CMOS Design of ADM-PCM Codec Chip using Silicon Compiler with Performance Evaluation

by

Brent Robert Petersen

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical Engineering

Waterloo, Ontario, 1987

ISBN 0-315-38754-8
email: b.petersen@ieee.org

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

B. R. Petersen

I further authorize the University of Waterloo to reproduce this thesis by photo-copying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

B. R. Petersen

The University of Waterloo requires the signatures of all persons using or photo-copying this thesis. Please sign below, and give address and date.

Abstract

A silicon compiler is a computer program that generates IC layouts from a high-level specification. In the SPIL silicon compiler a digital signal processing algorithm is specified in a language similar to Pascal, and IC layouts are generated using a simple register-transfer architecture with a data path controlled by a finite state machine. For a silicon compiler to be an effective design automation tool it is essential to include some type of performance evaluation to estimate power dissipation, layout area and propagation delays of the generated chip layout.

A design aid called EPAD is used for performance estimation of propagation delays, power dissipation and silicon areas of CMOS VLSI circuits. The objective is to provide the designer with an analysis of the IC layout. The designer can use this analysis to change the algorithm or intermediate levels of silicon compilation.

SPIL and EPAD are used in the design of a chip which compresses and decompresses speech by performing conversions between pulse code modulation and adaptive delta modulation. The algorithms for these conversions were written in the SPIL language, and layouts were generated and combined into one coder-decoder (codec) chip. An EPAD analysis was performed on this chip. Simulation files obtained from EPAD made it possible to identify possible design errors and to predict the maximum operating frequency. This was followed by fabrication and testing of the chip. Test results on fabricated chips compare favourably with EPAD predictions. The results made it possible to evaluate the effectiveness of the silicon compiler and to calibrate the performance evaluator.

The goal of this research is to show that the state of the art is advanced enough that a chip can be efficiently designed using SPIL with a EPAD and that the chip can satisfy the requirements of specific applications.

Acknowledgements

Few students work solely by themselves; they work with the assistance of professors and colleagues. This thesis would not be complete without acknowledging the contributions of those who have assisted me.

Professor Mohamed I. Elmasry was a superb supervisor. Without his patience my work would never have been successful.

I would never have been able to meet the chip fabrication deadlines without the help of Dan Salomon, one of the original authors of the SPIL silicon compiler.

Brian White, the author of EPAD and of the program to partition finite state machines, set the perfect example for me by being so methodical.

The Canadian Microelectronics Corporation (CMC) assisted by providing the VLSI implementation services. CMC's pride in its work is exemplified by Peter Ellis who continued the design rule checking of my chip even up to a few days before Christmas. He did this by logging on to the CMC computer from his home terminal.

Finally, I am grateful to the Natural Sciences and Engineering Research Council (NSERC) and Control Data Canada (CDC) for providing me with direct and indirect funding.

Table of Contents

List of Tables

List of Illustrations

CHAPTER 1

Introduction

Improvements in VLSI fabrication technology have increased the number of transistors that can be put on a silicon chip. The major improvements have been due to a reduction in the size of features used in photolithographic fabrication processes. With these reductions have come improvements in circuit performance. The three major performance criteria used to evaluate a chip are the area, delay and power. Area refers to the silicon area consumed by the circuitry of the chip. Delay refers to the time required for signals to propagate through the chip. Power refers to the power dissipated by the chip during its operation.

Improvements in technology allow more circuit functions to be placed within a given area. Alternatively, the same circuit function can be performed using less dense circuitry. Therefore, a given design does not have to be as elaborately constructed as a design in an earlier technology, to achieve a similar performance.

Increases in circuit density imply that signals have shorter distances to propagate and that capacitive and resistive loadings on transistors are reduced. However, a trade off exists: reducing the size of transistors also reduces the ability of transistors to drive their capacitive or resistive loads. Recent advances in

technologies, such as Bipolar and CMOS, have kept a reasonable balance between these two trade offs in order that propagation delays decrease in an improved technology. Since propagation delays decrease, a calculation, required to be performed within a fixed time, can be implemented with less elaborate approaches and less fine tuning, compared to the same circuit implementation in an earlier technology.

Increases in circuit density tend to increase the power dissipation for a fixed chip area with a fixed chip power supply voltage. This problem has been overcome in the past by reducing the supply voltage, from a standard value of 12 volts to 5 volts (and may soon drop to 3.3 volts). Power dissipation problems have also been solved by improved chip packaging. As technology improves, power dissipation problems must be overcome.

The complexity of designing chips increases with the number of transistors. In order to manage this complexity, design methods have become hierarchical, as have the representations of designs. The lowest level of representing a design is a description of the masks used in the fabrication process; an example of this representation is the CIF format [1]. A higher level of design description can be achieved using procedural layout languages such as ICEWATER [2] and IGLOO [3] which describe transistors, their relative placements and their interconnections in a layout. A higher level of description is a netlist, a description

of transistors or logic gates and their interconnections without any reference to their positions in a layout. Sometimes the components of a netlist can be associated with a standard set of cells which contain mask or procedural language descriptions of each netlist function. An intermediate-level of description is about the Boolean functions that the circuit is to perform. These functions can further be grouped into blocks which describe the structure and the architecture of a chip. The highest level is a behavioral or functional description which specifies the function of the chip without implying how that function is implemented.

To accommodate the wide range of descriptions, numerous CAD programs, called tools, exist to generate design descriptions or make conversions between the different levels of design descriptions. The most simple designs can be generated by manipulating representations of the masks used in fabrication. This is facilitated by a CAD tool called a layout editor, such as Caesar [4] or Magic [5]. A higher-level structural description can be generated using a schematic editor from graphical input.

CAD tools perform conversions between many levels of representation. A layout extractor analyses a mask representation and generates a netlist. A performance evaluator analyses a mask representation and estimates propagation delays, power dissipation and silicon area. The objective of a performance evaluator is to provide the designer with performance data to be used in

evaluating or improving design. Above the level of an extractor are procedural layout language compilers, such as ICEWATER and IGLOO, which convert a procedural layout description into a mask description. Another tool called a module generator performs a conversion from an intermediate-level description, such as Boolean equations, into a lower-level description, such as a netlist or mask layout. Module generators can be used to efficiently generate parts of a chip such as Programmable-Logic Arrays (PLA), Random-Access Memories (RAM) or Read-Only Memories (ROM). At the highest level of design automation is a silicon compiler. It performs a conversion from a designer's high-level structural or functional description into a chip layout. The silicon compiler may transform the high-level description into many intermediate-level descriptions before finally generating the layout description [6].

An important point to consider when using CAD tools to generate a chip is that the lower the level of design detail, the greater the amount of design effort. If designs are performed using higher levels of design description, less effort is required by the designer because the CAD tools manage more of the details. The trade off with using higher-level design tools is that the performance of the chips they generate is inferior to the performance that can be achieved using lower-level design tools. This trade off is due to less flexible approaches and assumptions that the CAD tools are forced to make in order to hide details from the designer.

The goal of this research is to show that the state of the art is advanced enough that a chip can be efficiently designed using a silicon compiler with a performance evaluator and that the chip can satisfy the requirements of specific applications.

Chapter 2 contains a more detailed discussion about silicon compilers as well as a description of the specific silicon compiler that was used to design a Digital Signal Processing (DSP) chip. This chapter also discusses the methods a designer can use to improve the performance of the chip by using the data obtained from a performance evaluator.

Chapter 3 discusses the tools used to support the design of a chip: the performance evaluator (EPAD), the logic simulator, the Finite-State-Machine (FSM) partitioner and the data-path-to-FSM interconnection program.

Chapter 4 discusses the coder-decoder chip (codec) which was designed using the SPIL. This chip compresses and decompresses speech by performing conversions between two speech coding formats, Adaptive Delta Modulation (ADM) and Pulse Code Modulation (PCM). This chapter discusses chip design, performance estimation, simulation and preparation for fabrication.

Chapter 5 includes test results and comparison to EPAD predictions. It also includes suggested improvements to the silicon compiler and performance evaluator. Chapter 6 contains the conclusions of this research.

CHAPTER 2

Design Automation and Silicon Compilation

A silicon compiler accepts a functional or behavioral language description as input and generates a low-level description of chip fabrication masks as output. This chapter will discuss the wide variety of silicon compilers as well as the similarities and differences between them and the SPIL silicon compiler. The majority of this chapter discusses SPIL since it is the silicon compiler which was used for the codec chip design.

2.1. Introduction to Silicon Compilation

The high degree of design automation achieved using a silicon compiler is due to the number of design details which the compiler manages. Ideally, the designer need only to specify the highest overall function that the chip is to perform. This design style is very efficient in terms of the time it takes to design a chip. However, the produced design suffers from a lack of application flexibility and chip performance. This is due to the restrictions in the compiler's input language and compiling methods. Restrictions in compiling methods result, for example, from using fixed circuit architectures.

The languages used to specify a designer's algorithm are divided into two main categories: structural and functional. Structural languages hierarchically describe the interconnections of parts of a chip. The silicon compilers YASC [7], FIRST [8] and Apollon [9] accept as input structural descriptions of a data path. YASC and FIRST are based on a data flow architecture. Apollon is based on two data buses. Functional, also known as behavioral, languages describe what kind of function the chip is to perform without necessarily implying how the chip is to perform it. A sub-class of functional languages is architectural languages [10] which directly imply certain architectural features. Architectural languages are advantageous in that they compile faster than a pure functional language because the compiler does not have to choose between architectural features. The disadvantage of architectural languages is lack of flexibility in what the compiler can put out. The silicon compilers MacPitts [11] and SPIL [12,13] accept architectural input language descriptions.

The circuit architectures generated by silicon compilers tend to be fixed in various aspects. This makes compilation easier, but reduces the number of applications of the produced chip because the architecture has been designed for a specific performance. The SPIL, MacPitts and Apollon architectures are distinctly divided into a data path and a control path. SPIL uses a single bus data path. Apollon uses a two-bus data path. MacPitts uses an arbitrary number of buses in the data path. The more buses, the more parallelism in the

architecture. However, more buses mean that the architecture is more complex to generate and the performance of the architecture is not as consistently predictable. The control path for MacPitts is based on microcoding. The control path for SPIL is based on an FSM. The architecture for YASC is based on its data flow input language. The architecture contains asynchronously connected data flow blocks and it is well suited to parallel calculations. However, a large data flow architecture is more difficult to lay out than simple data bus based architecture. The architecture for the FIRST silicon compiler is based on serial processing elements connected with globally-clocked latches between them. This architecture is well suited to high-speed DSP applications, but it is not very flexible in its ability to implement a variety of algorithms. There are no looping constructs in the FIRST language such as `WHILE-DO` loops.

Since there are many silicon compilers which are capable of generating a wide variety of architectures, it is necessary to choose a specific silicon compiler for a particular design. The major advantage of using the SPIL silicon compiler is its ability to communicate to the performance evaluation tool EPAD.

2.2. Silicon Compilation using SPIL

In order to demonstrate the aspects of chip design using a silicon compiler, a specific silicon compiler, SPIL, was used. The name SPIL is an acronym which means **S**implified **P**ascal **I**nto **L**ayout. SPIL was intended to generate chips for digital signal processing applications. As its name implies, SPIL accepts as input a designer's high-level program which is written in a language similar to Pascal [14] and then from the designer's program, SPIL generates a mask-level description of a chip layout. The format of the mask-level description is the **C**altech **I**ntermediate **F**ormat (CIF).

Since SPIL's input language very closely resembles Pascal, the language describes the function that the chip performs. However, the components of the SPIL language, such as variable names, directly imply architectural features, such as registers. Thus, the SPIL language is an architectural language [10].

SPIL translates an architectural language into a mask-level description of the layout. These two levels of design description are nearly at opposite ends of the spectrum in terms of containing high and low level design details. This results in a high level of design automation because the designer is not concerned with all of the details of the intermediate-level descriptions of a design, such as a logic description. In addition, the architectural implications of the SPIL language mean that compiling can be done more easily than a behavioral

language.

SPIL uses a fixed architecture with a single-bus data path controlled by a **F**inite **S**tate **M**achine (FSM). This architecture does not have any data path parallelism; however, it is very regular and very efficient to generate. The process of generating the layout does not involve explicit translations to logic circuits or transistor circuits. For example, SPIL chooses cells from the SPIL cell library and places the cells in a matrix arrangement to generate the data path. The architecture is still general enough to implement any numerical algorithm.

More details of SPIL will be described in the following sections, including types of designs that SPIL is capable of generating, additional language issues, the steps to compile a program, the circuit architecture, timing considerations and, finally, some design trade off techniques.

2.2.1. SPIL Usage

The first issue that a designer must consider is whether SPIL is the correct choice of tool to design a chip. This suitability of SPIL will depend greatly on three criteria: the expertise of the designer, the performance required from the chip and the time available in which to design it. These three criteria are described in more detail below but they are merely heuristics to guide a potential designer in selecting SPIL. A more quantitative argument based on the required

performance of the chip can be found in chapter 4 (ADM-PCM Codec Chip Using SPIL). ADM means **A**daptive **D**elta **M**odulation; PCM means **P**ulse **C**ode **M**odulation [15]. The word *chip* in this discussion is synonymous with *chip set*.

