NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

CMU-CS-81-105

Testing and Debugging
Custom Integrated Circuits

Edward H. Frank
Robert F. Sproull

Department of Computer Science
Carnegie -Mellon University
Pittsburgh, PA 15213

February 1981

Abstract

Although designing custom integrated circuits does not seem to be any harder than writing computer programs, debugging integrated circuit designs is much more cumbersome than testing and debugging programs. In this paper we examine some of the traditional approaches to testing and debugging integrated circuits and then describe how access to internal state, and simulation can make testing and debugging custom IC designs at least as easy as testing and debugging software.

Copyright © 1981 Edward Frank and Robert Sproull

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Table of Contents

1. Introduction 1
2. Terminology 2
    2.1. Testing and debugging 2
    2.2. Integrated circuits 2
    2.3. Integrated circuit designs 2
    2.4. Statefull and stateless chips 3
3. A comparison of techniques for testing and debugging programs and integrated circuits 4
    3.1. Checking syntax 4
    3.2. Checking semantics 6
        3.2.1. Designing to reduce errors 9
    3.3. Checking performance 9
4. Current techniques for testing and debugging integrated circuits 10
    4.1. Testing and debugging IC designs 10
    4.2. Testing and debugging production IC's 15
        4.2.1. Testing inputs and outputs 15
        4.2.2. Accessing internal state 17
5. Future trends in IC debugging and testing 20
    5.1. Designing to reduce errors 20
        5.1.1. Microcode 20
        5.1.2. The CMU Design Automation System 24
    5.2. Design for testability and debuggability 24
        5.2.1. In-situ testing 25
        5.2.2. Muffler extensions 27
        5.2.3. On-chip signature analysis 29
        5.2.4. On-chip or on-wafer testers 30
        5.2.5. Summary of access to internal state 30
    5.3. Organizing and using a debugging system 31
        5.3.1. Names 31
        5.3.2. Organizing simulations 32
        5.3.3. Debugging strategies 33
        5.3.4. Debugging at CMU 33
6. Conclusions 34

1. Introduction

The revolution heralded by the book *Introduction to VLSI Systems* [Mead 80] has made it possible for designers with little previous experience in hardware or integrated circuit design to create complex custom integrated circuits in a matter of weeks or months. Moreover, the multiproject-chip (MPC) technique that fabricates many designs together on the same wafer to reduce fabrication costs has been very successful, returning packaged chips to designers as soon as a month after the final design is submitted [Conway 80].

Although designing this sort of custom integrated circuit does not seem to be more difficult than programming, we have found that testing and debugging these chips is a much more *cumbersome* task. We use the word *cumbersome* to indicate that we believe there is fundamentally very little difference between testing and debugging programs and testing and debugging integrated circuits. Yet as practiced today, checking the design and fabrication of integrated circuits is a much more primitive art.

As computer scientists designing custom systems on silicon, we want to decrease the time required to reduce an abstract specification to a working implementation. This is in contrast to the usual situation in industry where the overriding concern is reducing manufacturing costs. However, as microprocessors and other complicated systems begin to require million-dollar investments in design, reducing the time required to get the first chip working is becoming increasingly important to industry as well.

The problem, therefore, is how to make testing and debugging custom integrated circuits at least as easy as testing and debugging programs. This paper explores this problem in four major sections. Section 2 discusses the terminology we use and gives the reader who is unfamiliar with integrated circuit design some background. Section 3 compares and contrasts the efforts required to test and debug programs and integrated circuits. Section 4 describes the current state of the art and section 5 presents some ideas for simplifying chip testing and debugging. Some of the sections in this paper have appeared in part in [Frank 80a].

While this paper concentrates on testing and debugging integrated circuits, most of the techniques presented are directly applicable to testing printed-circuit boards and other hardware assemblies as well. In fact, some of the techniques we shall discuss originated as board-testing methods. We believe that testing and debugging integrated circuits is more interesting than testing and debugging non-integrated systems because it is extremely difficult to probe the insides of a chip directly.

2. Terminology

2.1. Testing and debugging

Testing is the process of detecting errors and *debugging* is the art of determining the exact nature and location of suspected errors and removing them (paraphrased from [Myers 79]). We shall term the object being tested or debugged the *unit under test*. In the case of integrated circuits there are actually two kinds of errors: first, the design of the integrated circuit may be incorrect and second, the fabrication of the chip that physically realizes the design may be flawed. Hence there are two forms of testing and debugging that must take place: first, testing and debugging the chip design and second, testing and debugging physical chips, typically in large numbers as part of a manufacturing process.

2.2. Integrated circuits

Integrated circuits, or *IC's*, or *chips*, are *fabricated* from a piece of silicon and geometric *masks* that indicate where the silicon should be altered in order to create components such as transistors, wires, and bonding pads. Integrated circuits are fabricated on a *wafer* that is three or four inches in diameter and may contain as many as a hundred chips or *dice*, each about one centimeter square.[1] After the wafer is fabricated, the dice are cut apart and packaged in carriers that may be easily mounted on printed-circuit boards. Connections between the carrier and the chip are made using bonding wires that attach bonding pads on the chip to *pins* on the carriers; typically, carriers have from 14 to 64 pins.

The fabrication process can be likened to printing, where the masks serve the role of printing plates and the wafer the role of paper. The job of the chip designer is to specify the precise geometry of the masks, which determine the details of component fabrication, placement, and connection on the wafer.

2.3. Integrated circuit designs

Although chip fabrication is controlled principally by mask geometry, the task of designing an integrated circuit usually employs several additional kinds of specifications. Throughout this paper we will use the phrases *integrated circuit design* or *chip design* to indicate all of the specifications which a designer might use when designing a custom integrated circuit. Van Cleemput [vanCleemput 79] has broken these specifications into three different hierarchies:

1. The *behavioral hierarchy* specifies the functional operation of a design, including performance requirements such as speed or power consumption. A *hardware*

---

[1] The reader interested in a more detailed description of the steps required to fabricate a chip is referred to [Mead 80] and [Hon 80a].

description language such as ISPS [Barbacci 77] might be used to record some of the behavioral specifications.

2. The *structural hierarchy* or logical hierarchy describes how the design is partitioned into different logical pieces and how these pieces are interconnected. A common device for illustrating the structural organization of a system is the *block diagram*.

3. The *physical hierarchy* describes how the design is physically implemented. For example, the surface of a microprocessor chip may be divided into separate regions for an ALU, for a memory, and for a control ROM. In turn, the memory is divided spatially into identical rows of "words." A word is divided into identical "cells," each of which stores a single bit. In large designs, the physical hierarchy extends upward from the chip to carriers, boards, cages, racks, and cabinets. Of particular interest to the designer is the *first silicon*--the first chip to be fabricated from a completed design.

These are three different hierarchical representations of the same design, and as a consequence interact strongly. For example, the speed of a circuit, determined by a behavioral specification, depends on the sizes of transistors and wires that implement it, determined by a physical specification.

From the point of view of testing and debugging, behavioral specifications are of greatest interest. One breakdown of the behavioral hierarchy might be:

- An abstract specification written in a language such as Alphard [Wulf 76] or something as simple as "I want a chip to interpret PASCAL."

- A high level specification in a register-transfer language such as ISPS.

- A low-level specification that describes the behavior of computations in terms of interconnections of individual transistors or gates. The mask geometry and the chip itself reflect precisely this lowest-level specification.

In addition to checking chip behavior, debugging and testing real systems often requires checking aspects of the physical specifications as well. In particular, the chip designer must verify that her design meets certain *design rules* for the particular fabrication process to be used.

2.4. Statefull and stateless chips

We have found it useful to classify designs based on the amount of state recorded internally in the design. A *stateless* chip is one that has no internal state. Its outputs can be described by logic equations written in terms of its inputs, and can be generated by combinatorial logic on the chip. A *statefull* chip is one that has a good deal of internal state that cannot be observed directly by sensing output pins. We term this state *revealed* if there is some way to report each bit of state to an output pin independently of all other state, or *concealed* if there is no way to detect the state externally except in a way that depends on other internal state.

3. A comparison of techniques for testing and debugging programs and integrated circuits

One often hears that software and hardware are two different ways of implementing the same thing. Indeed, it is possible to convert an abstract specification of a digital computation to a program for a general-purpose computer or to an IC design. The techniques and tools for carrying out these implementations are similar in many respects and different in others. One area where differences arise is in testing and debugging the two implementations.

Whether testing and debugging programs or chips there are usually three areas of concern: checking syntax, checking semantics, and checking performance.

3.1. Checking syntax

The syntax of a design is checked by some process that is responsible for converting it into an executable object. A compiler normally checks the syntax of a program in part to ensure that the compiler will generate code that correctly implements the programmer's specification as given in the program. Checking program syntax is usually straightforward since programming languages are designed with computer processing and syntax checking in mind.

Some of the constraints on hardware designs have a simple syntactic flavor. For example, design rules for TTL circuits forbid wiring two output pins together, and forbid loading an output with more than a specified current. These rules can be checked easily by computer-aided design (CAD) systems that are given a description of the design and the properties of the TTL parts used.

The integrated circuit fabrication process imposes additional syntactic rules on a design. These rules are intended to describe the class of designs that the fabrication process will implement correctly. The rules are more complex than programming-language syntax rules because the fabrication process is limited by physical, pattern-replication, and performance constraints that do not always admit a simple fabrication syntax. Instead, the design rules that must be followed to ensure proper fabrication are often complex and difficult to check mechanically. To make matters worse, the amount of information to be checked is usually many orders of magnitude more than that in a computer program: the pattern-matching cell of [Foster 80] (see Figures 3-1, 3-2, 3-3) can be specified with three simple statements in a logical notation, but requires approximately 1500 fabrication rule checks for a circuit containing only 13 transistors. Fabrication rules are generally geometric in nature, but also depend on electrical properties.[2] Until very recently design rule-checking was often done by eye: studying large plots of masks, even for very big chips. Computer

[2] A typical design rule is 'Metal wires must be *n* microns wide and must be separated by at least *k* microns, unless the wires carry the same signal, in which case they may be arbitrarily close together.'

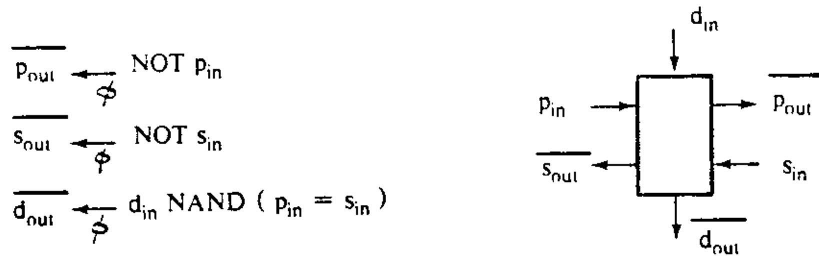

Figure 3-1: The logical connections and operation of the comparator cell of a pattern-matching pipeline. The arrows are labeled with the symbol φ to indicate that the outputs are clocked.

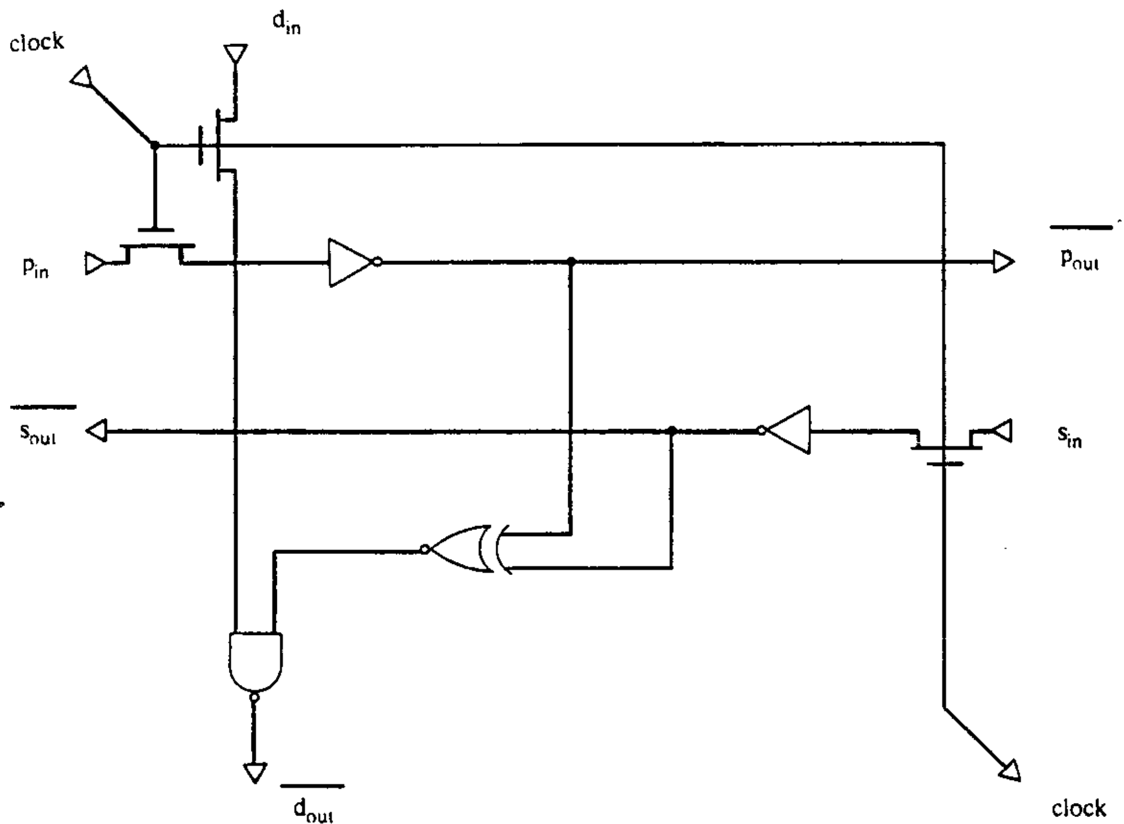

Figure 3-2: A circuit diagram of the comparator cell. Note the use of pass transistors gated by the clock signal to provide storage.

programs to check geometric rules are now in widespread use, although they often require hours or days of computer time to check large designs [Baker 80, Haken 80, McCaw 79]. Because the complexities of the fabrication process cannot be encoded in geometric rules alone, these programs often report "errors" that an engineer familiar with fabrication can determine to be benign.

