can be simplified to provide only for 1X and 2X values of B. In the case of N_f odd (e.g. $N_f = 5$), the lowermost recode module

and multiplexers are omitted entirely since the lowest row will always add B to the result product.

- The bottom row of adder/mux modules includes a set of carry-propagate modules (CPA) to compute the final result. The CPA modules are full adders like the CSA modules except that they are optimized for maximum carry propagation speed.
- The rightmost column of modules computes the carry from the least significant result bits. This is required for effective infinite precision arithmetic before result truncation. The CP module is a carry-propagate adder module which has had its sum bit cicuitry removed and hence only computes the carry. The CP' module is a half adder of the same form as the CP module.

5.5 Multiplier Function Blocks

Like many array multiplier forms, the Baltus-Evans multiplier accepts the input operands along two adjacent edges and produces the double length result along the remaining two edges. The B operand on the top edge has its MSB at the left. The A operand on the left edge has its MSB at the bottom. The result MSB is at the bottom left corner. This orientation matches well with the adder bus orientations. The option of routing the A input wiring to the top edge to match the adder topology is left to the end user of the multiplier layout. The exponent and sign wiring is arranged to match the adder topology as well as possible.

The input driver blocks are similar to those in the adder unit. The exponent section input cells are identical to the adder unit input cells. The fraction section input cells are sized to fit the pitch of the cells internal to the multiplier block.

The zero detect blocks in the exponent section can be built using the "sticky" bit OR cell from the adder denormalizer block.

The exponent add and adjust blocks can use the full adder and half adder cells used in the adder unit. Care must be taken to make sure that the exponent bias of $2^{N_e-1} - 1$ is taken into account in the exponent arithmetic. For example, if $N_e = 4$, then the exponent addition must be offset to obey $0111_2 + 0111_2 = 0111_2$ ($2^0 2^0 = 2^0$). This can be done by complementing the MSB's of each exponent before addition and setting the carry input of the exponent adder to 1. The MSB of the output of the exponent adjust block must then be complemented before the test for zero result exponent. This encoding allows the carry outputs of the exponent arthmetic blocks to be used directly for overflow and underflow detection as in the adder unit.

The normalize block can be built using the multiplexer cell from the adder unit, sized to fit the cell pitch in the output section of the multiplier block. The two output blocks can be constructed using the same set of cells as in the adder unit.

The sign and exception handling blocks must, of course, be specially designed for the multiplier unit since their logical structure is much different from the structure of the corresponding blocks in the adder unit.

CHAPTER SIX

Conclusions

This research revealed that an important part of the creation of a procedure for building VLSI layout is providing for the verification of the correctness of all layouts that the procedure is capable of generating without actually having to construct and test each one. This verifiability was obtained by adherence to design methodologies at all levels of abstraction, nameley: the functional level, the logical network level, the physical circuit level, and the layout level. At each level a set of rules for constructing well-formed and/or well-behaved structures was given; each set of rules being divided into two classes: rules for guaranteeing correctness of individual modules and rules for combining correct modules into larger modules which are also correct. The particular set of rules used in this thesis are largely technology independent and can be adapted for use in a variety of design environments.

At the functional level, the primary rule was to use linear structures to implement the function blocks in the top-level descriptions of the arithmetic algortihms. These structures mapped into the lower levels of abstraction so as to reduce the verification of structures with variable bus widths to simple induction on a small set of test cases. The price of placing this kind of restriction on design is the exclusion of tree-structured circuit forms which can achieve O(log N) performance in certain cases where the performance of the linear structure is O(N) or worse.

At the network and circuit level, a set of rules was provided which guarantee that a precharged combinatorial network is free of critical races. A simulation program was developed to check adherence to the rules. This methodology was presented at several levels of generality so that it can be adapted to technologies other than the NMOS technology used in this thesis. The cost of using this particular methodology is the area overhead required to implement logical signals as multi-wire physical signals which seems to run between 50 and 75 percent increase over a conventional circuit.

At the layout level, conventional geometric design rules were combined with the linear structure rules from the functional level to obtain a set of rules for guaranteeing the geometric correctness of variable bus width structures. Circuit rules were checked at this level by extracting the circuit description of a layout and applying the circuit level verification program.

6.1 Improvements

There are a few areas which this research did not explore which would be of interest:

- The combinatorial arithmetic units described above could be split into sequential sections (i.e. pipelined) to achieve greater throughput at the expense of total time to compute a single result. The combination of pipelining with the self-timed combinatorial methodology could be investigated.
- There probably exists a set of rules for constructing variable-size tree-structured layout forms. This topic could be investigated more vigorously.
- It was fortunate that the simple routing procedure was able to do all of the routing tasks in the adder unit. A more robust system for doing procedural design would need to have available a powerful routing mechanism for handling wiring which cannot be efficiently done by abutting fixed cells.

References

[1] Intel Corp., *Intel Component Data Catalog*, 1981.

[2] Weitek Corporation, *WTL1032/33 Preliminary Data Sheet*, 1983.

[3] F. Ware and W. McAllister, "CMOS Chip Set Streamlines Floating-point Processing," *Electronics*, Vol. Volume 55-3, Feb. 10 1982, pp. 149-152.

[4] "A Proposed Standard for Binary Floating-Point Arithmetic," *Computer*, Vol. 14, March 1981, pp. 51-62.

[5] P. M. Ebert, J. E. Mazo, and M. C. Taylor, "Overflow Oscillations in Digital Filters," *Bell System Technical Journal*, Vol. 48, 1969, pp. 2999-3020.

[6] J. T. Coonen, "Underflow and the Denormalized Numbers," *Computer*, Vol. 14, March 1981, pp. 75-87.

[7] D. Hough, "Applications of the Proposed IEEE 754 Standard for Floating-Point Arithmetic," *Computer*, Vol. 14, March 1981, pp. 70-74.

[8] A. V. Oppenheim and C. J. Weinstein, "Effects of Finite Register Length in Digital Filtering and the Fast Fourier Transform," *Proceedings of the IEEE*, August 1972, pp. 335-354.

[9] R. T. Masumoto, "A 16 Bit LSI Digital Multiplier," Master's thesis, California Institute of Technology, 1978.

[10] C. Mead and L. Conway, *Introduction to VLSI systems*, Addison-Wesley, 1980.

[11] C. J. Terman, *User's Guide to NET, PRESIM, and RNL*, Massachusetts Institute of Technology, 1982.

[12] S. P. Mccormick, "Automated Circuit Extraction from Mask Descriptions of MOS Networks," Master's thesis, Massachusetts Institute of Technology, 1984.

[13] D. G. Baltus, "Design of an Assembler of NMOS Fast Parallel Fractional Multipliers," Bachelor's thesis, Massachusetts Institute of Technology, 1983.

[14] W. H. Evans, "An MOS LSI Digital Signal Processor for Speech Synthesis Applications," Master's thesis, Massachusetts Institute of Technology, 1982.

[15] L. P. Rubenfield, "A Proof of the Modified Booth's Algorithm for Multiplication," *IEEE Transaction on Computers*, Vol. C-24, October 1975, pp. 1014-5.