As to the first criterion, SPIL does not require that the designer have very much expertise in logic and circuit design, compared to that which may be required to do a similar design with lower-level tools such as a schematic or lay-out editor [4]. In fact, the expertise required to design a chip using SPIL lies in the design trade offs that occur when choosing different high-level statements to perform a task. A more detailed explanation of these trade offs can be found in section 2.2.6 (Design Trade Off Techniques).

The second criterion is the type of performance required from the chip. Since the SPIL output data path does its calculations sequentially, this implies low-speed applications, such as speech processing or communications, requiring low data rates such as 19.2 kilobaud. SPIL should only be used if on-chip memory requirements are low. A typical design contains 10 registers up to a maximum of approximately 22 bits wide. Since SPIL can be used to generate one or a few **A**pplication **S**pecific **I**ntegrated **C**ircuits (ASIC), SPIL chips may be much more desirable than having an off-the-shelf microprocessor and all its associated peripheral circuitry.

The third criterion is the time available in which to generate a chip. Using SPIL requires less design time than lower-level design tools. The design time can be reduced due to two reasons. The first reason is that SPIL manages intermediate-level details which would otherwise require more designer effort when lower-level design automation tools are used. The second reason is increased reliability in getting a functionally correct design, because of a reduced chance of designer error. An indication of reduced design time is given in chapter 5 (Test Results and Suggested Enhancements).

2.2.2. Input Language

It is shown in this section how the SPIL language is an architectural language. The functional aspects of the SPIL language stem from its similarities to Pascal. The architectural aspects stem from language constructs which directly imply architectural features in the layout.

The SPIL input language was designed to adhere as closely as possible to Standard Pascal [14]. However, it was necessary to make changes to Pascal because it was never intended as a hardware description language., These changes were made for efficiency. However, the high degree of similarity between the two languages makes the SPIL language easy to learn for designers who are already familiar with Pascal. Some of the important differences

between the SPIL language and Pascal will be summarized here.

The SPIL language does not support the type `real`, nor the type constructors `set`, `record` and enumeration. However, it does support the `integer` type, since integer arithmetic can be efficiently done on a chip. It provides for subrange types and one-dimensional arrays. It provides for pointer types, but since it has no dynamic memory allocation, pointers refer to static locations only. The function `ADDR` has been added to the SPIL language to return the address of a variable or array element. The SPIL language has the added types `input_port` and `output_port` for describing chip input and output registers.

Any integers declared in a SPIL program correspond on a one-to-one basis with storage registers created for them in the data path. This is an example of why the SPIL language is an architectural language.

The SPIL language has binary-mask constants with *don't care* bits of the form `??10B`. The question marks are the *don't care* bits. The `B` means binary. These help hand optimization of data path conditional tests. This is a very good example of an addition to the SPIL language from Pascal for reasons of efficiency. Bit masks can greatly reduce the size of the control path. This will be described in more detail in section 2.2.4 (The Circuit Architecture of SPIL).

The SPIL language has some predefined variables and constant names. All such added symbols begin with an underscore to distinguish them from ordinary variables. There is a predefined variable name for each of the input and output registers of the computational units in the data path to permit hand optimization of critical computations in the case where the automatic code improvement is inadequate. An example of this is shown in the SPIL program in appendix A (SPIL Codec Files). Some examples of these variables and constants follow. The predefined symbolic constant `_data_width` controls the width of the data path. The predefined variables `_add_in_1` and `_add_in_2` represent the two input ports of the adder computational unit. The predefined variable `_add_out` represents the output port of the adder computational unit.

The SPIL language has some additional operators for shifting and complementing that were taken from the C language. These make low-level computation with the shifter and complementer easier. For example, the expression `X << 1` means the value of `X` shifted to the left one bit. The least significant bit will be padded with a zero. The most significant bit will be discarded.

The SPIL language has most of the control structures of Pascal, including `IF-THEN-ELSE`, `FOR-DO`, `WHILE-DO`, `REPEAT-UNTIL`, `GOTO`, and `CASE`, but does not include procedure invocation. The high overhead of procedure invocation implemented by finite state machine makes it impractical currently. The only

procedure definition that the designer is allowed is for a procedure called `_reset` which will be invoked when the chip reset input is enabled. Details of this will be discussed in the section 2.2.5 (Timing Considerations).

The SPIL language's similarities to a well-known language, Pascal, combined with its specific adaptations for hardware description, make it an effective language for describing chips.

In order to describe a chip using the SPIL language, the designer should begin with an algorithmic description in the form of a signal-flow graph, equations, or even a program in another language. All of these representations are well suited to step-by-step coding in the SPIL language.

A sample program to determine absolute value, illustrating some of the basic SPIL language features, is shown in Figure 2.1.

In the `CONST` section the width of the data bus is set to 8 bits. In the `VAR` section an input port called `input` and an output register called `output` are defined, as well as a register called `temp` to hold intermediate results. There is no `_reset` procedure in this example. The program reads in the input and stores it in the register `temp`, then tests the sign and takes the negative if necessary to determine the absolute value. The result is then moved to the output register.

```
PROGRAM Demo ;

CONST
    _data_width = 8 ;

VAR
    input  : input_port  CONNECT DOWNWARD ;
    output : output_port CONNECT UPWARD   ;
    temp   : integer ;

BEGIN
    temp := input ;
    If ( temp < 0 ) then  temp :=  - temp ;
    output := temp ;
END.
```

Figure 2.1. Demonstration SPIL Program

2.2.3. Compilation Steps

The steps involved in compiling a SPIL program are as follows (Figure 2.2). The program is stored in the file *demo.sp*. The SPIL compiler is then run by the command *spil demo.sp*. This command generates two new files. The first file, *demo.bm*, is a readable description of the arrangements of cells in the data path. The *.bm* file is called the bus map or a data path description. The second file, *demo.fsm* , is a high-level description of the FSM controller. The *.fsm* file is called the finite state machine file. The *spil* command also generates a source listing which contains a finite state control description. Examples of these files are shown in appendix A (SPIL Codec Files). The data path layout is produced by running the bus generator program; the command *busgen demo.bm* creates the

output file *demo_bm.cif*. The FSM layout is produced by running the finite state machine generation program called PLAmate [17]; the command *PLAmate demo.fsm* creates the output file *demo.cif*. The CIF layouts are for a 3μm CMOS technology [18,19].

Final connections of the FSM to the data path and address decoders, and of inputs and outputs to pad drivers are done manually using an interactive mask editor, *Caesar*, and a program called TARCON. TARCON (**T**erminal **AR**ray **CON**nector) will route the opposite sides of a rectangular channel provided that the channel that can be routed in one layer. This tool can save a great deal of time during manual routing. More importantly, it can also be used in a program which will automatically generate the interconnections between the data path and the control path.

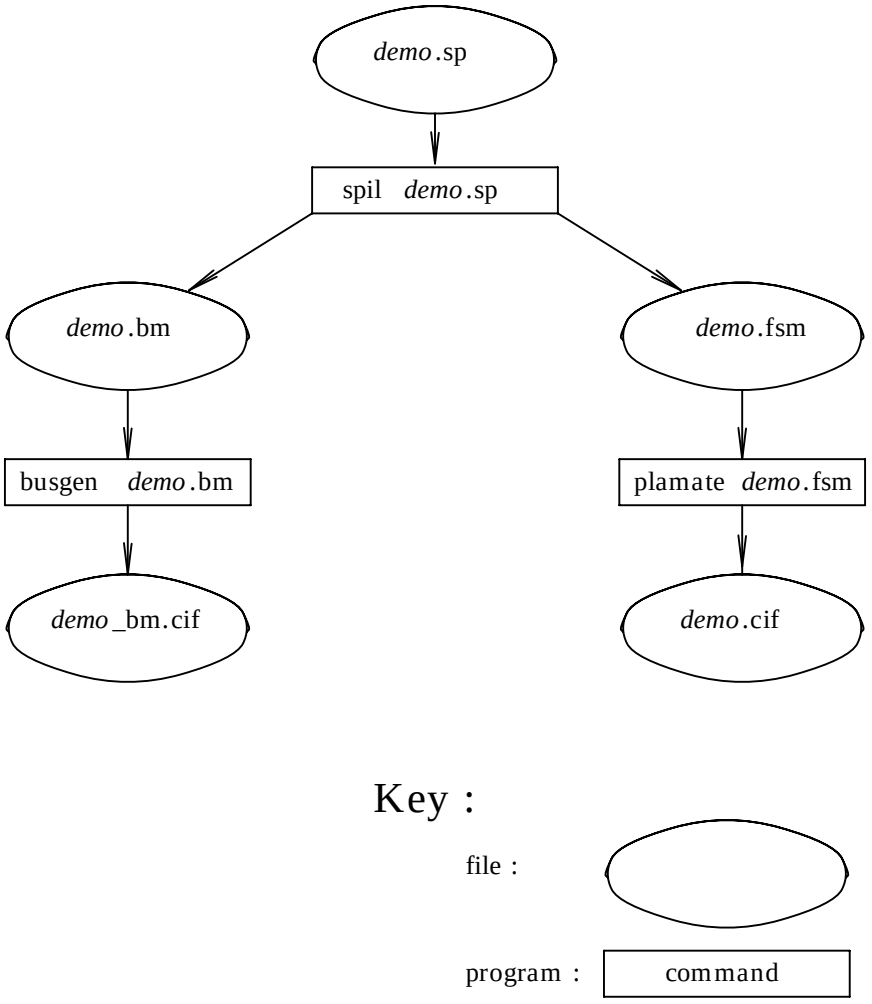


Figure 2.2. Compiling a SPIL Program

2.2.4. The Circuit Architecture of SPIL

A block diagram of the architecture is shown in Figure 2.3. The finite state machine provides the control flow for the algorithm and the data path performs the computations. The designer specifies a digital algorithm in SPIL's input language. After compiling the program, the designer's algorithm is permanently encoded in the finite state machine [1]. This encoding is achieved by having each state in the FSM controller represent one data transfer between two locations in the data path [20]. These two locations can be constants, storage registers or computational units, and they are specified by unique source and destination addresses. Thus an FSM state selects the two locations in the data path for the data transfer by generating a source and a destination address pair. The two addresses that the FSM generates are decoded by the address decoders in the data-path half of the architecture.

To illustrate this concept, suppose that in a SPIL program the designer has defined three variables A, B and C using the following syntax, which is identical to that of Standard Pascal:

```
var
  A, B, C : integer ;
```

These three variables correspond on a one-to-one basis with storage registers created for them in the data path. If the designer wanted to specify a calculation to add the contents of the B and C registers and store the result in the A

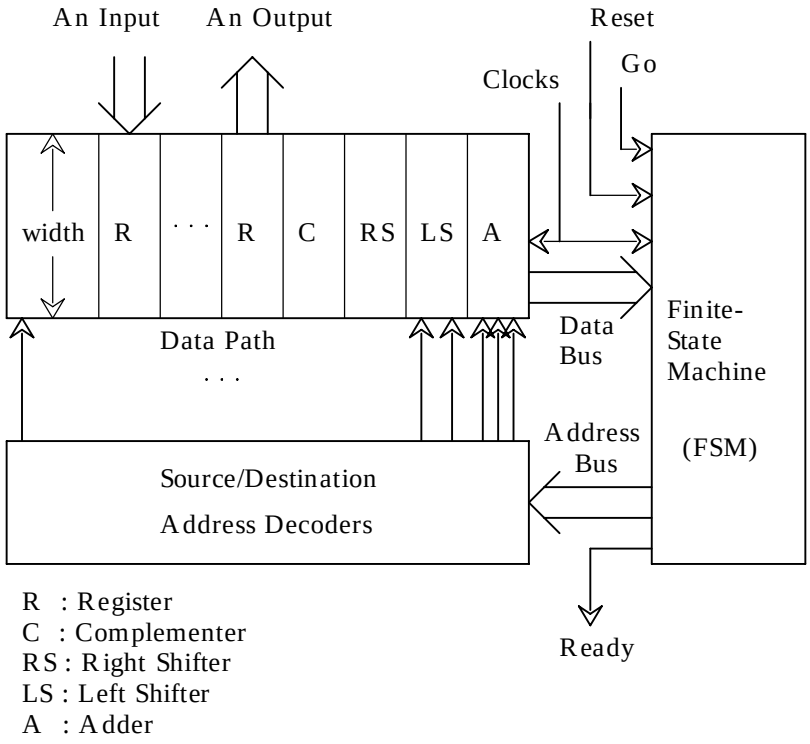


Figure 2.3. SPIL Circuit Architecture

register, the following program syntax would be used:

A := B + C ;

The architecture implements this high-level instruction by using three states of the FSM. During the first state, the source address will select the register B and the destination address will select the first input port of the adder computational unit. The second FSM state generates a source address for register C and a destination address for the second input port of the adder. Finally, the third state generates a source address for the output port of the adder and a

destination address for register A.

There are some further architectural features of SPIL worth noting. A minimum of four computational units are needed : addition, left shift by one bit, right shift by one bit with sign bit extension, and ones complement [20]. A ripple-carry adder was used. Another architectural feature is the data bus connection between the data path and some of the inputs to the FSM. This feedback permits the FSM to make conditional transfers to different states based on the result of some previous calculation and is used to implement high-level conditional tests in the designer's SPIL program. Note that this architecture does not have status bits or a condition code register. However, the designer could use an integer variable to store the result of a calculation for the purpose of conditional branching at a later time. There are also input port cells for connection of off-chip inputs to the data bus, and output registers to transfer data from the data path to off-chip outputs. The width of the data path (and hence the number of bits in the data bus) is variable and is specified in the SPIL program; the length of the data path is also variable and depends on the number of data registers, constants, computational units, chip inputs and chip outputs required by the algorithm. The width of each address bus is also variable and depends on the number of devices required in the data path.