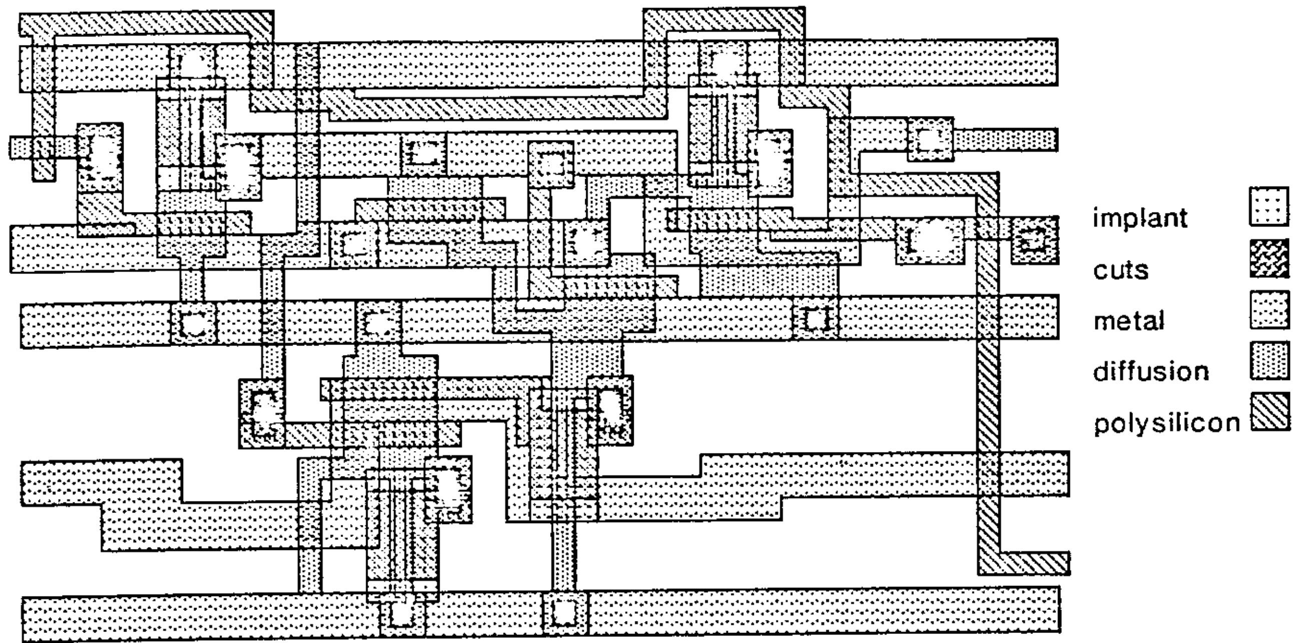

Figure 3-3: The layout of the comparator cell. Transistors are formed wherever diffusion and polysilicon cross. The overall placement of components is similar to that in the circuit diagram of Figure 3-2

3.2. Checking semantics

The only way to determine whether a design functions correctly is to demonstrate that a high-level specification of it is equivalent to a lower-level specification. This process may be repeated to show that the most abstract, high-level specification of a computation is equivalent to the lowest-level implementation description. Two broad classes of techniques are used to demonstrate equivalence: *testing* and *verification*. Verification checks formally that the input-output relation of an implementation always corresponds to that of another specification of the computation. A computation is tested by presenting test input data to an executable representation of it and then checking that the output data meets the input-output specification. While verification determines that the implementation works for all input data, testing must select a particular set of input data for trials.[3] In both cases, the correctness of the implementation is determined with respect to a more abstract or

[3] Verification is not very different from simulation: it amounts to a *symbolic* simulation of the computation, where the symbols stand for arbitrary, unknown data values. The verification simulates the implementation on unknown data. See [Darringer 79] for a discussion of the use of symbolic simulation for IC verification.

higher-level specification of its function. Ultimately, the correctness of the most abstract specification cannot be verified formally, but requires the designer or client to stipulate: "Yes, that specification correctly describes what I want the system to do." These techniques succeed at most in building our confidence that the implementation is correct, never in *proving* that it is right.

In practice, constructing correct computer programs depends heavily on testing. If one programmer writes a program to compute $\sin\theta$, another may write a test program that calls the *sin* program using various test values of θ to try to increase confidence that the *sin* program works correctly. The program tests well-known values such as $\theta = n\pi/4$, checks trigonometric identities such as $\sin\theta = \sin(\theta + 2\pi)$ and $\sin 2\theta = 2\sin\theta\sin(\theta + \pi/2)$ for various values of θ, and perhaps checks that $\sin\theta$ increases monotonically for $-\pi/2 \leq \theta \leq \pi/2$. If instead of a program we have a chip that computes $\sin\theta$, the same sort of test program can be devised. The test will increase our confidence that the software or hardware to compute *sin* is correct.

An important difference between testing software and hardware is that the software test determines not only that the *design* is correct, but that all present and future instances of the same program are correct. This advantage arises because we can copy software reliably from memory to disk to tape, using coding and redundancy techniques to ensure perfect copies. By contrast, testing a *sin* chip can determine only that the individual chip works correctly. If the chip works, we can conclude that the design is correct, but we cannot conclude that it can be manufactured repeatedly without error.[4] Hence the process of debugging chips is complicated. We must determine which errors we find in a chip are due to design mistakes and which are the result of bad fabrication. This is as hard as trying to debug a program when the compiler is randomly introducing errors into the compiled code!

Although a test program may determine that a unit under test does not function properly, the designer must still locate the design flaw. A programmer will often use a *debugger* to help locate the flaws in a program. The debugger allows her to access internal state of the program at various times during its execution, and perhaps to interrupt execution by inserting *breakpoints*. Sometimes debugging must be anticipated, so that the compiler can generate special code that helps the debugger access the program state. If no debugger is available, the programmer often inserts "print statements" to reveal state within the program.

Many of these same debugging techniques are used for hardware. If the probes of a *logic analyzer* are connected to various signals on a printed-circuit board, the analyzer will display a history of the digital states of these signals. The analogue of breakpoints is obtained by *triggering* the analyzer to

[4]This unfortunate situation arises because fabricating high density integrated circuits is a delicate process that may introduce errors. For example, impurities such as dust may contaminate the silicon during fabrication resulting in defects such as broken wires or non-functional transistors. It is not unusual for complex circuits to yield as few as 3% working parts out of a fabrication batch.

stop (or start) displaying when a particular combination of signals is detected. Because the analyzer has a small memory to record a history of 100 or so previous states of the probed signals, an engineer can observe what happened *before* the logic analyzer was triggered as well as after. Unfortunately, these techniques break down for debugging IC's because we cannot readily attach probes to signals within the chip. To use these techniques, we must arrange to reveal all the state within a chip (see sections 4.2.2 and 5). Like the compiler or programmer who includes special code for debugging, these techniques require incorporating into the design special hardware for debugging. Unfortunately, once the design is debugged, it is usually not practical to recompile it to remove the extra hardware inserted primarily for debugging. This may not a problem if the extra debugging hardware can also be used for production testing.

Both hardware and software are plagued by errors that occur only infrequently, or are hard to stimulate, or defy tracing back to an understandable cause. Both hardware and software debuggers can lay "bug traps" or "data-structure checkers" [Simonyi 76] for errors by assembling special hardware or software to detect a combination of events that is suspicious. This practice is straightforward with software, which can be easily modified and recompiled. Attaching bug traps to printed-circuit boards is less straightforward, but feasible. But for integrated circuits, installing a bug trap depends on designing and fabricating a new chip. Unless all imaginable bug traps are designed in at the outset, an unlikely possibility, laying traps within a single IC is not feasible.

The debugging process depends on correcting errors in the design and testing it again. When a programmer finds and corrects an error in a program it is usually only a couple of minutes until she has a new executable program which she can test again. While it may not take any longer to detect and correct an error in an integrated circuit design, it will take at least several weeks to get back a fabricated chip.[5] To compensate for the fabrication delays, the integrated-circuit designer relies heavily on simulation. A computer-readable specification of the chip can be changed in a few minutes, and the function of the chip tested again by simulation based on the new specification. If the simulation can mimic the behavior of the fabricated circuit exactly, all of the design errors could be found. Unfortunately, exact simulations of electrical behavior require enormous amounts of computation, and the designer usually compromises by simulating the design "at the logic level."

Simulation also helps to distinguish design errors from manufacturing flaws. If a chip is tested and found not to work even though a simulation is correct, the designer immediately suspects that the particular chip selected was flawed during manufacture and tries another chip.

---

[5] In the case of MPC79 turnaround time was one month. For MPC580 turnaround time was two months. Producing silicon wafers is "a complex procedure that involves over 40 individual steps (for silicon-gate NMOS) and roughly 50 hours of process time... The 50 hours of processing are typically spread over a month of calendar time because of the frequent inspections and economic realities of achieving high fab line throughput." [Hon 80b]

3.2.1. Designing to reduce errors

Designers of both hardware and software systems strive to reduce errors in a design by adopting design styles that help avoid mistakes. Synthesizing complex designs from already-tested modules is the most common technique. Current trends in programming-language design stress facilities for designing and coding modules separately and for carefully controlling the sharing of specification and implementation information among modules (e.g. MESA [Mitchell 79] or Ada [DOD 80].

Modularity is also an important part of digital system design. Engineers design printed-circuit boards that use integrated circuit modules from families that strive for interconnection compatibility (e.g., TTL, ECL). More recently, entire boards have become modules in systems organized around popular back-plane busses such as the DEC Q-Bus or the Intel Multibus. The modules have well-defined interfaces that are convenient to use. Moreover, each module may be tested individually, so that subsequent debugging and testing is confined to the program or circuit that employs these modules.

Modular structure within integrated circuits is also useful. The MPC projects designed by students and researchers in universities have used modules from a *cell library* that provides commonly-used circuits such as connection pads and output drivers. A popular chip design technique in industry is the *standard cell* approach, where the designer constructs integrated circuits by specifying the locations and interconnections of particular standard cells, whose designs are obtained from a cell library [Preas 77]. Although chips implemented using standard cells do not make efficient use of the silicon area available, they are much easier to debug because the modules, i.e., standard cells, have already been tested.

It is important to test all aspects of electronic modules that are critical to their correct functioning in the assembly. It is all too common to test integrated circuits in situations that are not as demanding as those in the final circuit. For example, a circuit output may be loaded more heavily in the assembly than in the test because it connects to more places. This may have enough effect on the performance of the module to cause the assembly to fail. This problem can also come up in software if the program used to test a module does not cover all of the module's specifications. It is for this reason that the module and the test program are often written by different people, both referring to the same specification.

3.3. Checking performance

It is not sufficient for a program or chip to function correctly--it must also perform adequately. Although experimental designs may have few performance requirements, often the most important aspect of a commercial product is its performance. For the programmer to measure the gross performance of a program she usually runs some test cases and measures the CPU time used.

Similarly, the IC designer can "turn up the clock speed" until a chip stops working.

Observations of gross performance are usually not sufficient to improve performance. The designer usually needs to locate bottlenecks in the system. For many programs we can recompile to request statement execution counts to determine where the program spends its time [Knuth 71]. Often, however, clues to performance bugs are best determined by measurements of particular properties of an algorithm, such as the occupancy of a hash table, the length of a list of free-storage blocks, or the length of a run-queue in an operating system. In these cases, the facilities to collect measurements must be incorporated into the design of the software. Much the same situation applies to hardware. Sometimes we can find performance bugs by observing activity on easily-accessible signals such as memory busses. More often, measurement facilities must be incorporated into the design. The execution speed of operations on a chip can be measured by incorporating special circuits (e.g. [Frank 81]) in the design or by simulating the chip in enough detail using model time accurately. Often timing information alone is insufficient. The designer of a cache chip, for example, will need to collect cache performance statistics such as the number of cache references, the number of main memory references, the number of reads, the number of writes, etc.. These usage-dependent statistics are required to determine whether the overall cache design performs as well as it should.

4. Current techniques for testing and debugging integrated circuits

This section explores the specialized techniques that have evolved for testing and debugging integrated circuits. The presentation is in two parts: techniques used during design, and those used in manufacturing. In both cases, we shall confine our attention to the problems of checking semantics, i.e., that the circuit operates correctly, and performance, i.e., that the circuit operates fast enough.

The discussion that follows emphasizes statefull chips and almost completely ignores stateless ones. Verifying the correct operation of a design or of a manufactured instance of a combinatorial circuit is straightforward, and is subsumed by techniques required by more complex statefull chips.

4.1. Testing and debugging IC designs

The fundamental problem with testing and debugging designs is that there is no way to exercise the design thoroughly before it is fabricated. As we mentioned in section 3.2, simulation tools are used to build confidence that the final design will perform properly. The most common simulations are performed with software that works from some form of machine-readable description of the design. Examples of common simulations used are:

- Register transfer level (RTL). A typical simulator of this kind is ISPS [Barbacci 77], which

allows a designer to describe a system at a high level. Storage structures such as registers and memory are declared explicitly. Control flow is also described explicitly, while dataflow is implicit. ISPS allows the description of parallel processes.

- Logic level. Circuits are described at the logic level in terms of AND and OR gates or modules such as TTL packages. Dataflow is represented explicitly by wiring connections, whereas control flow is implicit. Example: SALOGS [Case 78].

- Gate level. This is very similar to logic level simulation except that circuits are usually represented in terms of transistors or very primitive gates rather than logic functions. Example: MOSSIM [Bryant 80].