As indicated, the controller is a finite state machine. PLAmate [17] is a module generator that generates **P**rogrammable **L**ogic **A**rray (PLA) and finite state machine layouts in Caltech Intermediate Form (CIF) for a 3μm CMOS technology. PLAmate accepts a high-level description of the PLA or FSM, including input and output lines and their ordering, assignment statements with Boolean expressions, FSM states, FSM state transitions, FSM outputs to be asserted in a given state, FSM outputs to be asserted during a particular state transition, and the FSM RESET state and RESET input. PLAmate then automatically reduces the Boolean expressions to minterm form, minimizes them and then produces the PLA/FSM layout, as well as a listing of the input/output numbering sequence and a PLA connection matrix.

The PLA or FSM generated by PLAmate is currently a static complementary CMOS NAND-NAND structure (Figure 2.4). The two levels of the PLA are the *AND* plane, which generates the minterms that are required, and the *OR* plane, which combines the minterms to produce the functions required. This particular structure (Figure 2.4) allows all NMOS devices to be placed in a common P well, reducing the silicon area required. The FSM automatically includes input and output latches and the feedback paths required to form the state variables. The use of a complementary CMOS structure reduces power dissipation, but results in a large silicon area because both PMOS and NMOS transistors are required for each logic function. In addition, the width of the series NMOS

transistors in the NAND gates is multiplied by the number of NAND gate inputs in order to maintain symmetric rising and falling propagation delays [17].

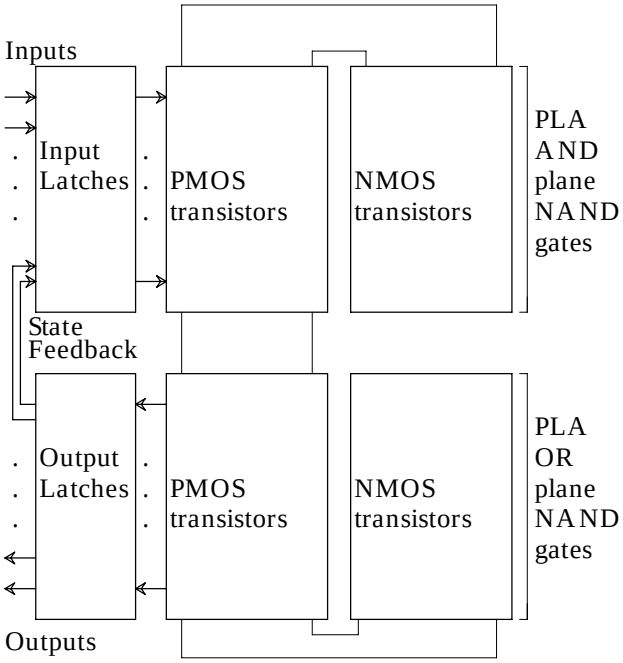


Figure 2.4. PLAmate Static CMOS FSM

Since the architecture has now been described, it is possible to explain why it was efficient to add the bit-mask tests such as `???0B` to the SPIL language. Essentially, the more *don't care* bits in the bit-mask, the fewer the number of connections from the data bus to the finite state machine (Figure 2.3). Reducing the number of these connections can significantly reduce the size of the PLA used in the FSM. Thus, using bit-mask tests can reduce the area of the design.

In summary, the SPIL architecture may be based on a simple register transfer design; however, it is very efficient to generate, and is adequate for low-performance chip designs.

2.2.5. Timing Considerations

2.2.5.1. External Signals

Three signals are used to control and monitor execution of the algorithm by the FSM. These are RESET, READY and GO (Figure 2.3). RESET and GO are inputs to the FSM. READY is an output from the FSM. When the RESET input is set *high* the FSM is forced into state 0. If the designer specified a `_reset` procedure, then state 0 will be its first state. This procedure will be followed by a *wait* state. If the designer did not specify a `_reset` procedure (Figure 2.1), state 0 will be the *wait* state. Figure 2.5 contains the FSM state diagram for the program in Figure 2.1. In the *wait* state, the READY output signal is set *high*. In this state, the FSM is waiting for a GO signal. When GO is set *high*, the FSM leaves the *wait* state which causes READY to be set *low*. This is followed by the FSM executing the main program of the SPIL algorithm. The main program is contained between the two statements `BEGIN` and `END` (Figure 2.1). When the FSM is finished executing the algorithm, it returns to the *wait* state where the READY output is again set *high* and the FSM waits for

another GO signal. This *READY-GO-execute* cycle is the method to perform repetitive calculations using the chip.

The FSM states (Figure 2.5) for the demonstration program illustrate the operation of the controller and the way that a designer's algorithm is coded in the FSM. State S0 is the *wait* state. In state S1 the chip input is moved to the register `temp` when the FSM sends the source address of `input` and the destination address of `temp` to the address decoders in the data path; also the FSM is loaded with the value of `temp` to be used in the conditional test in state S3. State S2 is a no-operation state; this is explained in the next section. In state S2 a source address of `temp` puts the register contents on the data bus for input to the FSM. The sign bit is tested and conditional branching occurs. If it is necessary to make temp positive because it is currently negative, this is done by moving `temp` to the complementer (S3), the constant 1 to the adder (S4), the output of the complementer to the adder (S5) and the output of the adder to `temp` (S6). This generates a two's complement using a one's complement plus addition of one. Finally, `temp` is moved to the output register (S7) and the FSM returns to the *wait* state and turns on the READY signal.

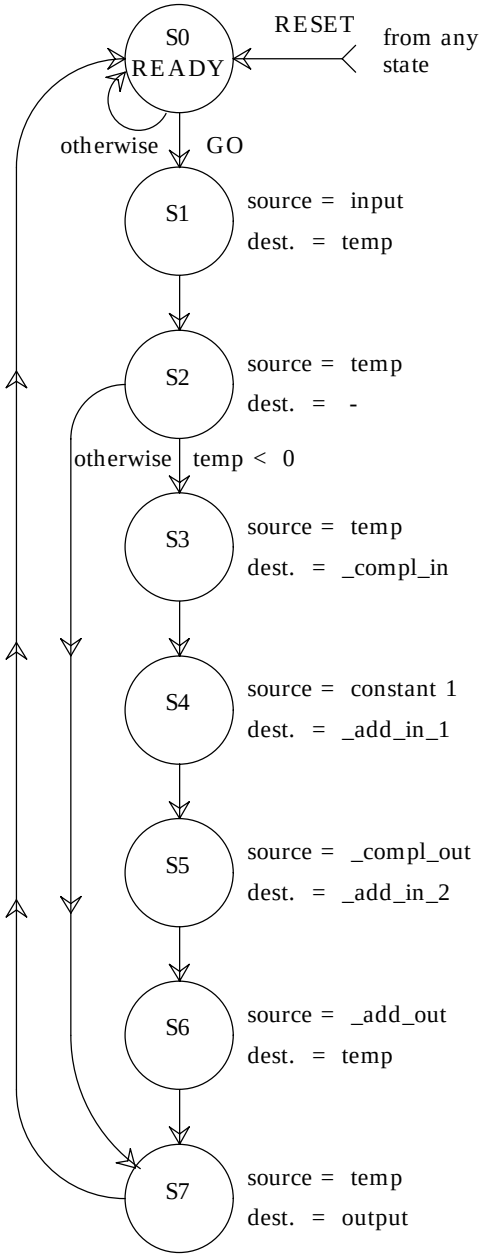


Figure 2.5. Sample Program FSM Controller

2.2.5.2. Critical Path Analysis

The remainder of this section on Timing Considerations contains a detailed analysis of the races which could occur in the SPIL architecture during its operation. This analysis is important because it is the basis for predicting the operating speed of the chip. The races will be specified in an expression which gives the minimum clock period. Since the architecture uses two non-overlapping clocks, there are four phases per clock cycle. The result of the critical path analysis is a specification of the minimum possible value for each of the clock phases.

Figure 2.6 shows all of the essential timing units of the architecture. The control path is represented by the PLA and the two sets of FSM latches. These FSM input and output latches are shown by the two blocks to the left of the PLA. The top latch, labelled *In*, is the FSM input latch, the bottom one, labelled *Out*, is the FSM output latch. These level-sensitive latches have two modes of operation, locked and transparent. When a latched is locked, its output contains the data that was previously applied to the input of the latch, at the instant when it was locked. When a latch is transparent, the input is simply passed to output. The FSM latches are clocked by PHI2. When PHI2 is low, the input latch is locked and the output latch is transparent. When PHI2 is high, the modes of the latches are reversed. The data path is represented by the

remaining parts of Figure 2.6.

The P-channel transistor at the top of the data bus represents the precharge circuitry. The data bus has storage registers, input/output registers and computational units connected to it. However, only the adder computational unit is shown in this diagram, since this is all that is needed to demonstrate timing. One destination latch and one source discharge port are shown. The adder's source and destination ports are shown because the adder is the slowest unit on the data bus. The source block has three signals connected to it, labelled α, β and γ. Signal α is connected to the data bus in order to conditionally discharge it. Signal β is the data which has been computed by the adder unit. When signal γ is set high, it enables the source to conditionally discharge the data bus based on the data input from signal β. The destination block has three signals connected to it, labelled δ, ε and ζ. Signal δ is used to read from the data bus. Signal ε is the output of the destination latch and is input to the computational unit. When signal ζ is set low it enables the destination to load the value from the data bus using signal δ. The gates below the registers are used to enable a particular source or destination unit when the address decoder selects that unit and the appropriate clock signal, PHI1- or PHI2, is applied. The address decoder is a combinational circuit which selects one source and one destination unit when the appropriate addresses have been generated by the PLA.

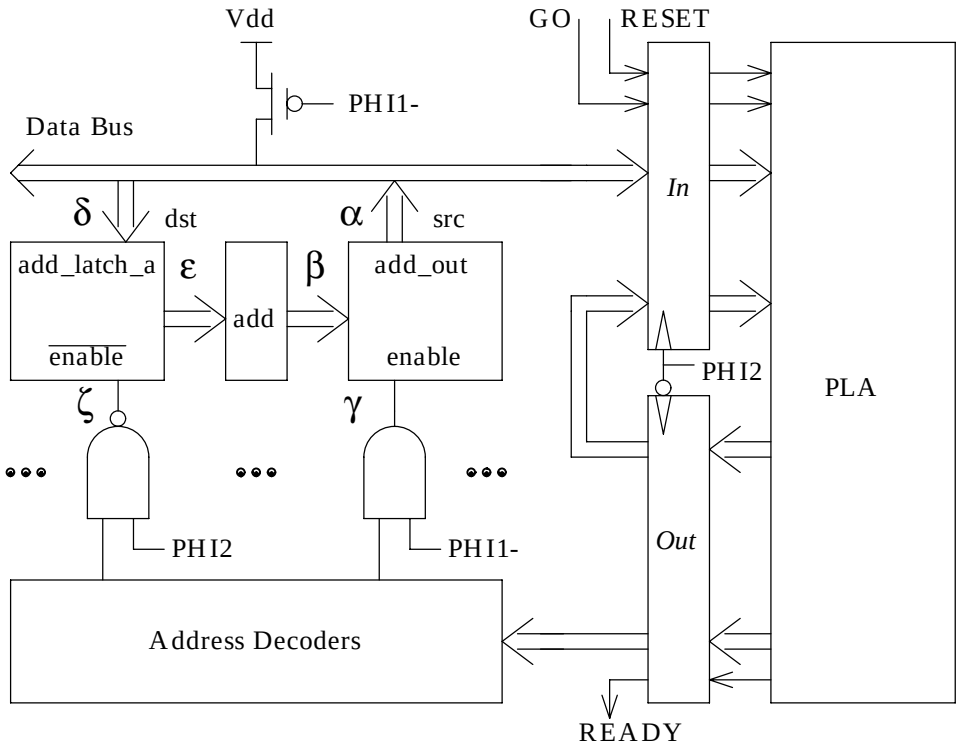


Figure 2.6. Timing Units of SPIL Architecture

The critical path will be described using Figures 2.7 to 2.10. These figures show, with bold lines, the signals which are propagating in each of the four phases. In each diagram, these boldly-drawn signals are the ones which have to settle before the next phase can be entered. Figure 2.11 shows the relationship between the four clock phase numbers and the clocks PHI1- and PHI2.

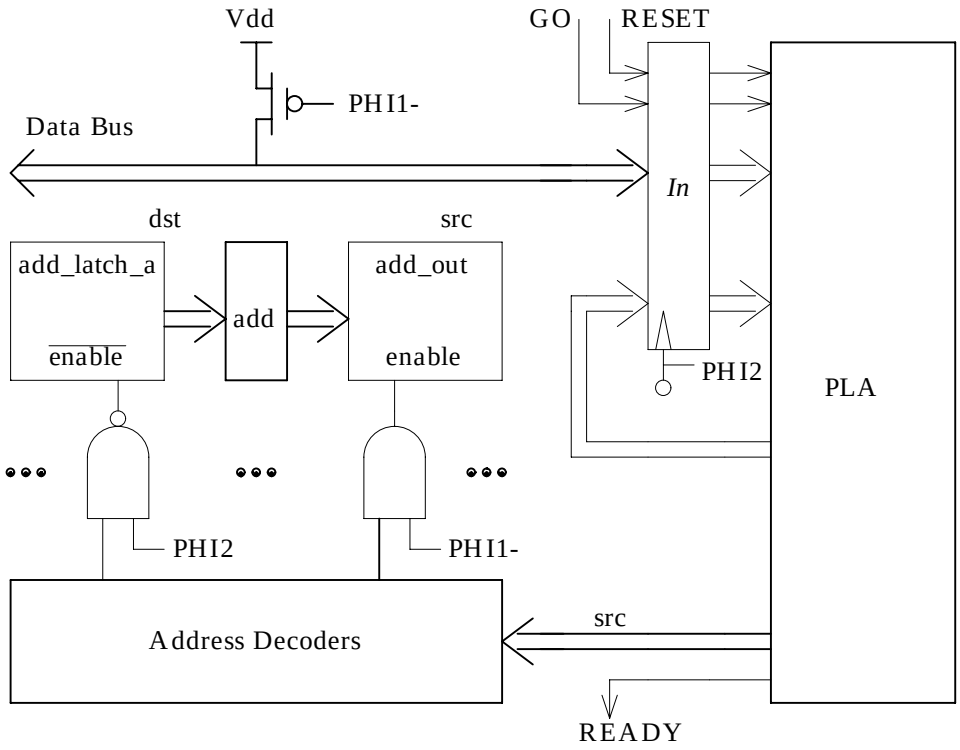


Figure 2.7. Phase 1 Timing, Precharge

In order to leave this phase and enter phase 2, three signals must have settled. First, the data bus must be precharged. Second, the data through the adder must have settled. And finally, the inputs to the AND gates which logically AND PHI1- and a source unit select line must have settled.

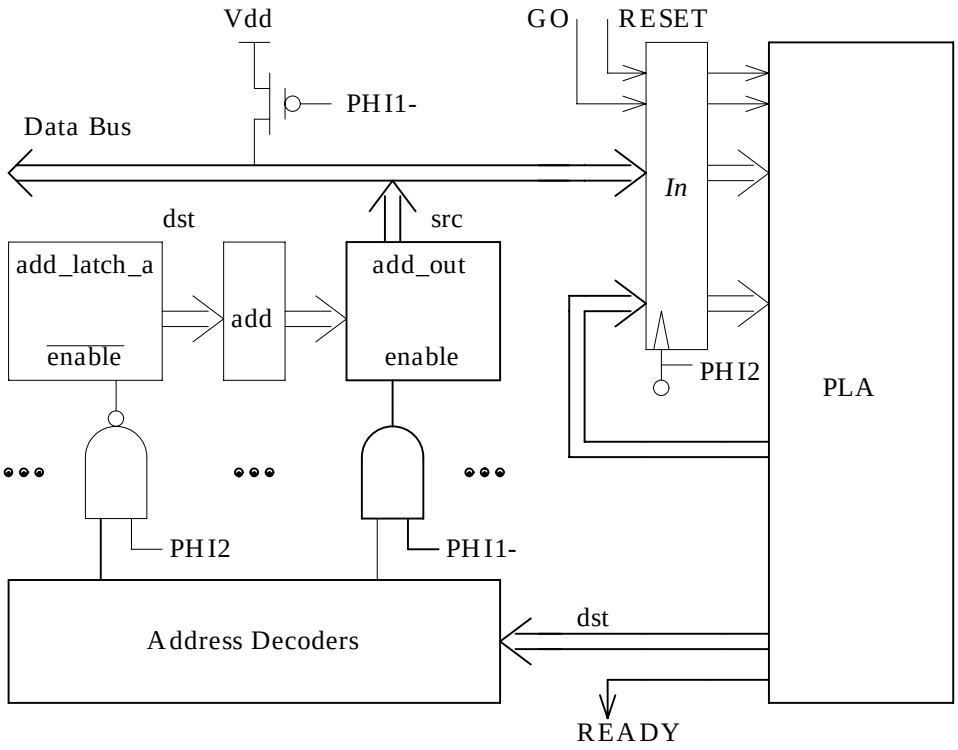


Figure 2.8. Phase 2 Timing, Source Discharge

In order to leave this phase and enter phase 3, three signals must have settled. First, the data bus must be conditionally discharged by the source unit. Second, all the signals through the output latches of the FSM must have settled; signals such as the READY signal drive output pads and could have much longer settling times than signals which go to the address decoders. And finally, the inputs to the NAND gates which logically NAND PHI2 and a destination unit select line must have settled.

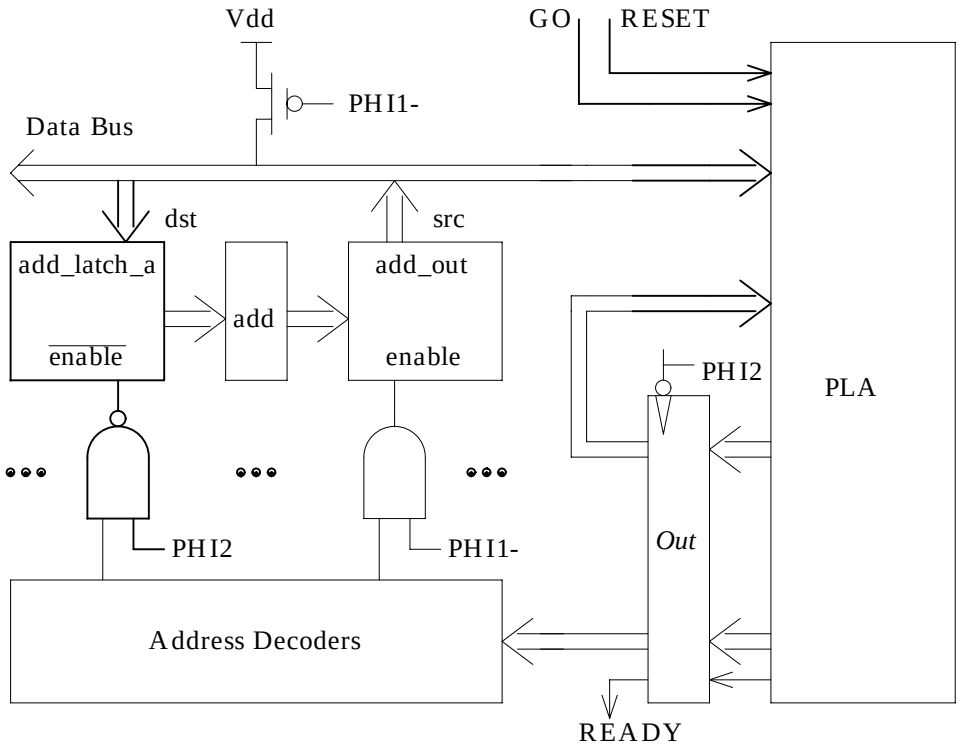


Figure 2.9. Phase 3 Timing, Destination Load

In order to leave this phase and enter phase 4, two signals must have settled. First, the data bus must be read into the destination register. Second, the data through the input latches of the FSM must have settled.

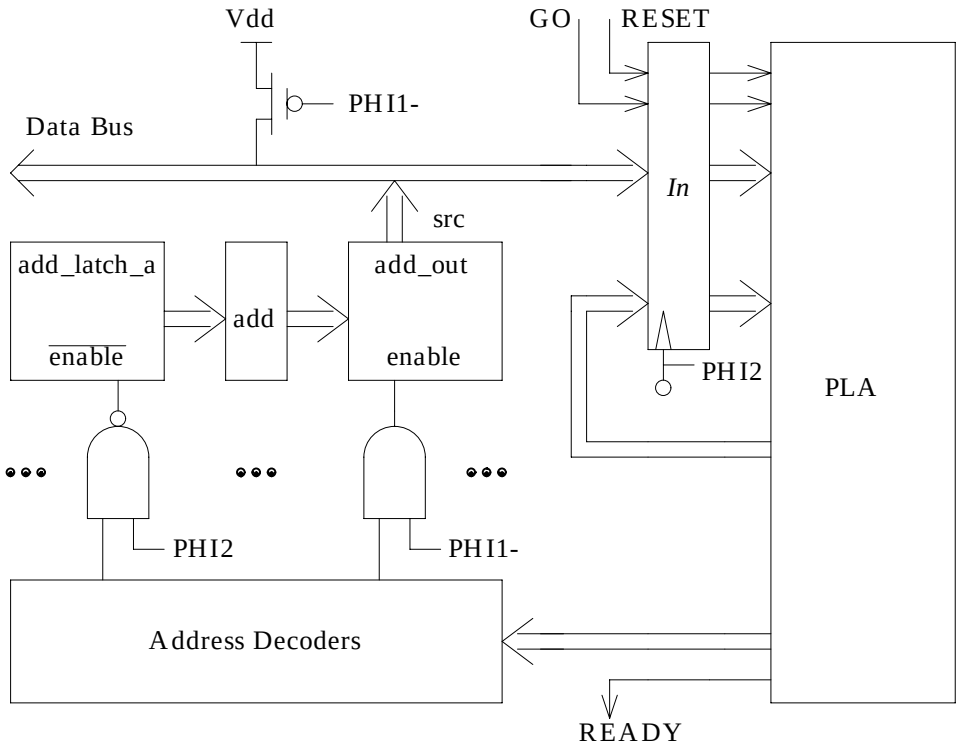


Figure 2.10. Phase 4 Timing, No Calculations

There are no signals propagating in phase 4 that prevent immediately entering phase 1. However, this is a theoretical restriction. In a real chip, there could be clock skew between phases. Thus, the PHI1- and PHI2 clocks which specify the phases may have to define phase 4 to be on the order of a few nanoseconds to avoid clock skew. This is discussed further in section 5.2 (Maximum Clock Frequency Determination).

Figure 2.11 contains a timing diagram which shows all of the signals which could possibly be part of the critical path. This timing diagram was obtained by analysing Figures 2.7 to 2.10. Here are definitions of the times in Figure 2.11.

T_{min} is the minimum possible clock period for PHI1- and PHI2.

t_1 is the minimum possible duration of phase 1.

t_2 is the minimum possible duration of phase 2.

t_3 is the minimum possible duration of phase 3.

t_4 is the minimum possible duration of phase 4.

t_{tol} is the time between the PHI1- and PHI2 clock edges which may prevent clock skew in a real chip in phase 4. The subscript *tol* means tolerance. This time will be on the order of nanoseconds. For the purposes of critical path calculations, this time will be set to zero.

t_p is the time to precharge the data bus.

t_{adder} is the time for data to propagate through the adder.

$t_{PLA,OL,src}$ is the time for the signals to propagate through the PLA, FSM output latches and address decoders to the input of all the source discharge units' AND gates which are gated by PHI1-. Note that only the inputs to these AND gates must have settled. It is not

necessary for all the outputs of the FSM latches to have settled.

t_c is the time to conditionally discharge the data bus by the slowest source unit.

$t_{PLA,OL}$ is the time for signals to propagate through the PLA to the FSM output latches.

$t_{PLA,OL,dst}$ is the time for signals to propagate through the PLA, FSM output latches and address decoders to the input of the destination units' NAND gates.

t_{dlat} is the time for a data bus value to be read into the slowest destination latch.

t_{IL} is the time for all signals to propagate to the output of the FSM input latches.