- Timing level. A timing simulator may use either logic or gate descriptions of a circuit. Timing simulation is similar to gate/logic simulation, but simulates the delays of circuits and wires in addition to their functional properties. Examples: MOTIS [Chawla 75], SCALD [McWilliams 80], and FETS [Frank 80b].

- Circuit level. Circuit simulation predicts the electronic behavior of a circuit, given detailed values for all components such as transistor parameters, resistance of wires, and capacitances. The most popular simulator of this kind is SPICE [Dowell 79]. Circuit simulation is extremely helpful for predicting performance and for analyzing complex analog circuits such as the sense amplifiers in dynamic MOS memories.

By way of example, Figures 4-1, 4-2, 4-3 show descriptions of the pattern-matching cell of [Foster 80] (see Figure 3-3) presented to different simulators. Also shown are outputs of the simulations. The input data presented to the simulations was: [Pin = 1, Sin = 0, Din = 1, Clock = 1] followed by [Pin = 1, Sin = 0, Din = 1, Clock = 0].

As the level of detail in the simulation increases toward the bottom of the above list, the complexity of the simulation calculations increases, thus requiring more computer time and memory. As a consequence, the more detailed simulation techniques cannot practically be applied to complex designs. While ISPS can simulate entire computers (e.g., PDP-11 or VAX-11), SPICE is limited to a few hundred transistors unless supercomputers are used to run it. Recently, techniques have been developed for multi-level simulators that simulate in detail only those nodes that are active at a given instant and moreover allow the designer to specify different levels of simulation detail for different nodes. For example, using only a circuit-level specification of a chip, the SPLICE [Newton 79] simulator is able to simulate an IC design at the gate, timing, and circuit levels.

Many designs are debugged using special-purpose simulations. Frequently, a TTL breadboard of a chip is built so that both the functional and performance aspects of the design can be studied extensively. Moreover, a breadboard simulation can be operated at the same speed as the eventual design and can be used for developing the hardware that will surround the ultimate part and the software that will operate it.

Special-purpose software simulations are also very valuable. Programs may be written in any

```
Comparator :=
        BEGIN
        ** input.pins  **{US}
        CLOCK<0>,                                   !A global clock signal
        PIN<0>,                                     !input pattern bit
        SIN<0>,                                     !input string bit
        DIN<0>                                      !compare result from above
        ** positive.comparator.cell.nn  **{US}
        PosComp.nn\Positive.Comparator.nn
                (PIN.H.nn<0>{REF},SIN.H.nn<0>{REF},DIN.H.nn<0>{REF}):=
                BEGIN
                ** local.registers  **{US}
                POUT.L.nn<0>,                       !Our output P register
                SOUT.L.nn<0>,                       !Our output S register
                DOUT.L.nn<0>                        !Our output D register
                ** parallel.actions  **{US}
                PosComp.action.nn {MAIN} :=
                        BEGIN
                        IF CLOCK EQL 1 => POUT.L.nn _ NOT PIN.H.nn;
                        IF CLOCK EQL 1 => SOUT.L.nn _ NOT SIN.H.nn;
                        IF CLOCK EQL 1 =>
                                DOUT.L.nn _ NOT (DIN.H.nn AND (PIN.H.nn EQL SIN.H.nn))
                        END
                END
        ** the.main.action  **{US}
        Main.Action {MAIN} :=
                BEGIN
                REPEAT
                        BEGIN
                        PosComp.nn(PIN,SIN,DIN)
                        END
                END
        END
```

```
Report:
@ Head of POSCOMP.ACTION.NN
CLOCK=#1
SIN.H.NN=#0
PIN.H.NN=#1
DIN.H.NN=#1
SOUT.L.NN=#0
POUT.L.NN=#0
DOUT.L.NN=#0

Report:
@ Tail of POSCOMP.ACTION.NN
CLOCK=#1
SIN.H.NN=#0
PIN.H.NN=#1
DIN.H.NN=#1
SOUT.L.NN=#1
POUT.L.NN=#0
DOUT.L.NN=#1

Report:
@ Head of POSCOMP.ACTION.NN
CLOCK=#0
SIN.H.NN=#0
PIN.H.NN=#1
DIN.H.NN=#1
SOUT.L.NN=#1
POUT.L.NN=#0
DOUT.L.NN=#1

Report:
@ Tail of POSCOMP.ACTION.NN
CLOCK=#0
SIN.H.NN=#0
PIN.H.NN=#1
DIN.H.NN=#1
SOUT.L.NN=#1
POUT.L.NN=#0
DOUT.L.NN=#1
```

Figure 4-1: Register-transfer level simulation (ISPS). (a) the input specification for the comparator cell, c.f. the 3-line logical notation of Figure 3-1. (b) A sequence of four output reports showing the states of the signals.

```
(D(I D1)(T Ne)(L 27 60)(P 1 G)(P 5 S)(P 12 D))
(D(I D2)(T ResUp)(L -3 57)(P 8 D))
(D(I D3)(T ResUp)(L 135 57)(P 10 D))
(D(I D4)(T Ne)(L -42 48)(P 1 G)(P 13 S)(P 17 D))
(D(I D5)(T Ne)(L 51 30)(P 10 G)(P 8 S)(P 40 D))
(D(I D6)(T pdNe)(L -9 27)(P 13 G)(P 8 D))
(D(I D7)(T pdNe)(L 129 27)(P 36 G)(P 10 D))
(D(I D8)(T Ne)(L 204 24)(P 1 G)(P 36 S)(P 45 D))
(D(I D9)(T Ne)(L 96 9)(P 8 G)(P 10 S)(P 40 D))
(D(I D10)(T pdNe)(L 39 -18)(P 40 G)(P 58 D))
(D(I D11)(T ResUp)(L 93 -27)(P 40 D))
(D(I D12)(T Ne)(L 39 -30)(P 12 G)(P 58 S)(P 64 D))
(D(I D13)(T ResUp)(L 51 -48)(P 64 D))

(N(I N1 CLOCK_H)(C 1))
(N(I N5 DIN_H.nn)(C 5))
(N(I N8 POUT_L.nn)(C 8))
(N(I N10 SOUT_L.nn)(C 10))
(N(I N12)(C 12))
(N(I N13)(C 13))
(N(I N17 PIN_H.nn)(C 17))
(N(I N36)(C 36))
(N(I N40)(C 40))
(N(I N45 SIN_H.nn)(C 45))
(N(I N55)(C 55))
(N(I N58)(C 58))
(N(I N64 DOUT_L.nn)(C 64))
```

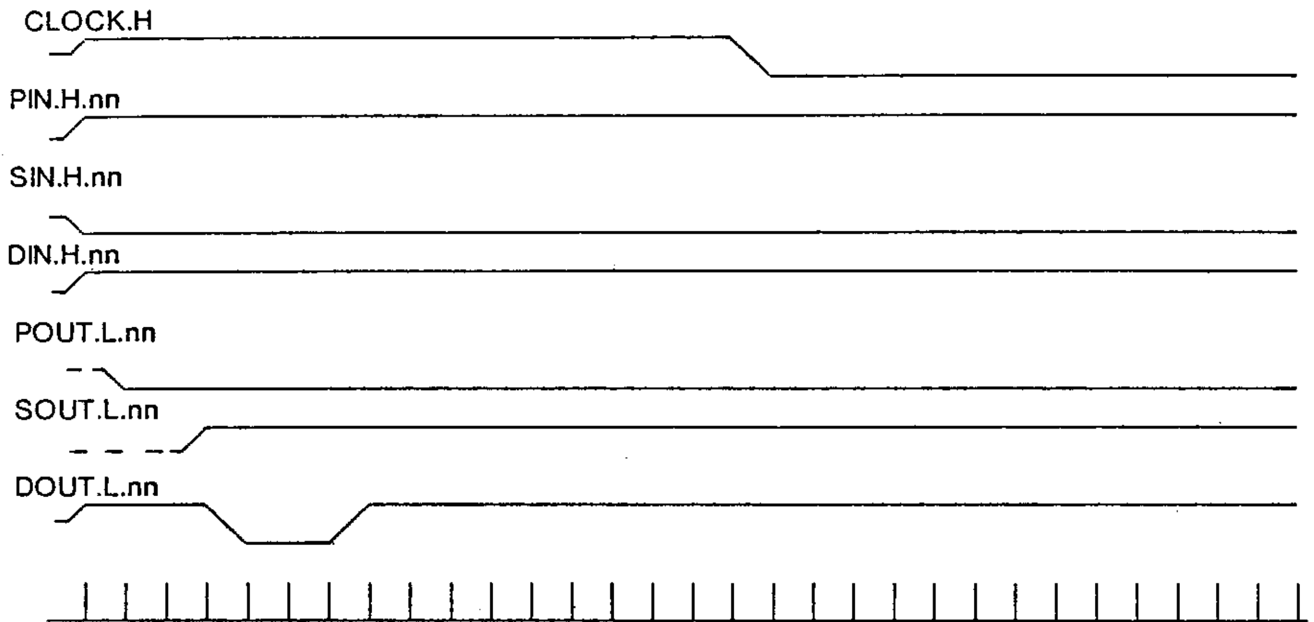

Figure 4-2: Gate-level timing simulation (FETS). (a) The input specification listing all transistors and their interconnections. (b) A display of the behavior of internal signals. This example show one clock period.

```
.option nonode nopage noacct nomod nolist
.width in=80 out=80
vds 1 0 dc 5
vpin 2 0 pwl(0ns 5)
vsin 5 0 pwl(0ns 0)
vdin 9 0 pwl(0ns 5)
vclk 13 0 pwl(0ns 0 10ns 0 11ns 5 30ns 5 31ns 0)
*
m1 2 13 3 0 menn l=6u w=6u
m2 1 4 4 0 mden l=18u w=6u
m3 4 3 0 0 menn l=6u w=18u
m4 1 8 8 0 mden l=24u w=6u
m5 8 7 4 0 menn l=6u w=24u
m6 8 4 7 0 menn l=6u w=24u
m7 1 7 7 0 mden l=18u w=6u
m8 7 6 0 0 menn l=6u w=18u
m9 5 13 6 0 menn l=6u w=6u
m10 11 8 0 0 menn l=6u w=18u
m11 12 10 11 0 menn l=6u w=18u
m12 9 13 10 0 menn l=6u w=6u
m13 1 12 12 0 mden l=24u w=6u
*
*
***** NMOS ENHANCEMENT-NOMINAL *****
.model menn nmos nsub=1e15 nss=-2.35e11
+xj=1u ld=.8u ngate=1e23 gamma=.43 nfs=1e11 lambda=1e-7
+uo=800 ucrit=6e4 uexp=.25 utra=.5 cbd=21e-5 cbs=21e-5 js=2e-5
*
*
***** NMOS DEPLETION-NOMINAL *****
.model mden nmos nsub=1e15 nss=7.05e11
+xj=1u ld=.8u ngate=1e23 lambda=1e-7
+uo=800 ucrit=6e4 uexp=.25 utra=.5 cbd=21e-5 cbs=21e-5 js=2e-5
*
.plot tran v(13) v(12)
.trans 1ns 40ns
.end
```

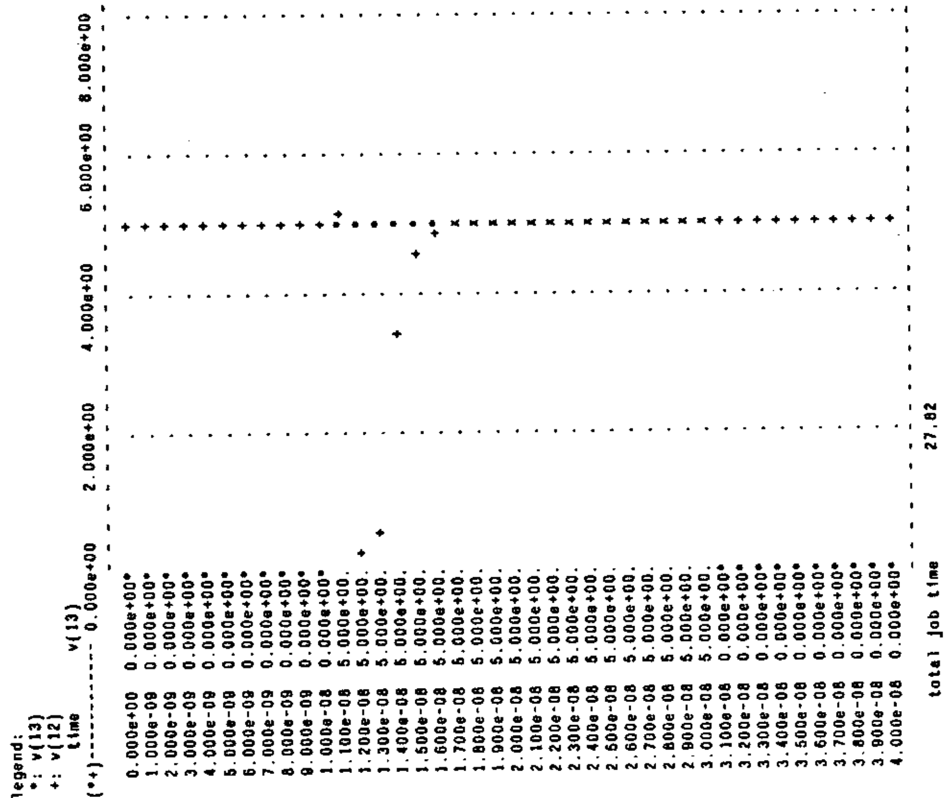

Figure 4-3: Circuit-level simulation (SPICE). (a) A description of circuit components and their interconnections. This description contains almost identical information to that of Figure 4-2a. (b) Output from SPICE. The + indicates the waveform of the DOUT signal. The * indicates the waveform of the clock signal. The x is used when * and + overlap.

convenient programming language to explore various aspects of a design. These simulations are generally used to explore widely varied design alternatives rather than to model detailed behavior of the design. The example shown in Figure 4-1 need not be programmed in ISPS--any convenient language will do. Furthermore, as discussed in Section 5.1.1, microcode can be simulated independently of the rest of chip.