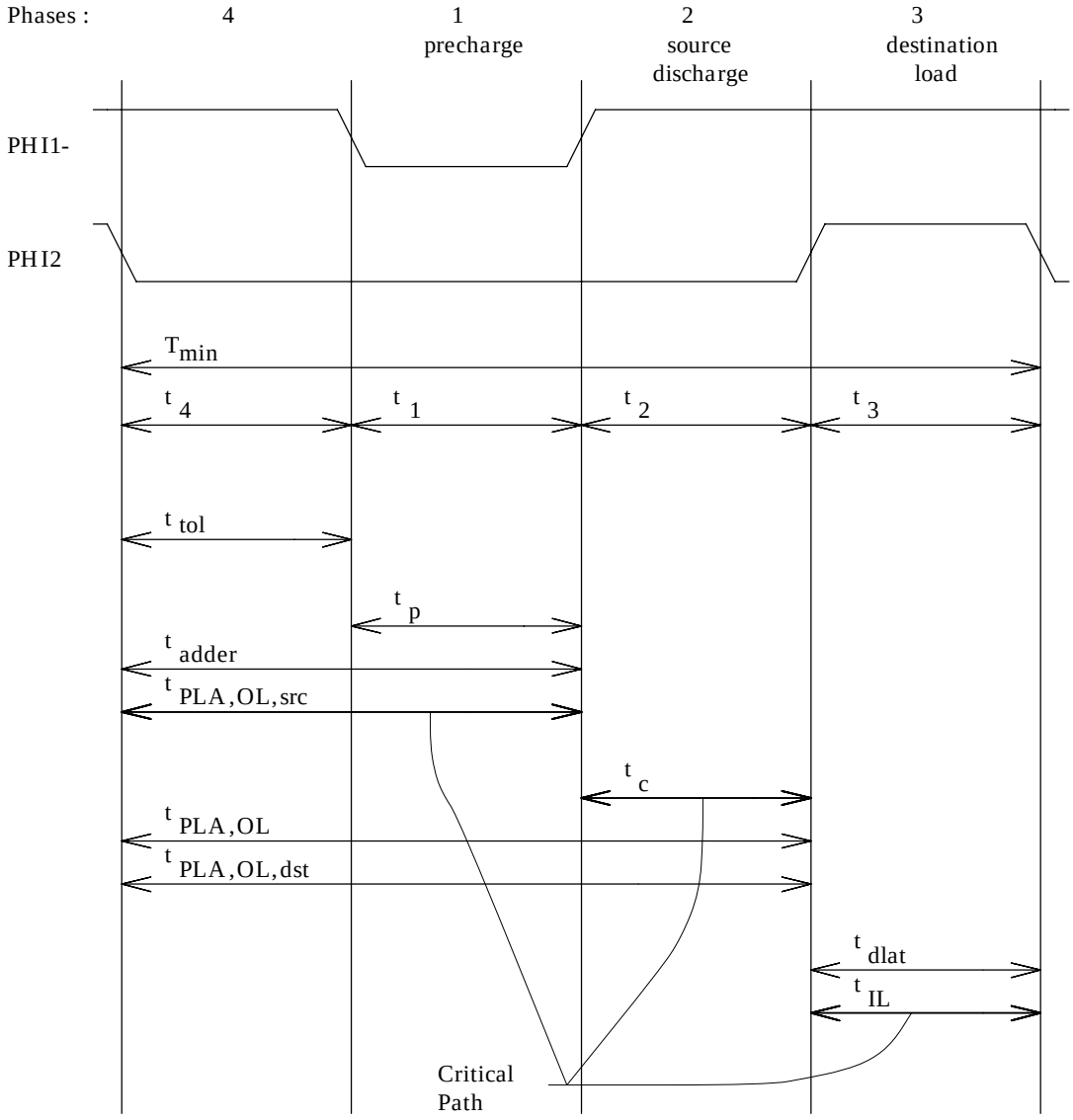


Figure 2.11. Critical Path Timing Diagram

The times shown in Figure 2.11 can be expressed as :

$$t_4 = t_{tol} \tag{2.1}$$

$$t_1 = \max (t_p , t_{adder} - t_4 , t_{PLA,OL,src} - t_4) \tag{2.2}$$

$$t_2 = \max (t_c , t_{PLA,OL,dst} - t_1 - t_4 , t_{PLA,OL} - t_1 - t_4) \tag{2.3}$$

$$t_3 = \max (t_{dlat} , t_{IL}) \tag{2.4}$$

$$T_{min} = t_1 + t_2 + t_3 + t_4 \tag{2.5}$$

$$f_{max} = \frac{1}{T_{min}} \tag{2.6}$$

Note that equations 2.1 to 2.5 will usually result in a simpler expression for T_{min}. From a detailed analysis in Section 4.3.2 (SILOS Critical Path Analysis), there is a most likely critical path, and it is shown in Figure 2.11. The significance of this is that the delay due to computational units in the data path does not contribute to the critical path delay. This is a result of having latches at both the inputs and the outputs of the PLA. These latches essentially create a two-stage pipeline. This covert concurrency using a pipeline increases the operating speed of the architecture and is very significant in a single-bus data path architecture.

The effect of the two stage pipeline can be observed in the FSM controller in Figure 2.5. In order to make a conditional branch, in state S2, the value to be tested must be put on the data bus the state before the FSM performs the test, state S1. The action performed by the source and destination addresses in state

S2, as shown in Figure 2.5, does not have any effect. Some architectures with pipelines solve the problem of starting calculations before a branch is taken by *flushing out* the pipeline before the branch instruction [21]. However, there is an approach to improve the speed of branch instructions, better-suited to this architecture. This approach involves moving calculation that occurs after the conditional branch block into the unused state which occurs just before the conditional branch. Naturally, this reordering of the calculations cannot change the final result of the algorithm. Only few calculations, if any, would satisfy this restriction. This kind of optimization is similar to the way that the RISC architecture and RISC compiler fill a pipeline before a branch with no-operation instructions (NOP) and then try to optimize the program by replacing the NOPs with other calculations that are independent of the branch [22]. An example of a situation where this optimization can occur is shown in appendix A (SPIL Codec Files), file *rx.spil_list*. The operations in state `S032` can be moved so that they are performed in state `S028`. This is an example of how to save one state. There are at least three states that can be deleted in this example. Currently, the SPIL compiler does not perform this kind of optimization with its branch instructions.

2.2.6. Design Trade Off Techniques

When a designer writes a SPIL program to describe a chip, there will be many trade offs required into order to obtain a design which satisfies desired design criteria. Methods of reducing the data path area, the finite state machine area, the number of external pins, the chip speed and the chip power dissipation are described below.

In order to reduce the size of the data path, the following techniques can be used. The number of registers, or program variables, that the algorithm requires should be reduced by reusing variables; however, this makes the algorithm more difficult to read. The number of constants that the algorithm uses should be reduced by combining other constants using addition and shifting, resulting in more states in the FSM. The number of input and output ports that the program uses can be reduced by combining similar ports (i.e. just input ports) into one port and then time-division multiplexing different data through that one port using a few additional output signals to control off-chip multiplexors; this also increases the number of states in the FSM. The number of computational units that a program uses should be reduced by trying to make use of the minimal number; this may increase the number of states in the FSM. An example of the use of the minimal number of computational units is a multiplication program which uses shift and add loops but tries to make use of only one shifter unit.

In order to reduce the size of the FSM, the following techniques can be used. Currently, SPIL does not perform data flow optimizations between separate high-level statements. This means that the designer can sometimes reduce the numbers of wasted states between high-level statements by expanding the high-level statements using lower level data transfers between the SPIL computational unit ports, such as `_add_in_1`, `_add_in_2` and `_add_out`; however, this makes the designer's program very difficult to read and thus more prone to programming errors. The size of the PLA can be reduced when the designer uses bit-mask tests in order to reduce the number of connections between data bus and the FSM. A very important technique to reduce the size of the FSM is to partition one FSM into a number of smaller FSMs. This is discussed further in the next chapter on SPIL Enhancement for DSP Design.

In order to conserve the number of external pins, the previously discussed techniques of time-division multiplexing and using bit-mask tests can be employed.

The operating speed of the chip can be improved by increasing the width-to-length ratios of transistors in the library of SPIL cells which are used to create the data path. A similar technique can be applied to the transistors in the latches of the FSM. The cost of these increases will be a greater power dissipation.

A chip's power dissipation can be reduced by decreasing the width-to-length ratios of transistors in the SPIL library for the data path and the FSM latches. The cost is a decrease in operating speed.

If it is not possible to trade off design criteria in order to achieve a desired performance, it may be necessary to consider if SPIL should be used for such a design or if the size of the algorithm should be reduced.

2.3. Performance Evaluation with Silicon Compilation

The previous section describes some of the trade offs among different aspects of a chip's performance in order to obtain a desired chip performance. This section discusses how performance measures are obtained and how they can be used to effect changes in the output of a silicon compiler.

In developing the design aid, the target silicon compiler was assumed to have the four levels shown in Figure 2.12. The behavioural description may consist of functional blocks and their interconnections (with a functional block such as an ALU specified in terms of inputs, outputs and the functions relating them). Logic synthesis then generates a logic level description, consisting of sets of Boolean equations. A given Boolean function can then be translated to the transistor circuit level, with a netlist specifying the connections. Through placement and routing of layout cells, the transistor circuit description is translated to

a layout level description.

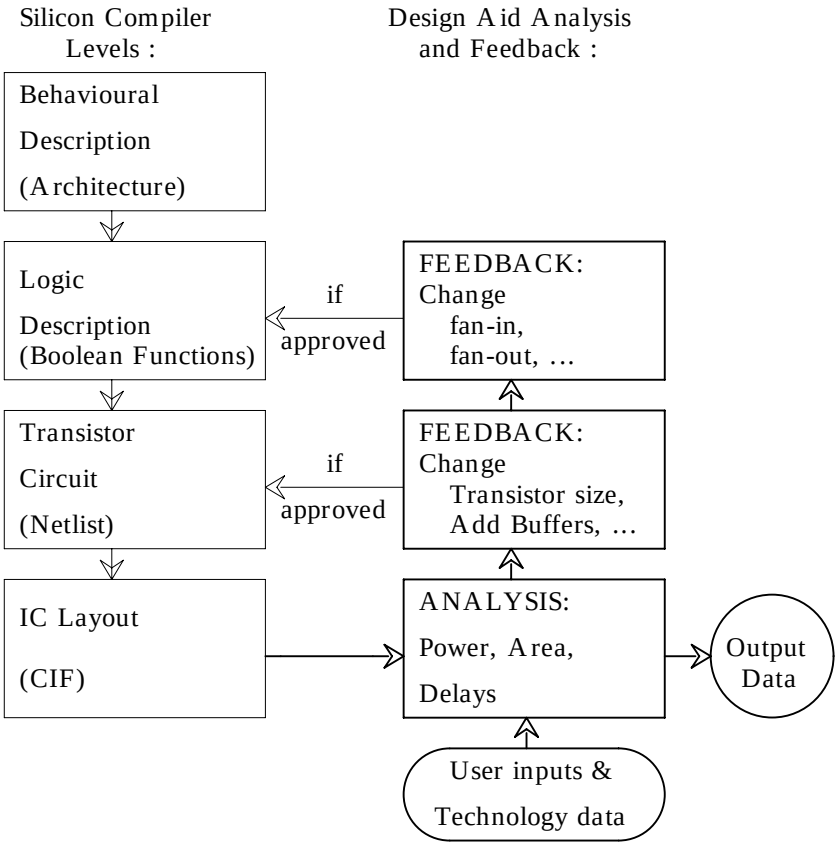


Figure 2.12. Design Aid Requirements

The analysis of the layout can be obtained using a performance evaluator which determines estimates of power dissipation, area and delay. The objective is to provide an analysis of the output of a silicon compiler and also provide feedback to higher levels in the silicon compiler. Currently in SPIL, all feedback is performed manually by the designer.

The logic level decisions on the definition of Boolean functions affect the number of transistors in one gate and the fan-in and fan-out, and changes could be recommended to improve performance. Transistor width and length parameters, plus the addition of buffers, also affect the performance measures, and changes could be identified. Any feedback changes would require designer approval before implementation.

CHAPTER 3

SPIL Enhancement for DSP Chip Design

In order to provide a complete design environment with SPIL, four CAD tools were used: (EPAD, SILOS, Partition and TARCON). These tools helped to generate and to analyse the final codec chip that was generated.

EPAD is a performance evaluator which analysed the codec chip layout from SPIL to obtain estimates about the codec's performance. SILOS is a logic simulator. The SILOS circuit description file of the codec was generated automatically from EPAD to allow the EPAD delay estimates to be used in a logic simulation. The program called Partition can be used to partition an FSM which is generated by SPIL, into two or more FSMs that have a combined area less than the original. The program called TARCON can be used to interconnect signals in the layout during preparation for chip fabrication.

3.1. EPAD

3.1.1. Incorporating EPAD into SPIL

A design aid called EPAD (**E**stimation of **P**ower **A**rea and **D**elay) has been developed for performance estimation of silicon area, power dissipation and propagation delays of CMOS VLSI circuits. EPAD automatically extracts estimates of performance measures from a description of the integrated circuit mask features and has been tailored for use with SPIL. EPAD was designed to provide feedback in the SPIL design cycle (Figure 2.12) Currently, EPAD performs only an analysis of the layout; feedback is performed by the designer.

3.1.2. Overview

Two input files are required to run EPAD (Figure 3.1). One file provides a CIF description of the layout, and the other provides a set of technology and input parameters. Performance measure estimates are written into the *.epad* file and designer parameters and errors are listed in the *.log* file. EPAD provides an input, in the *.dat* file, to the logic and switch level simulator SILOS [23]. In order to calibrate EPAD, the critical path simulations obtained from SILOS were compared to test results of fabricated chips; this is described in chapter 5 (Test Results and Suggested Enhancements). EPAD is written in *awk*, a pattern

scanning and processing language [24].