4.2. Testing and debugging production IC's

The goal of production testing is to determine quickly whether or not a chip has been fabricated correctly. Many of the techniques developed to test parts are also used to help debug a design once the *first silicon* instance of the design is fabricated.

The integrated-circuit industry distinguishes between two kinds of testing: *characterization* and *acceptance testing*. Characterization carefully measures the behavior of a chip: power dissipation; input and output currents; variations with temperature, humidity, and supply voltage; vibration and radiation resistance; etc. These measurements are used to determine whether the design and the manufacturing process are tuned suitably to produce parts with the desired specifications. They are also used to determine why flawed parts fail in order to improve the design or alter the production process in an attempt to produce a greater proportion of working parts. Once a production line is operating smoothly, characterization may be used only for occasional samples.

An acceptance test is applied to each part that is manufactured to determine whether it operates correctly and should be sold. Tests are often performed by probing a wafer before it is diced and packaged, so that non-functional die are not packaged. Once packaged, the chip is tested again. The test is either a "go/no-go" test for acceptance or a classification test that sorts parts based on their speed.

4.2.1. Testing inputs and outputs

Testing methods developed for simple chips aim to exercise all circuits and wires within the chip to gain confidence that the chip works. The functional correctness of a stateless chip can be tested by stimulating it with all possible input values and observing the outputs. The test is driven by a set of *test vectors*, each of which records the digital values of the inputs and the digital values of the expected outputs. If a stateless chip has n input pins, 2^n test vectors must be tried if the test is to be exhaustive.

Various methods are used to implement test-vector testing (based on [Hayes 80]):

- *Stored response testing* which applies prestored test vectors to the inputs of a chip and compares the outputs of the chip to prestored results. The Megatest Q2 [Megatest 80] is typical of this class of testing machines.

UNIVERSITY LIBRARIES
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA 15213

- *Comparison testing* applies test vectors to the inputs of two chips: the chip under test and a known working chip called the *gold unit*. The outputs of the two chips are continuously compared.

- *Algorithmic testing* in which test data is computed each time the unit under test is tested is perhaps the most powerful technique because of its flexibility and the compact representation of the test. Algorithmic testers allow us to write programs to generate the input-output pairs dynamically instead of having to enumerate the test data beforehand. However, in order to test chips at high speed, a good deal of computation power is required.

Exhaustive testing becomes difficult or impossible when chips have substantial internal state. An exhaustive test will require that every combination of input signals and internal states be tried, or 2^{n+s} combinations, where s is the number of bits saved inside the chip. To make matters worse, several steps may be required to obtain a particular internal state. A typical integrated-circuit memory chip ($n = 10$ and $s = 16000$) simply cannot be tested exhaustively: it would require 10^{4800} steps, or 10^{4784} years at 100 ns per test!

Rarely is it really necessary to test chips exhaustively, because every output does not depend on every input and every bit of state. Rather, we assume a failure mechanism within the chip that is simpler than the sort tested by exhaustion. One approach to reducing test time is to hypothesize failure mechanisms based on the physical structure of the chip. A common restriction is to test whether each internal signal on the chip is "stuck at 0" or "stuck at 1" due to manufacturing flaws. Algorithms such as *Roth's D algorithm* [Roth 67, Breuer 76] when given a combinational logic function can automatically generate all of the tests required to verify that no internal signal is stuck. For each internal signal, the algorithm postulates that the signal is stuck and tries to find an input vector that causes an output to be inverted because of the failure. For chips with very regular designs, deriving test vectors for stuck-at conditions is straightforward, even if it cannot be done automatically. For example, most of the stuck-at tests for a memory chip are those that simply test whether each individual bit in the memory can be set to 0 or 1. For complex chips without a regular structure, deriving test vectors can be very difficult. The integrated-circuit industry today believes that one of its major problems in chip design is the design of a suitable set of acceptance test vectors. The reader is refered to [Muehldorf 81] for a more detailed discussion of fault modeling and test pattern generation.

Testing a complex chip may require many thousand test vectors, which are costly to store and compare at high speed. A compression technique called *signature analysis* can be used to hash the sequence of test outputs, which may be several thousand bits, into a small "signature" that may contain only 16 bits [Frohwerk 77]. The inputs to stimulate a chip can come from an inexpensive ROM memory. After several thousand tests are applied, we examine the signature to see if it matches a pre-

stored correct value. This technique has been used quite successfully with printed circuit boards, especially those that are part of a computer. The computer executes a test program while the signature of a particular wire on the PC board is measured with a test instrument. The signature is then compared to the value that the test should yield.

4.2.2. Accessing internal state

Debugging and testing complex chips that contain a lot of internal state depends on accessing the internal state easily. The time required to test a complex chip can be drastically reduced if the internal state of the chip can be set and revealed easily during testing. Access to internal state (AIS) allows us to decompose a complex chip into a number of very simple subparts for testing purposes, thus avoiding the combinatorial explosion of *all* combinations of input and internal state values. Moreover, access to internal state can reveal to the external tester the state of an internal signal that might otherwise be inaccessible or accessible only by very indirect means. Internal signals that are not easily accessible may require a very complex set of test steps to determine whether the signal is stuck or operable.

Some chips are designed so that access to internal state requires no special circuitry. For example, a memory chip provides reasonably direct mechanisms for reading and writing every bit in the memory. Although these mechanisms depend on correct operation of a certain amount of addressing circuitry, these circuits represent only a tiny fraction of the entire chip. Consider by contrast a microprocessor's instruction register that contains a copy of the instruction being executed. The register is not externally accessible, even indirectly. Although we can observe externally some effects of an instruction's execution, we will almost certainly not be able to decide based on these observations whether the instruction register operates correctly under all circumstances.

Scan-in/scan-out. The most popular technique for accessing internal state is to link all such state in a long shift register. The state can be shifted into the chip, a test executed, and the new state shifted out; hence the name scan-in/scan-out (SISO). The idea has old origins: early IBM 360 models wrote several thousand bits of internal processor state on a magnetic tape whenever a hardware error was detected. The tape could then be analyzed to find the error. More recently, DEC and Evans and Sutherland have used shift registers to reveal state that would otherwise be hidden on printed-circuit boards in a large, complex system such as a VAX-11/780 processor or a real-time hidden surface eliminator. In addition to providing diagnostic help, the shift register can also be used to configure the system by setting internal state that governs its performance, for example, control registers that enable and disable memory modules.

The use of scan-in/scan-out in integrated circuits has been exploited by IBM, where it goes by the names Level Sensitive Scan Design (LSSD) or Shift Register Latch (SRL) (see Figure 4-4) [Williams

73, Eichelberger 78]. This technique was developed for chips designed with the Electronic Design System (EDS) in order to systematize and reduce the testing requirements for custom chips designed by hundreds of engineers within IBM. As a result, a single testing philosophy and one kind of testing hardware applies to vastly different chips. Moreover, when the chips are combined into larger systems, the shift registers can all be chained together to make the state of entire boards available for fault diagnosis.

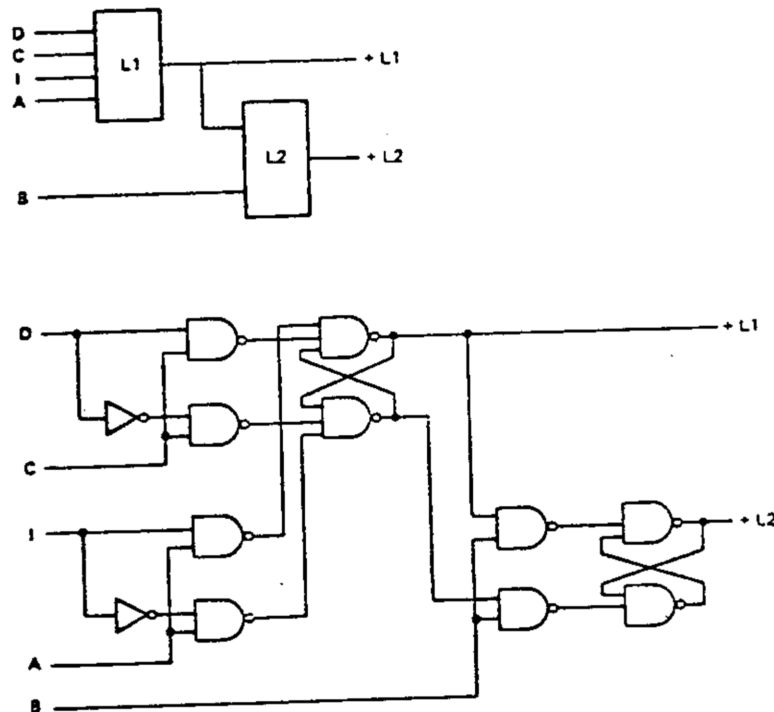

Figure 4-4: The shift-register latch (SRL) used by IBM to provide access to internal state. In normal use, input data is presented at D, clock (actually a latching hold signal) at C, and output is available at +L1. These latches are connected into a shift register by connecting +L2 (SHIFT DATA OUT) to I (SHIFT DATA IN) of the next latch. The hold signals A and B are used alternately to shift data from L1 to L2 and to the next L1, and so forth. Signals A and B can be generated on-chip from a single SHIFT CLOCK signal. *Figure from [Eichelberger 78].*

One of the charms of SISO is that very few pins are required to access the internal state. The IBM design arranges that each bit of internal state is saved in two latches, one normal one and one that is used to create a shift register. Three signals are required to set and retrieve the internal state: SHIFT DATA IN, SHIFT DATA OUT, and SHIFT CLOCK. Even fewer pins are required in the TRIMOSBUS design, which has two pins for shift register input and output, but encodes the shift clocks on a bus that is shared by all chips and normally devoted to other uses [Sutherland 79].

Mufflers. Another method for accessing internal state was used in the design of the Dorado processor [Lampson 80]. The packaging constraints of the processor resemble those of an integrated circuit: probes cannot be attached to the printed-circuit boards because they are very

closely spaced and because they cannot be operated when extended from the backplane due to signal timing problems. To observe internal signals, the designers arranged to use a set of selectors, termed *mufflers*, to select a single internal signal that is to be reported externally. The control of the selectors and the observation of the external signal are accomplished with a diagnostic computer attached via an umbilical cord for testing or debugging.

The muffler scheme is designed to use only three pins on each board: an ADDRESS pin, an ADDRESS-SHIFT pin and a DATA pin (see Figure 4-5).

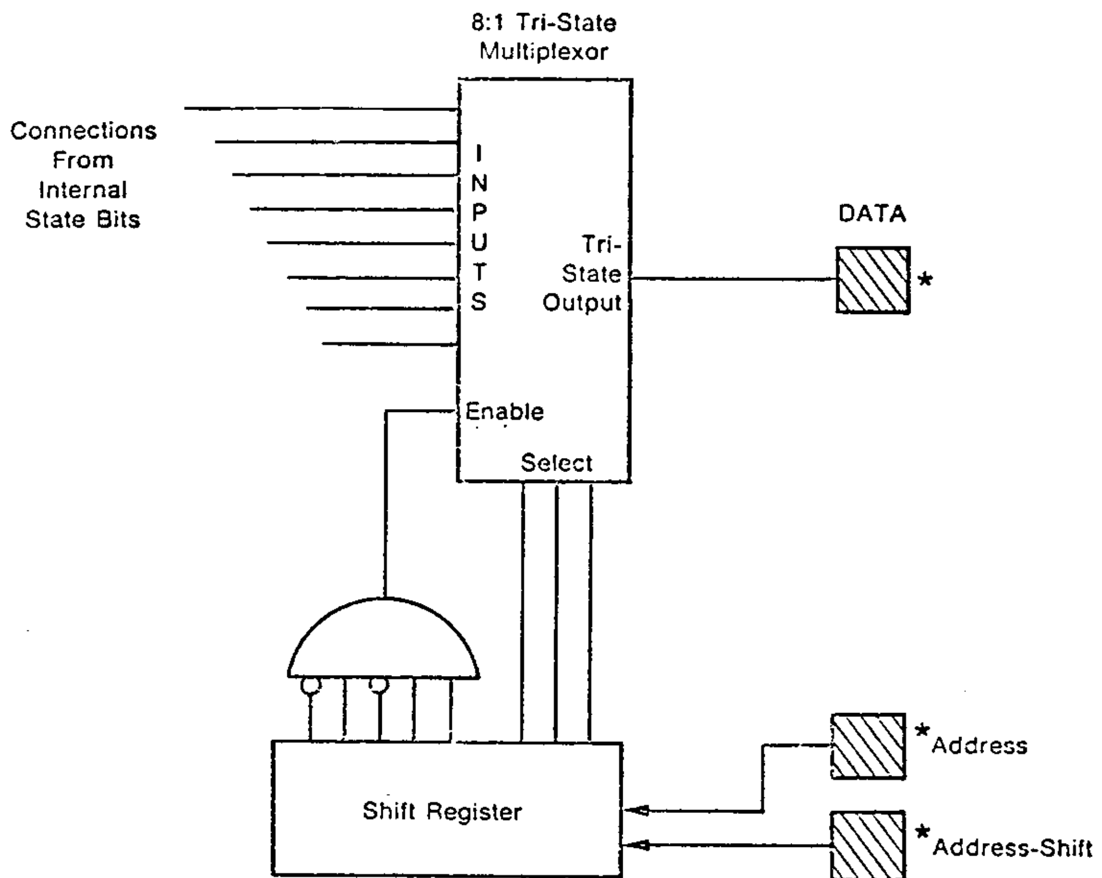

Figure 4-5: A TTL schematic representation of a muffler. The shift register is loaded with the address of a signal to sense. The AND gate will enable the one multiplexor that connects to the addressed signal, thus gating the signal onto the DATA wire. All three pins are bussed together, so that an arbitrary number of mufflers can be connected using only three wires.

These three signals are bussed together on the backplane. Each bit of internal state is given a unique n-bit address; addresses are presented in bit-serial form simultaneously to all boards in parallel using the ADDRESS and ADDRESS-SHIFT signals. Each board decodes the addresses and enables a selector that gates the desired signal onto the single-bit DATA bus. A single bit of internal state can be accessed in n address-shift cycles, which enter its address. By clever address permutations, however,

all 2^n addressable state bits can be read out in 2^n address-shift cycles. For complete state readout, therefore, mufflers and SISO require the same number of clock cycles, yet the muffler can access a single bit of state far faster than SISO can.

Both the SISO and muffler techniques require a small amount of extra hardware to access internal state. IBM estimates that roughly 20% of a chip is devoted to SISO mechanisms, although very often the latches required for the shift register are also used during normal operation. The muffler overhead, as used in the Dorado, is much lower than this, but the mufflers do not report as much internal state as IBM requires. SISO can be used both for reading and setting state, while the mufflers in the Dorado are used only to read state. Both schemes are able to reduce testing times and ease debugging significantly using only a few precious pins.

5. Future trends in IC debugging and testing

In this section, we explore the trends likely to influence testing and debugging in the near future. We tend to place greater emphasis on debugging, since we are more concerned with reducing design time than with the production of large numbers of chips. We also present a few new methods for testing and debugging integrated circuits and describe briefly the work underway at CMU.

5.1. Designing to reduce errors

The effort required to debug a design can be reduced by using design techniques that help avoid errors or that make errors particularly easy to detect, track down, and fix. An error may be difficult to fix if the remedy requires redesigning a great many associated circuits or if the fix requires a new circuit layout that causes changes to propagate to other parts of the chip, thus necessitating still more layout work.

5.1.1. Microcode

The most dramatic technique of this sort is the use of microcode control of some "execution machine." The microcode can be written in a reasonably high-level language (at or above the register-transfer level) [Hennesy 81, Gehringer 80, Gosling 80, Steele 80] and simulated extensively. The simulations can be made to execute very rapidly, for example by compiling executable code from the microcode source. The simulations may suggest alterations in the structure of the execution machine, which need not have been designed in detail at the time the microcode is written and tested. Once the execution machine is designed, it can be simulated separately using lower-level simulations to build confidence that microcode will be executed properly.

When the design of the execution machine is complete, the microcode can be compiled directly into ROM or PLA layouts that provide microinstruction memory on the chip. Techniques for reducing

the size of the microinstruction memory, such as *nanocode* [Stritter 78], can also be handled by this compilation step. The designer need never deal with the complexities of these structures--the original microcode source is the lowest-level representation of the control structure that she needs to manipulate.

The microinstruction memory can be easily fitted with AIS testing connections. All that is needed is a way to set and read the microinstruction memory address register (MAR) and to set and read the microinstruction register (MIR, the output of the microinstruction memory). These paths allow us to test the contents of the microinstruction memory exhaustively in a very short time. Moreover, they provide a way to test the execution machine: by loading values into MIR using AIS, the machine can be stepped through any series of operations, regardless of the contents of the microinstruction memory. During debugging, if an error is found in a microcode word, the correct microcode values can be substituted by the AIS mechanism each time the word is fetched, thus allowing the chip to be tested in spite of the error. A microprocessor designed with microcoded control may devote 50% or more of the chip area to microinstruction memory. The MAR and MIR registers and their test connections constitute a clean division of the chip into two parts, which may be tested or debugged separately.

The impact of microcoded control on the design and debugging of chips can be illustrated by comparing the history of two 16-bit microprocessors that have complex control requirements: the MC68000 [Motorola 79, Stritter 79] and the Z8000 [Zilog 79, Peuto 79]. The MC68000 was designed using microcoded control, while the Z8000 uses combinatorial logic for its control. Although the MC68000 was fabricated using a somewhat denser and faster technology than the Z8000, this difference is not important. The overall architectural complexity of the two microprocessor chips is comparable. Figures 5-1 and 5-2 show the actual chip layouts, illustrating the obvious regular microcode memories in the MC68000. As seen in Table 5-1 the most striking testimony to the advantages of microcoded design is the fact that the Z8000 required three times as many months to produce a working production version (i.e. a chip with no known bugs) as the MC68000. The various forms of simulations of the MC68000 and the test logic on the chip itself made it possible to examine essentially all internal signals in order to locate the source of errors. Moreover, even bugs which did not originate in the microcode of the MC68000 could often be fixed using microcode instead of eliminating the true source of the error by redesign. On the other hand, locating the sources of bugs in the Z8000 was often tedious and required difficult changes to the random-logic control.

Quite apart from testing and debugging efficiencies, use of microcode offers substantial savings in layout effort as well. Even though the MC68000 has four times as many transistors as the Z8000, both chips required approximately the same amount of effort to design and lay out.

CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PENNSYLVANIA

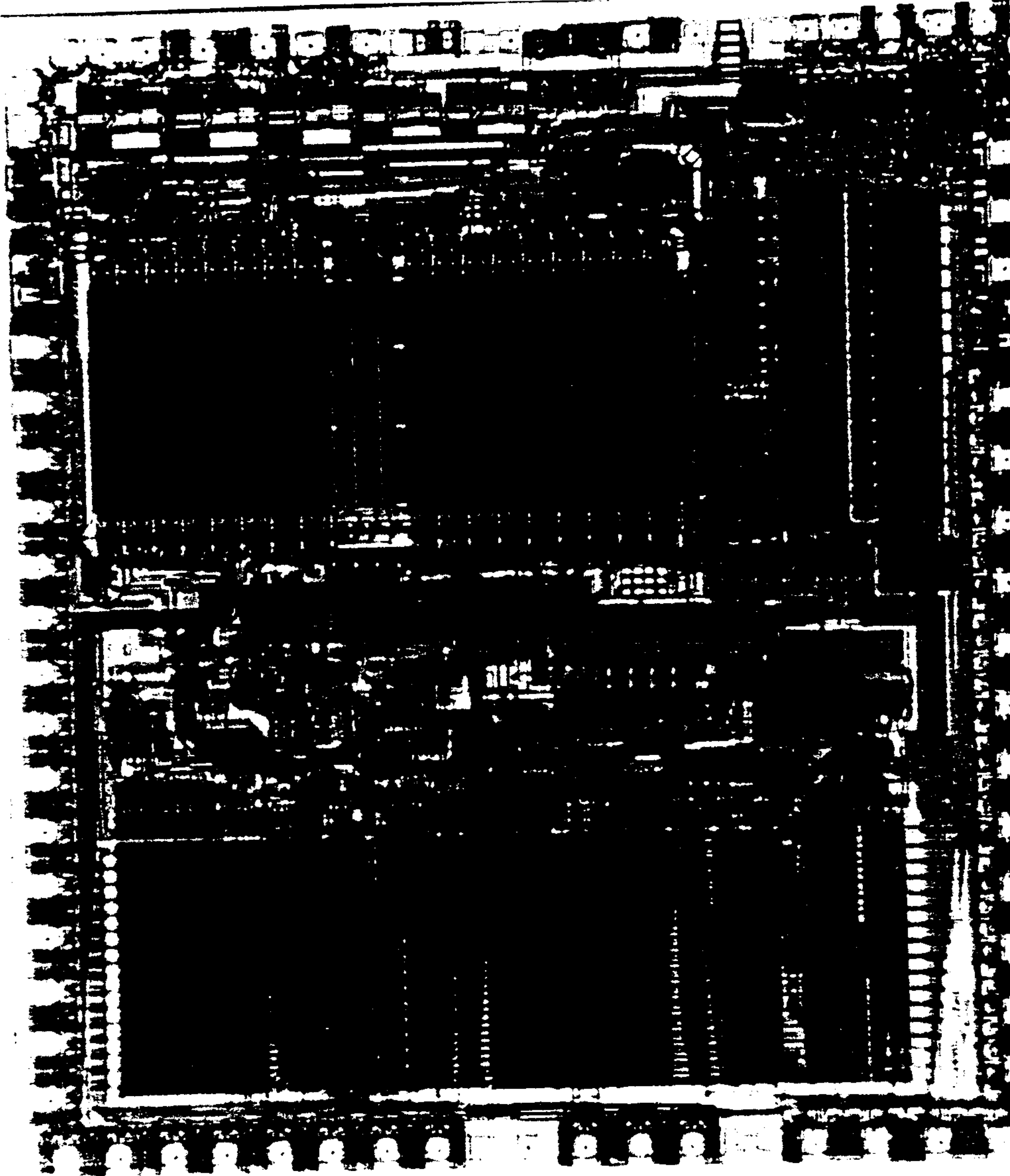

Figure 5-1: Photomicrograph of the MC68000. *Courtesy Motorola Inc.*

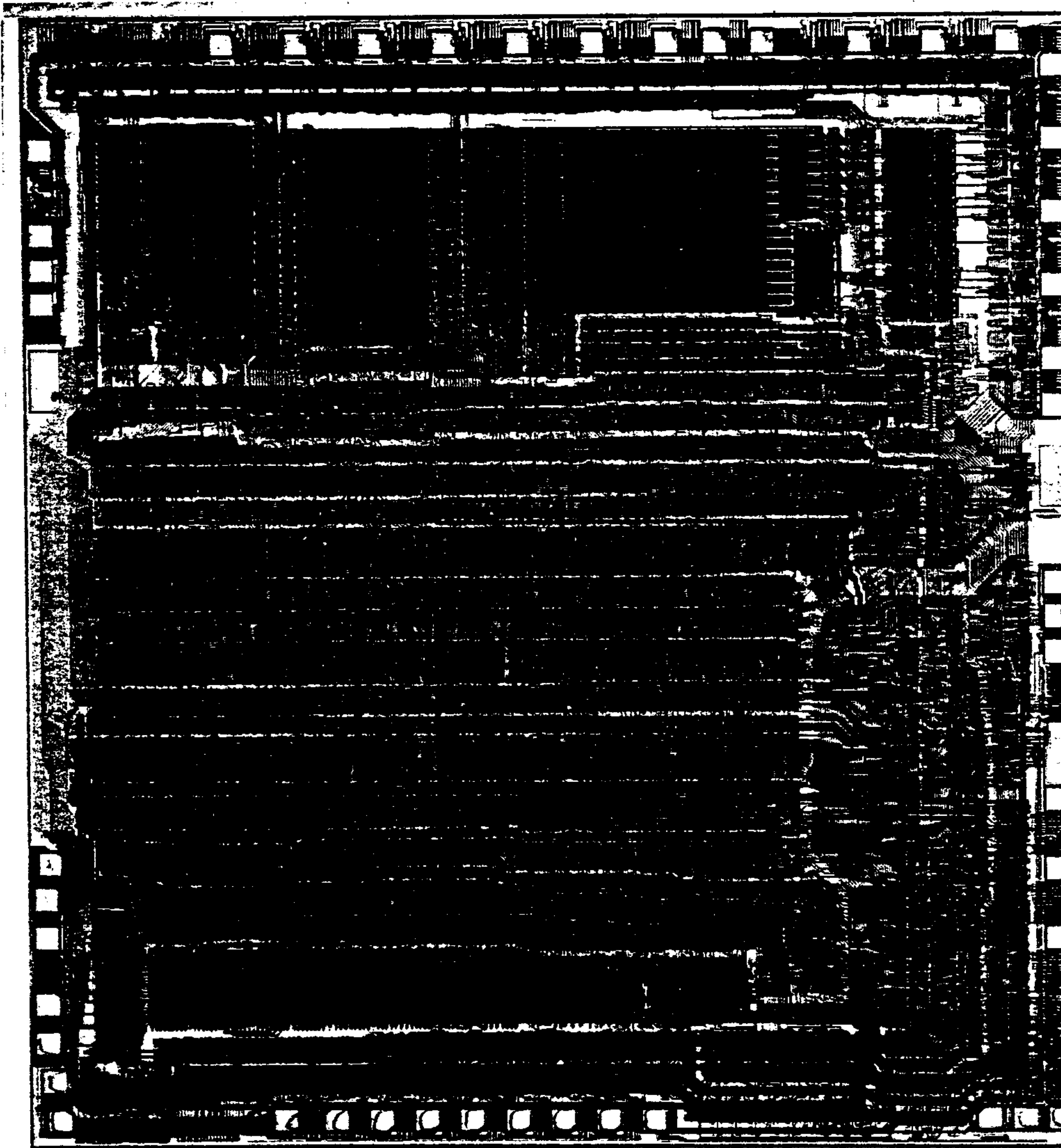

Figure 5-2: Photomicrograph of the Z8000. *Courtesy Zilog, Inc.*

	Motorola MC68000	Zilog Z8000
Design parameters:		
Elapsed time to first silicon	30 months	30 months
Design time	100 man months	60 man-months
Layout time	70 man months	70 man-months
Lambda	1.75μm	2μm
Chip size	246 mils x 281 mils	238 mils x 256 mils
Number of transistors	68000	17500
Number of transistors/ Number of designed transistors	74:1[6]	5:1 (estimate)
Debugging parameters:		
Elapsed debugging time[7]	6 months	18 months
Number of mask sets	3	3
Percent of chip design simulated at any level	100%[8]	20%
Percent of chip area devoted to test structures	4%	0%

Table 5-1: A comparison of the MC68000, and Z8000 design efforts. *Data supplied by Tom Gunter of Motorola, Inc., Bernard Peuto of Zilog, Inc., and Skip Stritter of Nestar Systems, Inc.*

5.1.2. The CMU Design Automation System

Microcode is one way in which we can automatically generate an integrated circuit from a high-level specification. Another approach has been used in the CMU Design Automation System (CMUDA) [Parker 78]. Instead of writing microcode to implement a particular application, the designer describes her task using the ISPS hardware description language. Given this ISPS specification and information about the particular high-level implementation style desired, the CMUDA system can generate a low-level implementation of a chip using *standard cells*. In addition, a system such as CMUDA could also automatically generate AIS for all of the internal state it allocates on the chip.