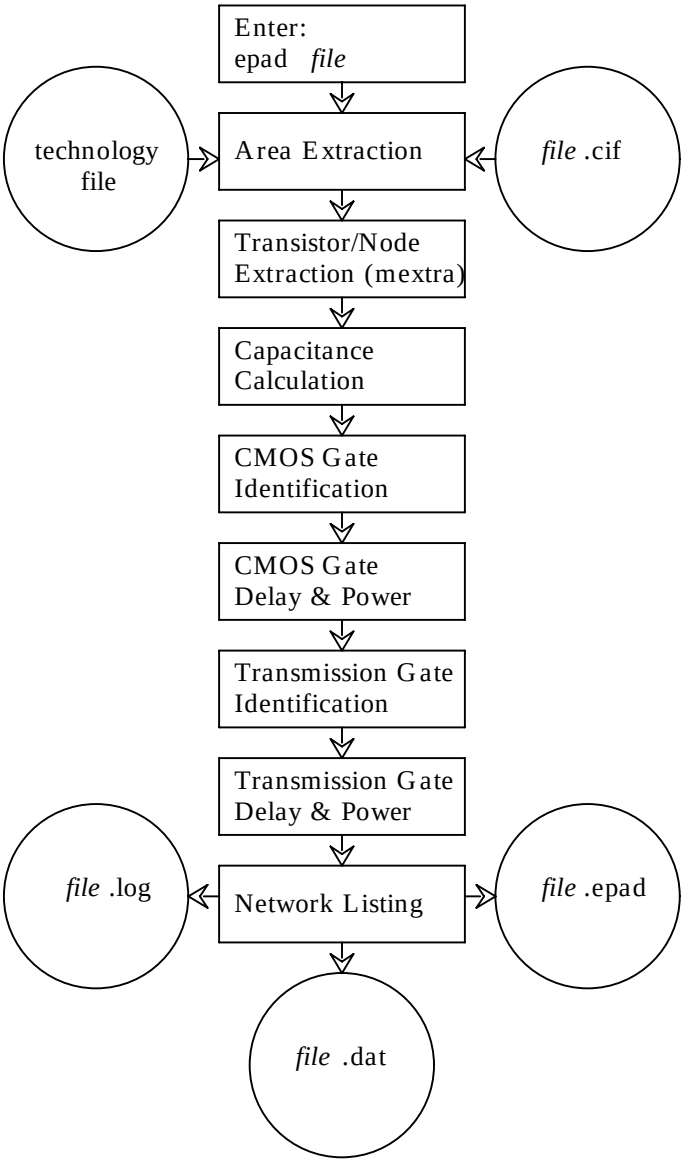


Figure 3.1. Overview of EPAD

3.1.3. Performance Measures

The three performance measures which EPAD extracts from layouts are power dissipation, layout area and propagation delay.

EPAD predicts the dynamic power dissipation associated with gate output capacitances charging and discharging. For every gate in the layout, EPAD applies the formula:

$$P_{dynamic} = C_{gate} V_{DD}^2 f \quad (3.1)$$

where:

$P_{dynamic}$ is the dynamic power dissipated by the gate charging and discharging is output node capacitance.

C_{gate} is the capacitance of the output node of the gate.

V_{DD} is the voltage swing of the gate's output node.

f is the switching frequency of the gate.

The areas predicted by EPAD are the area of minimal size bounding boxes around each of the cells in the CIF layout.

The delays predicted by EPAD are on a per gate basis. For every gate in the layout, EPAD lists the propagation delay times of the gate for the two cases when the output node is rising or falling. These propagation delay time

definitions are shown in Figure 3.2.

Figure 2. Propagation Delay Time Definitions

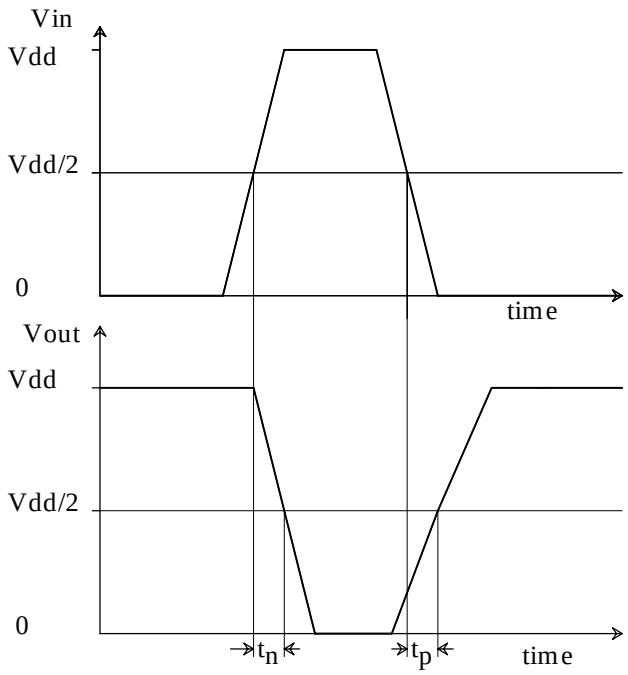


Figure 3.2. Propagation Delay Time Definitions

3.1.4. Delay Models

B.A.White and M.I.Elmasry considered several propagation delay models for CMOS inverters [25] and compared them using the circuit simulator SPICE [26]; they concluded that the Burns inverter delay model [27] gave the best results. To determine the propagation delay estimates for static NAND and NOR gates, it was necessary to transform the series and parallel transistor

groups in these types of gates into an equivalent CMOS inverter. For example, the series N-channel transistors in a NAND gate would be transformed into one N-channel transistor by determining a width-to-length ratio derived from the resistances of all the *on* transistors.

Equations 3.2 and 3.3 are used to combine the width-to-length ratios of parallel and series transistors.

$$\left[\frac{W}{L}\right]_{parallel} = \min\left[\left[\frac{W}{L}\right]_1, \left[\frac{W}{L}\right]_2\right] \tag{3.2}$$

$$\frac{1}{\left[\frac{W}{L}\right]_{series}} = \frac{1}{\left[\frac{W}{L}\right]_1} + \frac{1}{\left[\frac{W}{L}\right]_2} \tag{3.3}$$

The parameter *kcapseries* will be used to calibrate EPAD. It is a designer input which estimates how much of the capacitance of the nodes between the series transistors in a gate will be added to the gate output capacitance during the delay estimation. As *kcapseries* approaches 1, the gate output capacitance increases, resulting in longer estimated propagation delays. Thus, *kcapseries* can be used to obtain bounds on gate delays due to the worst and best case capacitive loading conditions. EPAD predications can be compared to the results of testing in order to calibrate *kcapseries*.

3.1.5. EPAD Example

The following is a summary of an example used to verify EPAD during development [16,25] and is included here because it explains relevant details of EPAD. The circuit of Figure 3.3 was used to compare EPAD delay estimation to SPICE. The circuit is a 44-transistor design of a four-to-one multiplexor made from three two-to-one multiplexors linked hierarchically. A summary is shown in Table 3.1. EPAD-0 gave estimates within 8% of SPICE, but requiring only 1% of the CPU time. Furthermore, the SPICE results are bounded by the EPAD-0 and EPAD-1 calculations.

When a SILOS simulation was performed on this example, using a *.dat* file generated from an EPAD-1 run and using similar worst-case conditions that existed in the SPICE simulation, the propagation delays were as follows. The propagation delay (rising input) from *s_0* to the node *end* was 38 ns, and the delay for falling input was 32 ns. These SILOS times compare favourably with the summations of EPAD-1 times of 39.55 ns and 32.77 ns (Table 3.1). Differences can be accounted by the fact that SILOS uses only integer values.

The EPAD power dissipation estimates were compared to a SPICE calculation of power dissipation, using a method proposed by Kang [28]. This involved adding a dependent current source and parallel RC circuit to the SPICE netlist in series with the V_{DD} line so that the voltage across the capacitor indicates the

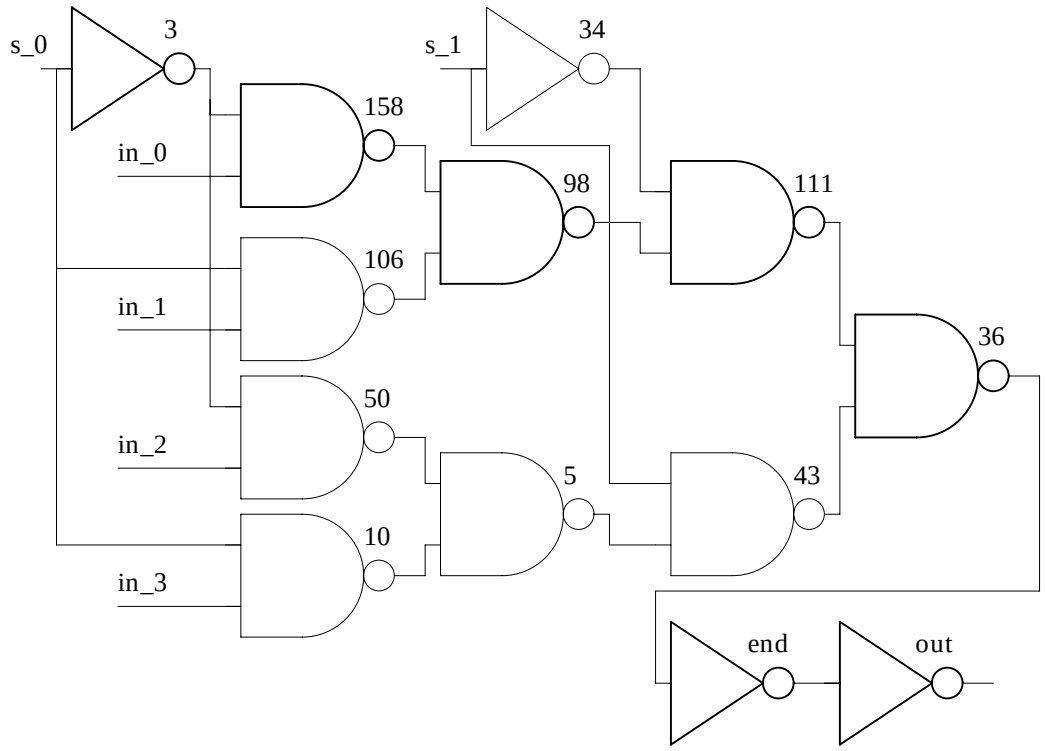


Figure 3.3. An EPAD Evaluation Circuit

average power consumption, without disturbing circuit operation. The SPICE simulation showed a power dissipation of 21.17 micro-watts at 1 MHz. The EPAD power dissipation estimates were 42.17 micro-watts (EPAD-0) and 52.38 micro-watts (EPAD-1). As expected, the EPAD estimates were higher, due to the assumption that every gate is switching in every clock cycle.

Table 3.1. Comparison of EPAD to SPICE

Input Node	Gate Type	Output Node	50%-Point Propagation Delays (ns)					
			Rising s_0			Falling s_0		
			SPICE	EPAD-0	EPAD-1	SPICE	EPAD-0	EPAD-1
s_0	.INV	3	3.10	3.48	3.48	4.34	6.05	6.05
3	.NAND	158	7.05	6.83	8.27	4.63	4.36	5.29
158	.NAND	98	6.10	5.41	8.42	5.08	4.70	7.32
98	.NAND	111	7.09	6.83	8.27	4.63	4.36	5.29
111	.NAND	36	5.83	5.21	8.22	4.96	4.52	7.15
36	.INV	end	4.06	2.89	2.89	2.83	1.67	1.67
s_0	total	end	33.23	30.65	39.55	26.47	25.66	32.77

EPAD-0 : *kcapseries* = 0.0
EPAD-1 : *kcapseries* = 1.0

3.2. SILOS

SILOS was chosen as the logic simulator to interface with EPAD because of four important features.

The first feature is that the logic simulation capabilities of SILOS properly simulate exclusive or gates modelled using transistors, as shown in Figure 3.4. In less robust simulators, the feedback through transmission gates is not correctly modelled as a combinational circuit. The ripple-carry adders used in the SPIL architecture are based upon circuits similar to the exclusive or gate.

The second feature of SILOS is its ability to model high-resistive transmission gates, as well as low-resistive transmission gates. This is important to the SPIL architecture because the source discharge units must discharge the data bus against the pull of of the *trickle-charging* data-bus precharge circuit.

The third feature of SILOS is its ability to perform fault simulation, confirming the quality of the circuit test pattern.

The fourth important feature of SILOS is its speed of execution. When EPAD delay models were used in the logic simulator, it gave the accuracy approaching that of a timing simulation, but requiring less CPU time.

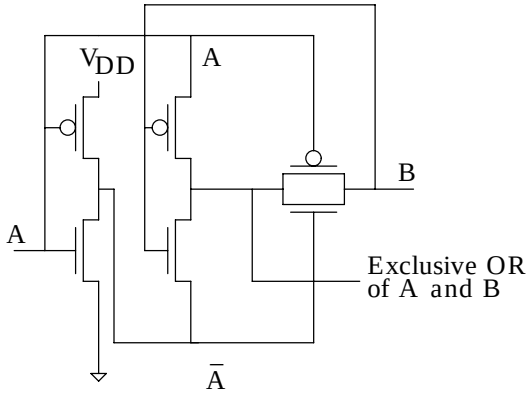


Figure 3.4. Exclusive OR Gate Circuit

3.3. FSM Partitioning

The silicon area required by the FSM generated by PLAmate was observed to increase rapidly as a function of the number of states in the algorithm, and the number of product terms required, primarily due to the scaling of NMOS transistor widths with the number of NAND gate inputs. One alternative for reducing the FSM area (while still using PLAmate) is to partition the state

diagram into separate sequential parts. The separate parts are each used to generate finite state machines that are combined into one controller. The FSMs run in sequence; the completion of the first one activates the second one, and so on. The splitting can be done by modifying the *.fsm* PLAmate input file created by SPIL; therefore no changes to SPIL are required.

Partitioning was tried on the receiver's FSM controller. For the case of two partitions, states 0 to 16 were used for the first FSM, with an added output to generate the GO signal for input to the second FSM. New states were created to assert this GO and then return to state 1. The second FSM was composed of states 17 to 32. A new state 0 was added to assert the READY signal from the second FSM and wait for the GO signal from the first FSM. State 32 was modified to return to this new state 0 which is also the RESET state.

Partitioning was achieved automatically by running a program, *Partition* (written by Brian White), on the *.fsm* file generated by SPIL to produce new *.fsm* files for the partitions. To run the program on the receiver, enter:

partition 3 17 32 receiver

where:

3 is the number of the state that waits for the GO signal.

17 is the starting state for the second partition.

32 is the last state.