5.2. Design for testability and debuggability

If a system cannot be designed using a style that reduces the number of errors or makes testing inherently simple, it can nevertheless incorporate features that aid debugging and testing. Industrial response to the increasing difficulty of testing complex chips has been to emphasize *design for testability*. In our environment, it is more important to design chips that can be debugged easily because we cannot afford the time or expense of iterating a chip design to eliminate design errors.

We observed in section 3.2 that techniques for debugging software modules require two primitives: access to internal state in the module and the ability to stop execution (breakpoints). These two functions can be offered to the hardware debugger as well by designing the chip with sufficient

---

[6] This is actually the weighted average of three different ratios. 50000 transistors sites are in PLA/ROM structures with a ratio of 50:1. 5000 transistors sites are in the registers with a ratio of 500:1, and 13000 transistors sites are in random logic with a ratio of 5:1.

[7] Time from first silicon to a working production version (i.e. no known bugs) of the chip.

[8] In addition, there also exists a TTL implementation of the MC68000 which is 100% functionally equivalent to the chip.

access to internal state information and by debugging a chip with a driver that can single-step a sequence of operations.[9] Thus we see that access to internal state is useful for debugging as well as for testing.

The remainder of this section explores ways to design access to internal state to ease both debugging and testing. The final paragraphs (section 5.2.5) contain a summary of the kinds of internal state that should be made available externally.

5.2.1. In-situ testing

A chip can be designed so that AIS access can be used to test a chip even after it is soldered into a printed-circuit board. The boards and systems built around these chips can then be tested as well. The key requirement is that the AIS access be able to decouple all of the pins from their normal connections (except for power and ground) and provide an externally-generated stimulus in place of the signals present on the pins. These functions can be provided by suitable design of the pin interface circuits, and are independent of the design of the rest of the chip.

Figure 5-3 illustrates schematically how the pin circuit module might be designed. Each signalling pin is connected to the chip circuits with three wires: one that reports data coming in from the pin (IN), one that delivers data to be placed on the pin (OUT), and a third (DRIVE) that says whether the pin is to be driven. If the pin is used only as an output, it will always be driven; if it is used only as an input, it will never be driven; if it carries a bidirectional signal such as a bussed data wire, then the DRIVE signal will vary with time. Figure 5-4 shows how the testing and debugging circuits might work. The OPERATE signal couples the chip signals directly to the pin driver for normal operation. If OPERATE is not asserted, the pin and the chip internals are electrically separated. By asserting DEBUG CHIP, we couple the three chip signals to AIS connections that can sense the state of the DRIVE and OUT signals and control the state of the IN signal.

This design can also be used to test the interconnections among chips in a large system. By asserting the DEBUG PINS signal, we couple the AIS connections to the pin driver signals. In this way, we can drive a single pin in the system and measure the state of all other pins to verify that each pin is connected properly to others. This test will uncover bad chip bonding, bad chip sockets, bad printed-circuit boards, bad printed-circuit board sockets, and bad backplane wiring.

This scheme greatly simplifies the design of a hardware tester, because we can decouple the testing of the pin electronics from the testing of the bulk of the chip. The chip function is tested using

---

[9] This comment applies to sequential circuits. Asynchronous or self-timed designs have no global concept of sequence, so it is not possible to "stop" an entire circuit and to measure its state. However, such designs generally are composed of a number of smaller circuits, each of which is sequential [Seitz 80]. In this case, our comment applies to debugging the sequential components but not to debugging the asynchronous whole.

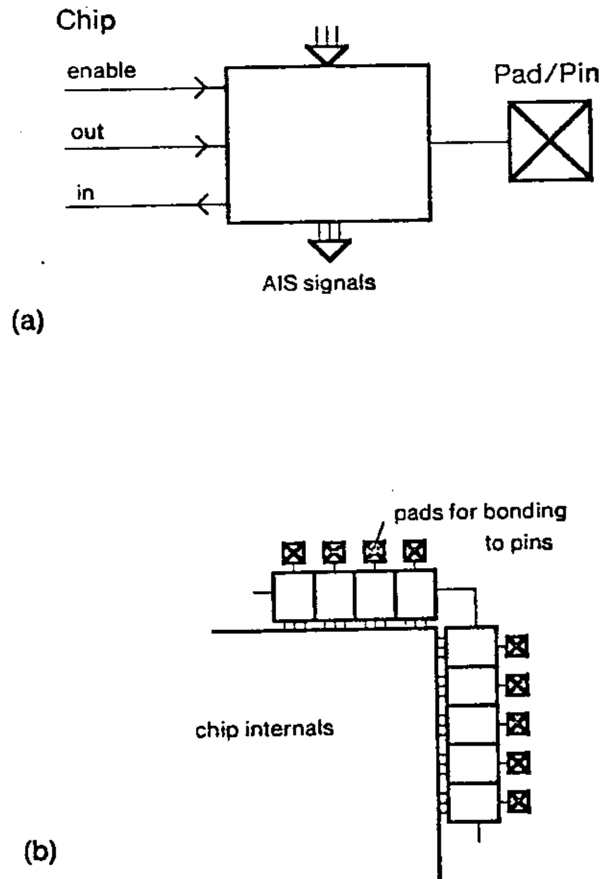

Figure 5-3: In-situ testing by decoupling pins from the chip's internal circuitry. (a) The logic structure of the pin circuits. (b) Pin circuits arrayed around the chip internals to connect to pad. Notice that the AIS signals thread through all pin circuits.

only AIS access to the signals that connect to the chip internals. For this test, the tester needs interface only to the few pins that control the AIS functions. Thus if individual die are being test before packaging, only a half-dozen points need to be probed: power, ground, and four AIS signa . Once the chip is determined to work, the pin drivers are tested separately, using AIS to drive a pin and using a simple multiplex switch on the hardware tester to verify the operation of the pin under test.

The effect of the circuitry that decouples the chip from the pins is to place a sentinel at a key interface in a digital system. It is analogous to tracing a software procedure by examining all calls to it, the calling parameters, and the return values. The sentinel need not be used only to control chip pins. For example, we could design printed-circuit boards so that each pin on the board connector is

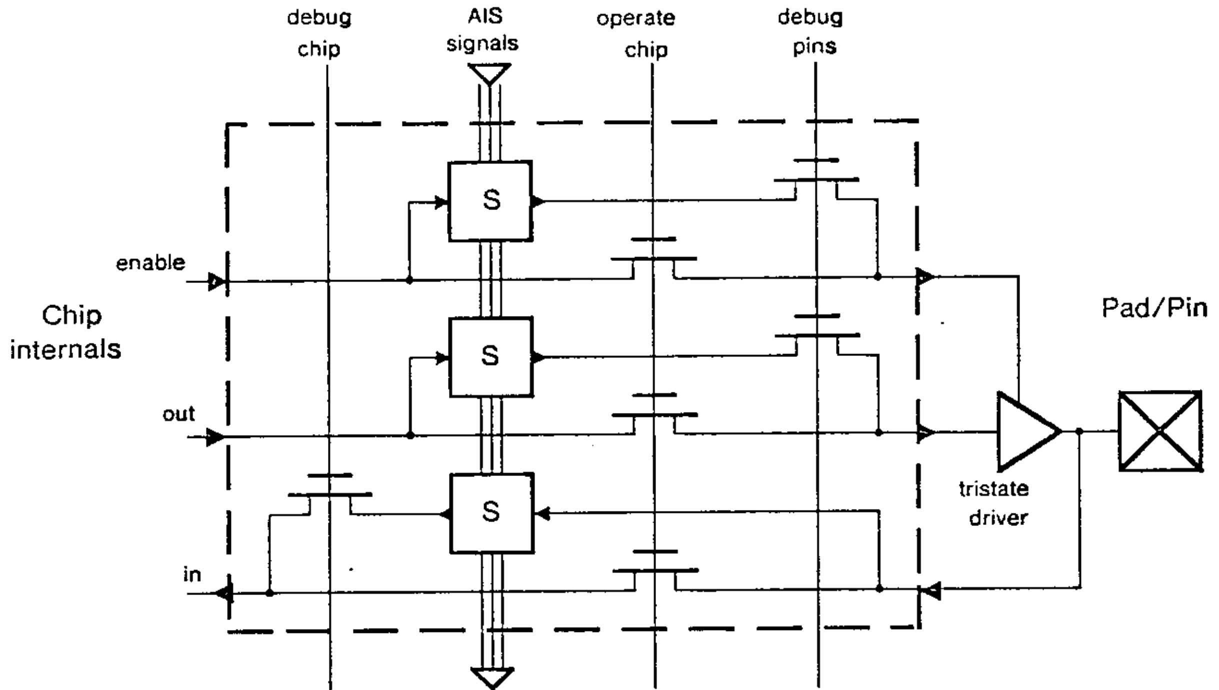

Figure 5-4: A sketch of the pin circuit for decoupling the chip internals from the pins. If the OPERATE signal is high, the pins are connected to the chip internals. Otherwise, if DEBUG CHIP is asserted, the chip internals are driven and sensed by three bits of state (S) accessible using an AIS mecahnism. Alternatively, if DEBUG PINS is asserted, the three bits of state control the pin only, thus allowing chip-to-chip interconnections to be tested. Each bit of state S has controls that allow it to be loaded from its inputs, to be read off the chip or to be set from of the chip.

driven in this way, perhaps using an integrated circuit that conveniently packages the functions sketched in Figure 5-4. The board itself might contain conventional TTL packages that have no provision for AIS connections.

5.2.2. Muffler extensions

Although the muffler design was used only to read internal state and not to set it, a simple extension in MOS technology will make the technique bidirectional. Figure 5-5 shows how MOS switches can connect the DATA wire to the bit of state to be sensed (READ signal asserted) or set (READ signal not asserted). This design has a number of advantages over SISO:

- Bits of state can be set individually. Even though the READ signal is applied to all bits of state, the circuit can be designed so that only that bit with the selection switch closed will be altered. (For example, if READ is normally high, the wire R will be normally charged to the state S, so that even when READ is brought low, the state will be set to R = S unless the selector switch is closed, in which case it will be set to the value on the data wire.)

- A single bit of state is constantly reported externally, and can thus be observed at high speed. By contrast, a shift chain will require hundreds of shifts to observe each value of an internal signal. This ability to sense a signal constantly gives rise to the next two

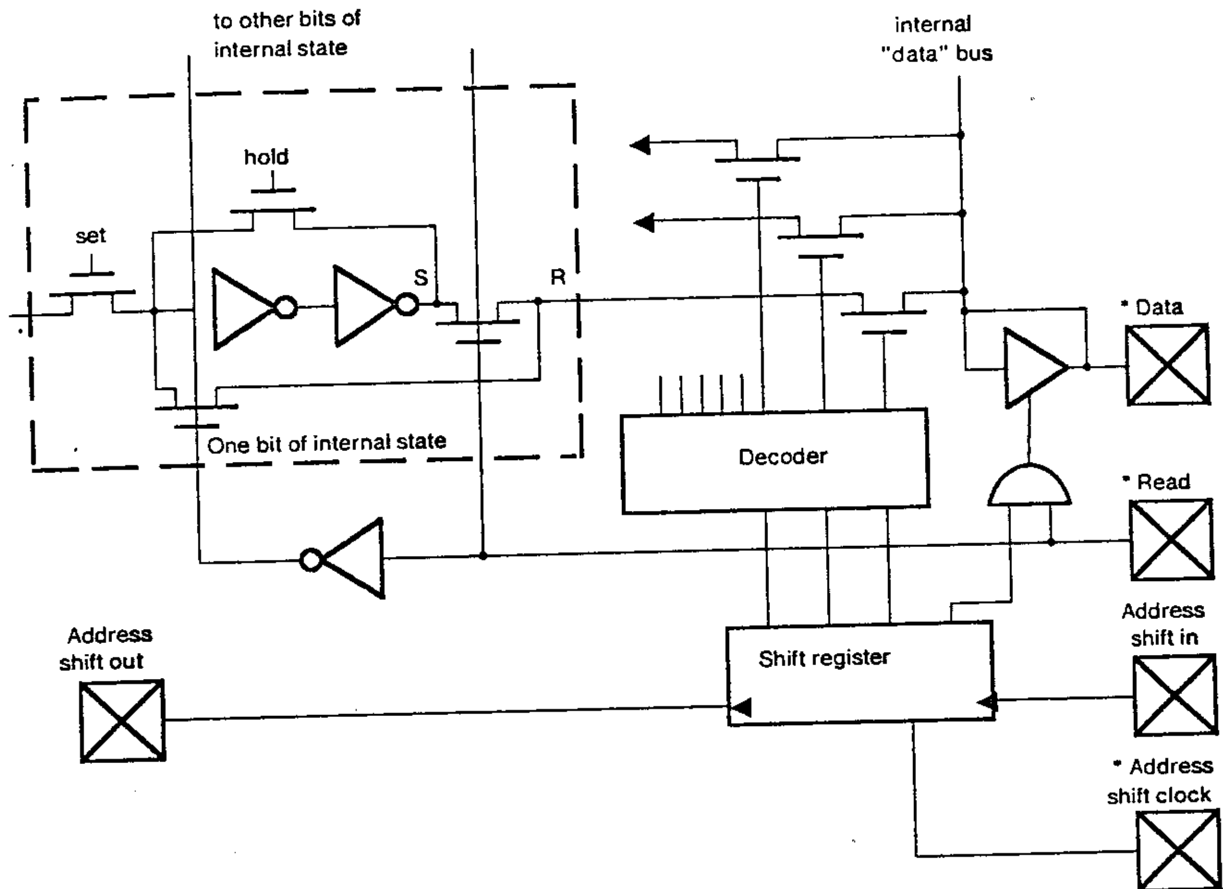

Figure 5-5: Extensions to the muffler scheme. A bit of internal state can be sensed (READ high) or set (READ low) via the DATA pin. In order to allow many separate chips to use mufflers, the muffler address shift registers are linked on a chain using ADDRESS SHIFT IN and ADDRESS SHIFT OUT. The remaining pins are bussed together for all mufflers.

advantages as well.

- The signature-analysis idea can be used for testing. The DATA pin is connected to a signature-computing circuit in the tester, and the muffler address is set to select a particular internal signal. Now the system is put through a fixed test sequence and the signature is computed and compared with a known good value. Note that during this test, the system can be operated at full speed.

- The muffler can be set to select reporting of an internal signal that yields useful performance information. (This idea was used on the Dorado [McDaniel 80].) Because we can observe internal events at full speed, we can operate the system normally and count events, measure arrival-time statistics, etc.

The muffler appears to have a disadvantage compared to SISO in that a fixed address is required to address each bit. This property would appear to make it infeasible for use on chips, because all instances of a particular chip type in a system would decode the same muffler addresses and report data concurrently. This problem can be remedied by putting the muffler address bits on a shift register and arranging that a chip will drive the data wire only if it detects an odd address. Thus if each chip has an n-bit address shift register, it can connect 2^{n-1} different signals to the data wire. A system with m chips will have a shift chain that is only mn bits long that provides access to $s = m2^{n-1}$ bits of internal state. By contrast, if the internal state is connected directly to the shift register (SISO), the shift register needs to be s bits long, or a factor of $2^{n-1}/n$ longer than the muffler chain. These comparisions show that the muffler allows quick access to a particular bit, but also show that SISO provides a faster mechanism to obtain the entire accessible state of the system.

The pin requirements of the two schemes must also be compared. The SISO technique (Figure 4-4) can be operated with three pins: SHIFT DATA IN, SHIFT DATA OUT, and SHIFT CLOCK. The first two pins are chained from chip to chip and the other is are bussed together. The simplest muffler scheme requires five pins: ADDRESS SHIFT IN, ADDRESS SHIFT OUT, ADDRESS SHIFT CLOCK, DATA, and READ. The first two pins are chained from chip to chip and the remaining three are bussed together. However, the READ pin can be avoided by using the ADDRESS SHIFT CLOCK pin as the READ signal. While the ADDRESS SHIFT CLOCK is high, an internal bit is read onto the DATA bus; while it is low the same bit is written from the bus. Thus the standard cycle for shifting in addresses will read and re-write a number of internal bits. If we ever want to change the internal state, we simply arrange to place the new value on the DATA wire while the ADDRESS SHIFT CLOCK is low.

5.2.3. On-chip signature analysis

One of the difficulties with SISO testing is that a great many shift steps are required to read out an internal state and that the operation of the system must be halted while this shifting is done. An alternative is to build on-chip signature checkers to reduce the amount of data that must be reported externally. Although the muffler mechanism allows checking a single signature while the circuit operates at full speed, on-chip signature checking could check many different signals at once. These could be checked with separate signature generators or with a single circuit that hashes together the states of several signals at once.

In addition to generating signatures on the chip, we might also consider including an on-chip ROM that contains known good signatures to verify the computed signatures.

A scheme called BILBO (Built-In Logic Block Observer) [Koenemann 79] incorporates many of these ideas and can be used to generate pseudo-random test patterns as well.

5.2.4. On-chip or on-wafer testers

Imagine that instead of receiving a single die back from fabrication, the designer received an entire test system in the form of a wafer to which she had only to connect power and a terminal (via a standard RS-232 interface). The wafer contains several copies of the designer's chip, some of which are already interfaced to a special test system. The test system provides the designer with convenient hardware and software for performing tests on the chip. Assuming the designer provided ways of accessing internal state, the test system can be programmed to set and retrieve this state, and moreover, can be connected to a larger computer for more sophisticated testing. There are several advantages to a test system of this kind:

- Custom IC designers do not have to buy or build their own testing systems. Moreover, they are able to take advantage of common development on one test system, a new version of which they get every time they fabricate a new chip.
- Even signals which are not going to be connected to pins could be connected to the test system. This would greatly enhance the ability to test and debug the chip.

The on-wafer tester is an excellent example of a hardware module fabricated along with a design to help debug or characterize the design. A less powerful example of such a debugging module is a *timing sampler* [Frank 81] which accurately measures the time of occurrence of an on-chip signal.

5.2.5. Summary of access to internal state

To take full advantage of the many uses of access to internal state, systems must be designed so that the appropriate internal signals can be examined. For example:

- Manufacturing testing. Access is needed to any state that cannot be set or read easily by the normal input/output pins of the chip. It is possible to analyze a gate-level design to determine whether each signal can be *controlled* from the outside (i.e., set to both binary values) and whether each signal can be *observed* from outside. Programs such as TMEAS [Grason 79] can analyze a circuit and report how difficult it is to control and observe signals internal to the circuit. If an important internal signal is found to be hard to control and/or observe, the circuit can be redesigned to improve its testability.
- System testing. Extra provisions, such as those described in section 5.2.1, are needed to test an entire system without removing components .
- Debugging. The needs of debugging may exceed those of testing. For example, we may wish to be sure that the entire internal state of a chip can be set to a given value for debugging, while testing requires only that the state of small portions under test be controlled.
- Performance measurement. Signals that are important for performance measurement must be sensed directly, whether or not they are needed for testing and debugging. The reason is that if only a single signal is being reported (e.g., using mufflers), we can measure the activity on only the one signal. Thus a "dirty miss" signal in a cache must be available as a single signal; we will not be able to detect "dirty" and "miss" separately and perform the AND function in the performance monitor.

- Initialization and configuration. As we observed in section 4.2.2, paths for setting internal state can be used to configure or initialize a system. We can enable or disable individual chips, load memories for microcode or table-lookup functions, etc. These requirements may demand that more signals be accessible via AIS paths. If the in-situ methods are used (section 5.2.1), they allow chips to be operated completely independently of their environment, and thus clearly provide full initialization and configuration control.

It is important to emphasize that a special AIS mechanism is not always the best way to access internal state. What is required is that there be *some* way to set and reveal the state. Consider, for example, a 16K dynamic MOS RAM chip. It has 16,384 bits of internal state in its memory, 128 bits of state in its sense amplifiers, and 14 bits of state in its two address registers. The 128 bits of sense amplifier state can be read out of the chip by a series of operations on the pins (page-mode CAS cycles). Likewise, the 16,384 bits stored in the memory can be read (conventional RAS, CAS cycles). Unfortunately, the contents of the address registers cannot be detected outside the chip, and to make matters worse, the operations for reading the other state destroy the contents of these registers.

5.3. Organizing and using a debugging system

Tools for debugging a design can be integrated into the system used to design the chip itself. In the early stages of a design, the designer debugs using simulations, while in the later stages, the debugging and testing are performed on the chip itself. The chip simply represents the most accurate simulation of the design.

This approach leads the designer to build programs for testing and debugging from the very beginning of the design. The test programs written to explore high-level simulations are equally applicable at lower levels, even when first silicon is obtained. To achieve this effect, we need to cast all of the simulation levels used during the design into a common framework. The following sections outline how this can be accomplished.

5.3.1. Names

A standard use of signal names links all levels of a design. In this way, a functional simulation at one level can be compared to a simulation at another level. In higher-level abstractions, many signals may be treated as one name, such as a register that need not name its bits. At the lowest level, geometry for a design may have very complex paths for a single signal (e.g., a clock). The details of the routing and placement of these paths may have a profound effect on the performance of the design, but from the point of view of a functional simulation, they constitute one signal.

The hierarchical nature of design specifications gives rise to a hierarchical notation for names. A 10-bit shift register composed of ten identical cells would give rise to names such as "cell1.shift-in", "cell2.shift-in" and "cell1.ground", where the first part of the name identifies an instance of the shifter cell and the second a signal name within that cell.

UNIVERSITY LIBRARIES
CARNEGIE MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA 15213

5.3.2. Organizing simulations

Simulations at different levels can all be organized to have identical interfaces. A simulator takes in a set of signal names and new values, computes until this new set of inputs creates a stable state, and then stops. Output signals that change may be reported from the simulation, or the simulation can simply stand ready to respond to requests to report the state of various internal signals. Using these primitives, debuggers can be built that provide all of the conveniences of software debuggers, such as printing out variable values and stopping at breakpoints, as well as all the conveniences of hardware debuggers, such as tracing signal values as a function of time.

We believe that simulations at three different levels of detail will suffice for our designs:

- High-level simulations, written in a programming language chosen for convenience by the designer. If the designer is also using a programming-language environment to help construct or simulate additional representations of the chip (e.g., microcode, or programs to "assemble" the chip [Johannsen 79, Steele 80]), the same environment is likely to be chosen for simulations. In this way, name correspondances are particularly easy to maintain.[10]

- Functional simulation of the chip circuit, with timing information retained. The chip's circuit is extracted by analyzing the mask geometry. In this way, any errors in the "automatic" algorithms of a design system (e.g., automatic PLA or ROM generation) can be detected by the simulation. Once extracted, the circuit can often be simplified greatly by analyzing it for common patterns (e.g., gates, or NOR-NOR combinations often used in PLA's, or even entire cells of a known kind), and then using more efficient simulations for the regular structures identified. In this way, we believe that functional (and timing) simulation of an entire complex chip is feasible.

- The chip itself, operated by a *tester* that can control its inputs and observe its outputs, and that can measure output timings fairly accurately. In order to report internal state, as required by the general model of our simulations, the designer must provide *access procedures* that specify the steps required to read and set state. Often, these procedures will simply invoke an AIS mechanism designed into the chip. Sometimes the access procedures will specify sequences of operations on the normal pins that can be used to access internal state (e.g., reading a memory location in a conventional semiconductor memory). Note that the access procedures may themselves be debugged by functional simulation.

This structure also illustrates one of the advantages of simulation: the functional simulation allows access to *all* internal signals, even ones that cannot be sensed with an AIS mechanism designed into the chip.

The structure itself eases debugging: each simulation operates on a different representation of the chip. As we put these simulations through a test sequence, the debugging software can compare the results at the different levels. Before a chip is fabricated, the first two levels are compared; when a

---

[10] It is for this reason that languages designed to describe hardware easily may not be the best ones to use for high-level representations. The designer will also need to write simulations, test programs, etc.

chip is available, all three levels are used. This is an advantage not commonly available to the software debugger, as she has not prepared even two different representations of her module that can be simulated.

5.3.3. Debugging strategies

The primitives outlined in the previous section can be employed in several debugging strategies. Although many of these strategies are identical to those commonly used for debugging software, the additional strategies described in this section are particularly applicable to hardware debugging.

Checkpoints. Access to internal state allows us to record the state of a simulation at any level and to restore that same state later. If an error is detected after a several simulation steps, we can reload the original state and simulate again, examining more internal state to catch the error as it occurs. Moreover, it may be possible to set the state of a simulation from state checkpointed from a simulation at a different level.

Searching. Simonyi [Simonyi 76] has suggested using *binary search* and checkpoints as a systematic way of tracking down errors, a technique usually practiced intuitively by programmers. The general idea is to track down errors by retrieving state before and after an error occurs and gradually narrowing down the window that contains the error until the window is only one clock cycle long.

Patching errors. Access to internal state allows us to "patch" internal errors in a chip. Whenever the functional simulator determines that a bit of state will not be set correctly by the unit under test, the AIS mechanism can be summoned to change the bit, and then the chip testing can proceed. In particular, microcode bugs can be fixed this way. The effect is to be able to check out a great deal of the chip's functions even though errors are known to exist.

5.3.4. Debugging at CMU

At CMU, we are building a structure for debugging chip designs that is based on the philosophy described in this section. The basic parts are shown in Figure 5-6. We assume that the designer may use one of a number of design systems to write high-level simulations and to develop mask geometry, which is output in a standard form (CIF [Sproull 80]). The geometry drives the fabrication process that builds a chip that is inserted in a tester. The tester need not be located in our laboratory; indeed, we are planning to access via the ARPANET a tester being built at Caltech [DeBenedictis 80]. The same CIF geometry drives a circuit extractor that determines a circuit for the FETS functional and timing simulator. We have chosen to extract the circuit for several reasons: (1) because the functional simulation and the fabricated chip are built from the same data, it is more likely that the simulation will reflect errors in the chip itself; (2) the technique does not require a high-level design system to provide a circuit representation of the design--indeed, many design techniques bypass this representation altogether; (3) the extraction technique can be applied to any design, regardless of the

design style or representations of the design system used to construct it.

Debugging and test programs are constructed in a convenient programming environment and access the three levels of simulation using a common interface. All of the standard facilities of a high-level language are thus available for writing test programs. The debugger provides a user interface and displays of internal state that are common to all of the simulations.

6. Conclusions

The MC68000 experience suggests that chip design is beginning to use concepts of modularity so familiar in software design to help reduce design errors. By using access to internal state and simulation, it is possible for the integrated-circuit designer to debug chips as easily as programmers debug software modules. Moreover, it appears that the chip designer may be better off than the programmer by simulating several different representations of the chip to validate a design.

There are, however, some dark clouds on this rosy horizon. We have confined our attention to testing synchronous sequential systems. As Seitz points out, the second half of the integrated circuit revolution will require chips composed of asynchronous elements [Seitz 80]. Although these elements may be synchronous internally, testing the entire chip will require grappling with an asynchronous system. Seitz argues that the only hope in preventing design errors is to use only legal compositions of proven elements. This may solve the design problem, but how are the circuits to be tested in manufacture?