The partitions must be self-contained with a sequential state advancement at the partition point (in this case, state 16 goes to 17). Also note that state 17 is the beginning of a two-state conditional test block. It is not possible to start a partition in the middle of a two-state conditional block such as states 17 and 18, see appendix A (SPIL Codec Files), file *rx.spil_list*. The current version of the partitioning program does not indicate if the designer has made such an invalid partition. New *.fsm* files *receiver_1.fsm* and *receiver_2.fsm* are produced. The first FSM has an added output GO_2 to give a GO signal at the appropriate time to the second FSM. Appropriate reset and starting states were added to the second FSM, and appropriate ending states added to the first FSM. This is set up in such a way that the address line outputs can be ORed together, the overall GO signal is connected to only the first FSM, an overall READY signal is obtained by ANDing all of the individual FSM ready signals, and the overall RESET signal is connected to each FSM reset. The data path is connected to the input latches of each FSM.

The program handles up to 10 partitions; in these cases the first FSM has an output GO_2 to be connected to the GO input of the second FSM, the second FSM has an output GO_3 to be connected to the third FSM, and so on.

The results of partitioning are shown in Table 3.2. There is some reduction in the area in utilizing to two partitions. With this capability of partitioning, the algorithm could be rewritten to provide for partition points. This analysis does not include the area of the wiring and combinational circuits required to interconnect the partitions. The area saved by partitioning was not enough to warrant the extra design effort required.

Table 3.2. FSM Partitioning Results

Example	Number of Inputs	Number of Outputs	Number of States	Number of State Lines	Number of Product Terms	Height (*dsm**)	Width (*dsm*)	Area $(dsm)^2$
1. no partition								
receiver.fsm	10	9	33	6	38	2433	3137	7,632,321
2. two partitions								
receiver_1.fsm	10	10	19	5	22	1534	1942	2,979,028
receiver_2.fsm	10	9	17	6	20	1524	1935	2,948,940
Total Area								5,927,968

* Design Scale Microns [18,19] (1 *dsm* = 0.6 μm)

3.4. TARCON

TARCON (**T**erminal **AR**ray **CON**nector) was written by Dan Salomon. It is a small program that can be used to route between the opposite sides of a channel, provided that the channel can be routed in one layer, such as metal. This program will be the basis for automatically interconnecting the SPIL data path and FSM. Currently, TARCON can be run by manually entering the coordinates to be routed. Even used manually, TARCON is more efficient than Caesar [29]. TARCON can be instructed to route the channel in any width. If the channel cannot be routed in the specified width, the minimum possible width will be used.

CHAPTER 4

ADM-PCM Codec Chip Using SPIL

4.1. Chip Specifications

In order to evaluate SPIL and its support tools, such as EPAD, an appropriate design example had to be chosen. This example demonstrated that the design methodology of using SPIL and EPAD increases design automation.

For the design example to be effective, it had to satisfy certain requirements, such as power dissipation, area and delay. These requirements or constraints are imposed by the algorithm and the packaging of the fabricated chips. The algorithm will specify that the design will have to operate at a certain speed, and thus, delays in the design will have to be minimized to satisfy the speed requirement. The type of packaging for the chips obtained from the CMC imposes limits on the chip area, power dissipation and number of external pins [18,19].

Since the chips generated by SPIL have maximum clock frequencies on the order of MHz, the maximum data rates will on the order of kHz. SPIL chips can have approximately a dozen storage registers of up to about 22 bits wide.

The specific constraints imposed by the packaging are as follows. The ceramic **D**ual-**I**n-**L**ine (DIP) packages can reliably dissipate 750 mW. The area of a chip cannot exceed 4511µm x 4511µm. However, the design may use more than one chip. If two chips are required, the data path could go on one chip and the FSM could go on the other. The maximum number of pins for a ceramic DIP package is 40. The maximum number of pins for a **P**in-**G**rid-**A**rray (PGA) package is 68. The maximum power dissipation for the PGA is also 750 mW.

Based on the limitations, a design example was chosen for speech processing, specifically for speech compression. Since SPIL produces a digital signal processing architecture, the example had to perform an *all-digital* compression. The design example is a chip which performs conversions between high-speed bit-serial **A**daptive **D**elta **M**odulation (ADM) and low-speed bit-parallel **P**ulse **C**ode **M**odulation (PCM). The Song step-size predication algorithm was used [15]. Since hardware multiplications in SPIL have to be done using shift-and-add-loops, the cost of multiplications is very high. This means that more sophisticated techniques, such as **L**inear **P**redictive **C**oding [30], could not be implemented because they require multiplications. However, the Song algorithm, can still compress 8-bit 8kHz PCM to and from 32kHz ADM and maintain a **S**ignal-to-**N**oise **R**atio (SNR) of 25dB [15].

4.2. Algorithm Design

The chip is divided into two main parts. One part is called the transmitter which converts an 8-bit PCM signal to an ADM signal. The other part is the receiver which performs the reverse operation and converts an ADM signal to an 8-bit PCM signal. Eight bit PCM was chosen because it is a standard value and it provides a high enough SNR, compared to the theoretical restrictions of the Song algorithm. Consider that the SNR of 8-bit PCM is about 8 bits x ~6dB/bit = 48 dB. This is higher than the maximum theoretical SNR of the Song algorithm [15]. The transmitter and the receiver were designed separately using SPIL and then combined into one chip using a layout editor, *Caesar*. The transmitter and receiver operate independently. This means that with two chips located at opposite ends of a channel that full-duplex communication can occur because there are independent transmitter-receiver pairs at opposite ends of the channel.

It is necessary to describe the receiver before the transmitter since the transmitter is built upon the same circuitry present in the receiver. However, looking ahead at the block diagram of the transmitter (Figure 4.5) may make understanding the receiver easier.

A block diagram of the Song ADM-to-PCM conversion algorithm [15] used in the receiver is shown in Figure 4.1. At the left is the bit-serial ADM input. At the right is the bit-parallel PCM output. The block that the ADM bit stream passes into is known as the step-size predictor. It tries to predict the the general trend of the ADM input and adjusts the step size accordingly. The second block in Figure 4.1 is for the summation of the step sizes with the previous estimates of PCM values. Finally, the AND gate at the end of the diagram represents a simple way of obtaining a PCM output at a lower bit-parallel data rate by using only every N^{th} estimate from the summation block. Note that this simple method of sampling to convert the PCM output down to the lower bit rate results in about a 7dB loss in SNR compared to digitally filtering the high-speed PCM before sampling [15]. The loss in SNR is due to aliasing of the noise that the Song algorithm creates in the conversion process. It was impractical to add such a digital filter to the algorithm due to the multiplications that are required. However, filtering could be done as a post-processing operation on the output of the SPIL chip [31,32].

Equations 4.1 to 4.4 mathematically describe the details of the digital signal processing algorithm.

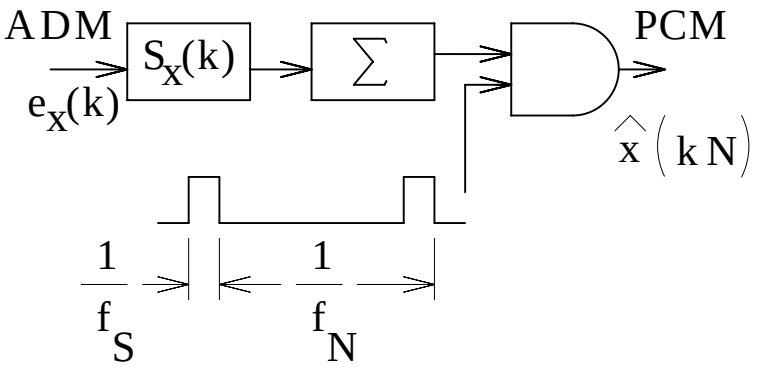


Figure 4.1. Receiver : ADM-to-PCM Converter

$$e_x(k) = +1, -1 \text{ represented in binary as } 1,\ 0 \tag{4.1}$$

$$S_x(k) = \left|S_x(k-1)\right| e_x(k-1) + S_{min}\ e_x(k-2) \tag{4.2}$$

$$\hat{x}(k) = \hat{x}(k-1) + S_x(k) \tag{4.3}$$

$$f_S = N\ f_N \tag{4.4}$$

where:

$e_x(k)$ represents the error between the true PCM value and the estimate at time step k. If $e_x(k) = 1$ then the true PCM value is greater than or equal to the estimate. If $e_x(k) = -1$ then the true PCM value is less than the estimate.

$S_x(k)$ represents the value of the step size at time step k. In other words, the estimate of the true PCM value at time step k has been increased by the value of $S_x(k)$.

S_{min} represents the smallest possible change that can be made to the step size. A value of $S_{min}= 1$ has been used.

$\hat{x}(k)$ represents the estimate of the true PCM value at time step k.

f_S represents the sampling frequency of the ADM values.

f_N represents the sampling frequency of the PCM estimates.

N is an integer greater than or equal to one. N is 1 in the implemented version. This is the most general implementation because it permits any interface to the chip to arbitrarily choose N.

Equation 4.1 specifies the ADM values and how they are represented in binary.

Equation 4.2 specifies how the next step size, $S_x(k)$, is calculated based on the trend of the ADM inputs, $e_x(k-1)$ and $e_x(k-2)$, over the previous two time steps. The predictor equation (4.2) changes the step size, $S_x(k)$, to satisfy two criteria. The first criteria is to adjust the step size's sign so as to move toward the true PCM value. The second criteria is to adjust the step size's magnitude to decrease or increase if the predicted PCM value appears to be converging or not converging on the true PCM value. In order to explain these two criteria of the predictor, the details of its operation are shown in Table 4.1. Table 4.1 shows

all the combinations of negative and positive values that can exist in equation (4.2). As an example, consider line three of the Table 4.1. The situation described by line three is shown in Figure 4.2. Since $e_x(k-1)$ is + 1 in Table 4.1, line 3, the true PCM value was greater than the predicted one at time k-1. The situation was reversed at time step k-2. From $e_x(k-1)$ which is positive, equation (4.2) causes the step size, $S_x(k)$, to be positive so that the predicted PCM value moves towards the true PCM value. Since the predicted PCM value's curve crossed just over the true PCM value's curve, the predicted PCM value must be converging on the true PCM value. Thus, the step size's magnitude is decreased to X-1 from X, and thus, the second criteria of the predictor is met.

Table 4.1. The Song Predictor (Equation 4.2) (X> 0) (S_{min}= 1)

$S_x(k)$	$\lvert S_x(k-1) \rvert$	$S_x(k-1)$	$e_x(k-1)$	$e_x(k-2)$
-X-1	X	-X	-1	-1
-X+ 1	X	-X	-1	+ 1
+ X-1	X	-X	+ 1	-1
+ X+ 1	X	-X	+ 1	+ 1
-X-1	X	+ X	-1	-1
-X+ 1	X	+ X	-1	+ 1
+ X-1	X	+ X	+ 1	-1
+ X+ 1	X	+ X	+ 1	+ 1

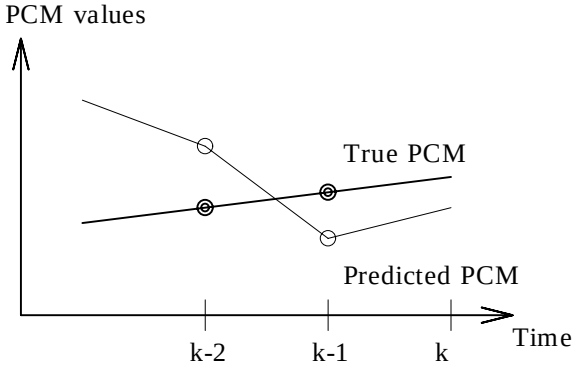


Figure 4.2. Situation in Table 4.1, line three.

Equation 4.3 specifies how the step size, $S_x(k)$, is added to the previous estimate of the PCM value, $\hat{x}(k-1)$, to produce the current estimate of the PCM value, $\hat{x}(k)$.

Equation 4.4 specifies that the sampling frequency of the ADM inputs must be N times greater than the sampling frequency of the estimates of the PCM at the output. As indicated, N is equal to 1 for generality.

Figure 4.3 is a signal-flow graph representation of equations 4.1 to 4.4, as well as an expanded version of the block diagram shown in Figure 4.1.