Acknowledgements

The authors would like to thank all those who helped with this paper. Tom Gunter of Motorola Inc. and Skip Stritter of Nestar Systems provided information on debugging the MC68000. Bernard Peuto of Zilog Inc. provided information on debugging the Z8000. Mike Foster provided information required to simulate his pattern-matching cell. Alan Bell and Lynn Conway of Xerox-PARC provided the inspiration for the section on wafer-scale test systems. Guy L. Steele Jr., Ivan E. Sutherland, and Hank Walker offered comments on drafts of the paper.

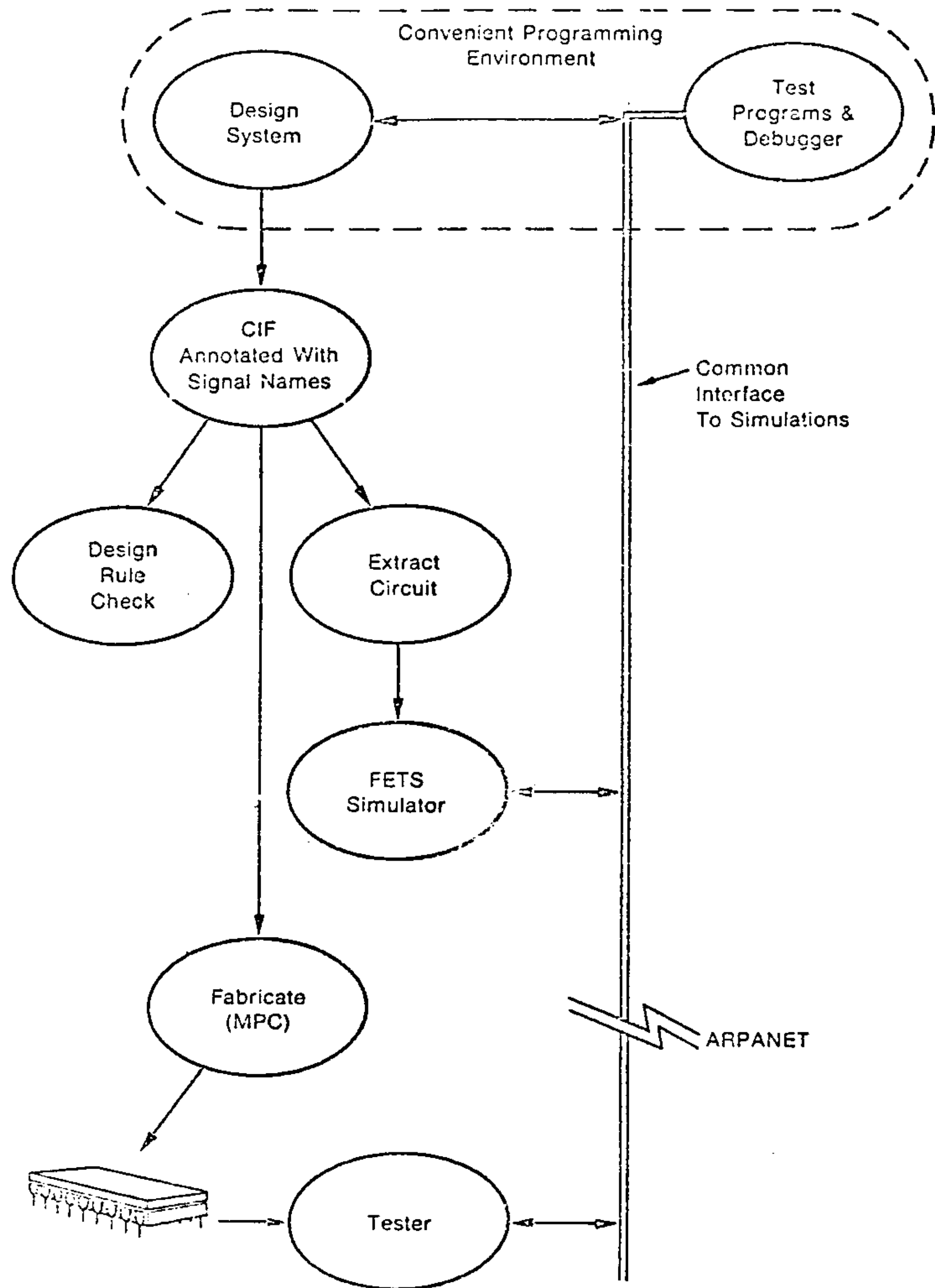

Figure 5-6: The organization of a system for debugging chip designs. Three levels of simulation are performed: high-level; functional and timing; and the chip itself, operated by a tester. Debugging and test software controls all three simulations identically.

References

[Baker 80] Baker, C.M., and Terman, C.
Tools for verifying integrated circuit designs.
Lambda 1(3):22-31, Fourth Quarter, 1980.

[Barbacci 77] Barbacci, M.R.
The ISPS Language.
Technical Report, Carnegie-Mellon University, Computer Science Department, 1977.

[Breuer 76] Breuer, M.A., and Friedman, A.D.
Diagnosis and Reliable Design of Digital Systems.
Computer Science Press, Inc., Potomac, MD, 1976.

[Bryant 80] Bryant, R.E.
An algorithm for MOS logic simulation.
Lambda 1(3):46-54, Fourth Quarter, 1980.

[Case 78] Case, G.R., and Stauffer, J.D.
SALOGS-IV: a program to perform logic simulation and fault diagnosis.
In *Proceedings of the 15th Design Automation Conference*, pages 392-397. June, 1978.

[Chawla 75] Chawla, B.R., Gummel, H.K, and Kozak, P.
MOTIS - An MOS timing simulator.
IEEE Transactions on Circuits and Systems CAS-22(12):901-910, December, 1975.

[Conway 80] Conway, L., Bell, A., and Newell, M.
MPC79: A large-scale demonstration of a new way to create systems in silicon.
Lambda 1(2):10-19, Second Quarter, 1980.

[Darringer 79] Darringer, J.A.
The application of program verification techniques to hardware verification.
In *Proceedings of the 16th Design Automation Conference*, pages 375-381. June, 1979.

[DeBenedictis 80] DeBenedictis, E.P.
Caltech Arpa Tester Project.
Technical Report, California Institute of Technology, Deparment of Computer Science, April, 1980.

[DOD 80] *Reference Manual for the Ada Programming Language*
United States Department of Defense, 1980.

[Dowell 79] Dowell, R., and Newton, A.R., and Pederson, D.O.
SPICE, VAX version 2 user's guide
University of California, Berkeley, Department of EE and Computer Science, 1979.

[Eichelberger 78] Eichelberger, E.B., and Williams, T.W.
A logic design structure for LSI testability.
Journal of Design Automation and Fault Tolerant Computing 2(2):165-178, May, 1978.

[Foster 80] Foster, M.J., and Kung, H.T.
Design of special-purpose VLSI chips: example and opinions.
In *Proceedings of the 7th Annual Symposium on Computer Architecture*, pages 300-307. May, 1980.

[Frank 80a] Frank, E.H., and Sproull, R.F.
An approach to debugging custom integrated circuits.
Carnegie-Mellon University Computer Science Resarch Review , 1979-1980.

[Frank 80b] Frank, E.H.
FETS: Fast Eddie's Timing Simulator.
VLSI Document, Carnegie-Mellon University, 1980.
(in preparation).

[Frank 81] Frank, E.H., and Sproull, R.F.
Two timing samplers.
In *Proceedings of the Second Caltech VLSI Conference.* January, 1981.

[Frohwerk 77] Frohwerk, R.A.
Signature analysis: a new digital field service method.
Hewlett-Packard Journal :2-8, May, 1977.

[Gehringer 80] Gehringer, E., and Vegdahl, S.
The CMIC microassembler and its software support.
In Jones, A.K., and Gehringer, E.F. (editors), *The CM* Multiprocessor Project: A Research Review*, chapter 3.2pages 30-32. CMU, 1980.

[Gosling 80] Gosling, J. A.
The MUMBLE microcode compiler.
In Jones, A.K., and Gehringer, E.F. (editors), *The CM* Multiprocessor Project: A Research Review*, chapter 3.3pages 32-37. CMU, 1980.

[Grason 79] Grason, J.
TMEAS, a testability measurement program.
In *Proceedings of the 16th Design Automation Conference*, pages 156-161. June, 1979.

[Haken 80] Haken, D.
A geometric design rule checker.
VLSI Document V053, Carnegie-Mellon University, June, 1980.

[Hayes 80] Hayes, J.P., and McCluskey, E.J.
Testability considerations in microprocessor-based design.
IEEE Computer 13(3):17-26, March, 1980.

[Hennesy 81] Hennesy, J.L.
SLIM: a language for microcode description and simulation in VLSI.
In *Proceedings of the Second Caltech VLSI Conference.* January, 1981.

[Hon 80a] Hon, R.W., and Sequin, C.H.
A Guide to LSI Implementation.
SSL, Xerox Palo Alto Research Center, January, 1980.

UNIVERSITY LIBRARIES
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA 15213

[Hon 80b] Hon, R. W.,.
IC fabrication for the independent chip designer.
Lambda 1(1):6-9, First Quarter, 1980.

[Johannsen 79] Johannsen, D.
Bristle blocks: a silicon compiler.
In *Caltech Conference on VLSI*, pages 303-310. January, 1979.

[Knuth 71] Knuth, D.E.
An empirical study of FORTRAN programs.
Software - Practice and Experience 1(2):105-133, 1971.

[Koenemann 79] Koenemann, B., Mucha, J., and Zwiehoff, G.
Built-in logic block observation techniques.
In *Test Conference Proceedings*, pages 37-41. October, 1979.

[Lampson 80] Lampson, B., and Pier, K. A.
A processor for a high-performance personal computer.
In *Proceedings of the 7th Annual Symposium on Computer Architecture*, pages 146-160. May, 1980.

[McCaw 79] McCaw, C.R.
Unified shapes checker -- a checking tool for LSI.
In *Proceedings of the 16th Design Automation Conference*, pages 81-87. June, 1979.

[McDaniel 80] McDaniel, G.
private communication.
1980.

[McWilliams 80] McWIlliams, T.M.
Verification of timing constraints on large digital systems.
In *Proceedings of 17th Design Automation Conference*, pages 139-147. June, 1980.

[Mead 80] Mead, C., and Conway, L.
Introduction to VLSI Systems.
Addison-Wesley, Reading, MA, 1980.

[Megatest 80] *Q2/60 user's manual*
Megatest Corp., 1980.

[Mitchell 79] Mitchell, J. G., Maybury, W., and Sweet, R.
Mesa Language Manual.
CSL 79-3, Xerox PARC, April, 1979.

[Motorola 79] *MC68000 reference manual*
Motorola Inc., 1979.

[Muehldorf 81] Muehldorf, E.I., and Savkar, A.D.
LSI Logic Testing - An Overview.
IEEE Transactions on Computers C-30(1):1-17, January, 1981.

[Myers 79] Myers, G.J.
The Art of Software Testing.
Joh Wiley, 1979.

[Newton 79] Newton, A.R.
Techniques for the simulation of large-scale integrated circuits.
IEEE Transactions on Circuits and Systems CAS-26(9):741-749, September, 1979.

[Parker 78] Parker, A., Thomas, D., Siewiorek, D., Barbacci, M., Hafer, L., Leive, G., and Kim, J.
CMU Design Automation System: an example of automated data path design.
In *Proceedings of the 16th Design Automation Conference*, pages 73-80. June, 1978.

[Peuto 79] Peuto, B.
Architecture of a new microprocessor.
IEEE Computer 12(2):10-21, February, 1979.

[Preas 77] Preas, B.T., and Gwyn, C.W.
Architecture for contemproary computer aids to generate IC mask layouts.
In *Eleventh Annual Asilomar Conference on Circuits, Systems and Computers*, pages 353-361. November, 1977.

[Roth 67] Roth, J.P., Bouricius, W.C., and Schneider, P.R.
Programmed algorithms to compute tests to detect and distinguish between failures in logic circuits.
IEEE Transactions of Electronic Computers EC-16(5):567-580, October, 1967.

[Seitz 80] Seitz, C.
System Timing.
In *Introduction to VLSI Systems*, chapter 7pages 218-254. Addison-Wesley, 1980.

[Simonyi 76] Simonyi, C.
Meta-programming: A software production method.
CSL 76-7, Xerox PARC, 1976.

[Sproull 80] Sproull, R.F., and Lyon, R.F.
A CIF Primer.
In *A Guide to LSI Implementation*, chapter 7pages 79-121. Xerox, 1980.

[Steele 80] Steele Jr., G.L., and Sussman, G.J.
Design of a LISP-based microprocessor.
Communications of the ACM 23(11), November, 1980.

[Stritter 78] Stritter, E.P., and Tredenick, N.
Microprogrammed implementation of a single chip microprocessor.
In *Proceedings of the 11th Annual Microprogramming Workshop*. December, 1978.

[Stritter 79] Stritter, E. P. and Gunter, T.
A microprocessor architecture for a changing world: the Motorola 68000.
IEEE Computer 12(2):10-21, February, 1979.

[Sutherland 79] Sutherland, I.E., Molnar, C.E., Sproull, R.S., and Mudge, J.C.
The Trimosbus.
In *Proceedings of the Caltech Conference on VLSI*, pages 395-427. January, 1979.

[vanCleemput 79] vanCleemput, W.M.
Hierarchical Design for VLSI: Problems and Advantages.
In *Proceedings of the Caltech Conference on VLSI*, pages 259-274. January, 1979.

[Williams 73] Williams, M.J.Y., and Angell, J.B.
Enhancing testability of LSI circuits via test points and additional logic.
IEEE Transactions on Computers C-22(1):46-60, January, 1973.

[Wulf 76] Wulf, W.A., London, R.L., and Shaw, M.
An introduciton to the construction and verification of Alphard programs.
IEEE Transactions on Software Engineering SE-2(4):253-265, December, 1976.

[Zilog 79] *Z8000 technical manual*
Zilog Inc., 1979.