In this section the SPIL program which generates the receiver is described in detail. The actual program follows this description as shown in Figure 4.4. In order to help with the explanation, references will be made to Figure 4.3 because the program implements, step by step, what is shown in the signal flow graph in

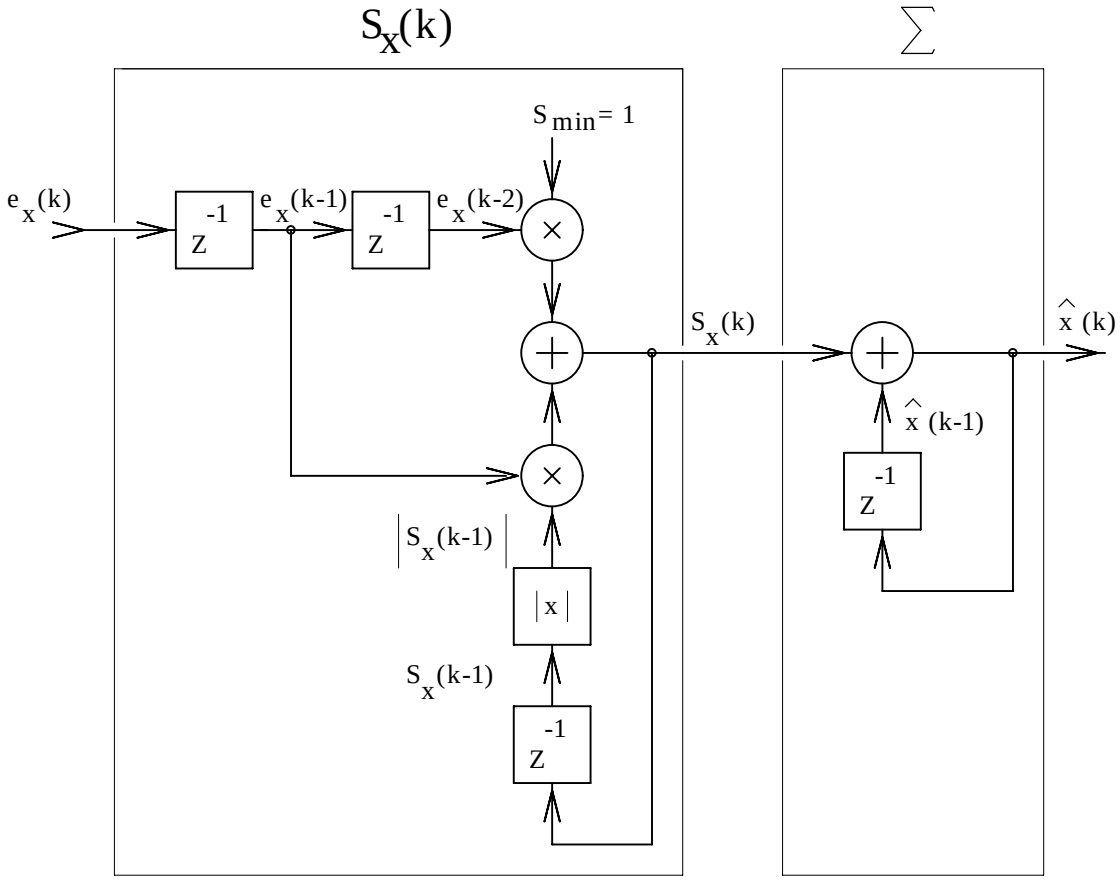


Figure 4.3. Receiver : Signal-Flow Graph

this figure. The program shown in Figure 4.4 is the final version of the algorithm. Many iterations of this algorithm were performed and these iterations required three weeks. A summary of designer experience and algorithm iteration techniques is given in section 2.2.6 (Design Trade Off Techniques).

The first part of the program is declarations. The first line of the program gives the title, `ADM_to_PCM` converter. As shown in Figure 4.3, this chip converts an ADM input, $e_x(k)$, to a PCM output, $\hat{x}(k)$. In the `CONST` section of the program, a variable is declared called `_data_width`. This variable is a reserved constant in SPIL which is used to set the width of the data bus. In this case, it means that all the variables declared in the `VAR` section will be 8 bits wide. Thus, the program will generate 8-bit PCM.

The first variable, `ADM_input`, is a register to bring off-chip values to the data bus when `ADM_input` is selected. The parameters `0..0 UPWARD` indicate that only bit 0, the least significant bit, is supposed to connect to the chip and that the off-chip input is suppose to run up the data path according to the orientation shown in Figure 2.3. This variable must contain $e_x(k)$ for the entire duration of the main program (between `BEGIN` and `END`.) in order to be latched by the hardware.

The second variable, `PCM_output`, is a register to latch data bus values when `PCM_output` is selected. The outputs of the register are taken off chip. The parameter `DOWNWARD` indicates the off-chip lines are supposed to down run data path according to the orientation shown in Figure 2.3. Since no bit range is specified (such as `0..0` for `ADM_input`), all eight register outputs will run off chip. This variable will contain $\hat{x}(k)$ at the end of the main program.

The third variable, `Ex`, is one 8-bit SPIL variable and one 8-bit register on the data path, but it holds two algorithm variables. The algorithm variables are $e_x(k-1)$ and $e_x(k-2)$. The least significant bit of Ex, bit 0, stores $e_x(k-1)$ at the beginning of the main program. The next significant bit of Ex, bit 1, stores $e_x(k-2)$ at the beginning of the main program.

The fourth variable, `Sx_of_k`, stores $S_x(k)$ at the end of the main program.

The fifth variable, `X_of_k`, stores x(k) at the end of the main program.

After the variable declarations is the `PROCEDURE _reset`. This is the only procedure that is allowed in SPIL. It specifies the operations to be performed when a hardware reset signal occurs. In this case, the procedure causes all three integer variables to be set to zero.

The remaining part of the program is between the `BEGIN` and `END`. statements. Here the algorithm is coded. The groups of statements in the program will be related to the operations in the signal flow graph in Figure 4.3.

The first statement, `IF Sx_of_k ...`, performs the absolute value at the output of the delay element, Z^{-1}, at the bottom of Figure 4.3.

The next statement, `IF Ex ...`, computes the output of the multiplication operator closest to the bottom of Figure 4.3. Note that the multiplication units in the signal flow graph only have to perform a multiplication with + 1 or -1,

and thus, the multiplication operation in the SPIL program can be performed using an `IF` statement.

The next group of statements spans six lines, starting with `_add_in_1 ...`. They perform three sequential functions in Figure 4.3. The first is to compute the output of the remaining multiplication operator. The second is to compute the output of the leftmost addition operator. The third is to propagate the value of $S_x(k)$ to the output of the delay element closest to the bottom of Figure 4.3.

The next statement, `X_of_k ...`, performs two sequential functions in Figure 4.3. The first is to compute the output of the rightmost addition operator. The second is to propagate the value of $\hat{x}(k)$ to the output of the rightmost delay element.

The next group of statements spans two lines starting with `Ex ...`. They perform two sequential functions. The first function is to propagate $e_x(k-1)$ to the output of the delay element which is fed directly from the output of the leftmost delay element. The second function is to read in the off-chip ADM signal and to propagate it, $e_x(k)$, to the output of the leftmost delay element.

The last statement, `PCM_output ...`, copies $\hat{x}(k)$ to the off-chip latches.

The various iterations of the receiver will not be discussed since the design techniques which were used to iterate the design have already been summarized in section 2.2.6 (Design Trade Off Techniques). A more detailed description of

the receiver may be found in Appendix A (SPIL Codec Files).

```
PROGRAM ADM_to_PCM ;

CONST
   _data_width = 8 ;

VAR
    ADM_input  : input_port  CONNECT 0..0   UPWARD ;
    PCM_output : output_port CONNECT      DOWNWARD ;

    Ex      ,        { Starting from LSB(0) : Ex(k-1), Ex(k-2) }
    Sx_of_k ,                    { Step to next predicted PCM }
    X_of_k  : integer ;                     { Last PCM output }

PROCEDURE _reset ;             { Chip initialization procedure }
BEGIN
    Ex      := 0 ;                { Make Ex(k-1) = Ex(k-2) = 0 }
    X_of_k  := 0 ;
    Sx_of_k := 0 ;
END ;

BEGIN
   IF Sx_of_k < 0    THEN Sx_of_k := 0 - Sx_of_k ;

   IF Ex = ????????0B THEN Sx_of_k := 0 - Sx_of_k ;

   _add_in_1 := Sx_of_k ;          { IF Ex(k-2) = 1 THEN       }
   IF Ex = ??????1?B THEN          {                           }
      _add_in_2 :=  1              {    Sx_of_k := Sx_of_k + 1 }
   ELSE                            { ELSE                      }
      _add_in_2 := -1 ;            {    Sx_of_k := Sx_of_k - 1;}
   Sx_of_k := _add_out ;           { (No over/under-flow check)}

   X_of_k := X_of_k + Sx_of_k ;    {  No over/under-flow check }

   Ex := Ex << 1 ;   { Shift signals left   i.e. one time step }
   IF ADM_input = ????????1B THEN Ex := Ex + 1 ;

   PCM_output := X_of_k ;
END.         { ADM_input must have remained valid all the time }
```

Figure 4.4. The Receiver Program

Figure 4.5 is a block diagram of the transmitter. It contains three parts. Two of the parts, the predictor and the summer, have already been described in the discussion about the receiver. The third block is a comparator, a combinational circuit which generates either + 1 or -1 depending on the two inputs. If the PCM input, x(k), is greater than or equal to the estimate, $\hat{x}(k)$, then the output of the comparator is 1; otherwise the output is -1.

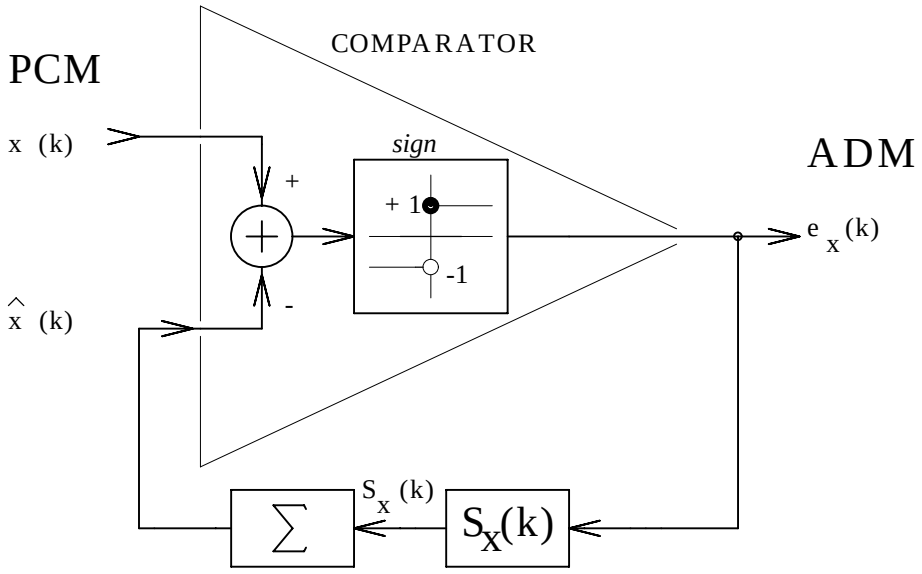


Figure 4.5. Transmitter : PCM-to-ADM Converter

The operation of the transmitter will now be described. Since the transmitter is based upon the receiver, equations 4.1 to 4.4 are still valid. PCM data is input to the transmitter at a rate f_N and ADM data is output at a rate N times higher, f_S. This means that a particular PCM sample should be applied to the input of the transmitter and held constant for N cycles so that N ADM data values can be calculated per PCM value.

The SPIL program which generates the transmitter will not be described in detail since it is very similar to the receiver. Only the significant differences between the transmitter and the receiver will be discussed. The program follows this description (Figure 4.6), but in order to help with the explanation, references will be made to Figure 4.3 and Figure 4.5 because the program implements, step by step, what is shown in these figures.

The first significant differences between the transmitter and the receiver are two variables. The first variable, `PCM_input`, is a register to bring off-chip values to the data bus when `PCM_input` is selected. The parameter `DOWNWARD` indicates the off-chip lines are supposed to down run data path according to the orientation shown in Figure 2.3. Since no bit range is specified (such as `0..0` for `ADM_output`), all eight register outputs will run off chip. This variable must contain x(k) for the entire duration of the main program in order to be latched by the chip.

The second variable, `ADM_output`, is a register to latch data bus values when `ADM_output` is selected. The parameters `0..0 UPWARD` indicate that only bit 0, the least significant bit, is supposed to run off chip and that the off-chip line is supposed to run up the data path according to the orientation shown in Figure 2.3. This variable will contain $e_x(k)$ at the end of the main program.

The remaining significant differences between the transmitter and the receiver are the last two groups of statements at the end of the transmitter program. The first of these two groups of statements spans two lines, starting with `Ex ...`. It performs five sequential functions. The first function is to propagate $e_x(k-1)$ to the output of the delay element which is fed directly from the output of the leftmost delay element shown in Figure 4.3. The second function is to read in the off-chip PCM data, x(k). The third function is to compute the output of the addition operator shown in Figure 4.5. The fourth function is to compute the output of the *sign* block shown in Figure 4.5. The fifth function is to propagate $e_x(k)$ to the output of the leftmost delay element shown in Figure 4.3. The last statement, starting with `ADM_output ...`, copies $e_x(k)$ to the off-chip latches.

A more detailed description of the transmitter may be found in Appendix A (SPIL Codec Files).

```
PROGRAM PCM_to_ADM ;

CONST
   _data_width = 8 ;

VAR
    PCM_input  : input_port  CONNECT      DOWNWARD ;
    ADM_output : output_port CONNECT 0..0   UPWARD ;

    Ex      ,        { Starting from LSB(0) : Ex(k-1), Ex(k-2) }
    Sx_of_k ,                    { Step to next predicted PCM }
    X_of_k  : integer ;                     { Last PCM output }

PROCEDURE _reset ;             { Chip initialization procedure }
BEGIN
    Ex      := 0 ;                { Make Ex(k-1) = Ex(k-2) = 0 }
    X_of_k  := 0 ;
    Sx_of_k := 0 ;
END ;

BEGIN
   IF Sx_of_k < 0    THEN Sx_of_k := 0 - Sx_of_k ;

   IF Ex = ???????0B THEN Sx_of_k := 0 - Sx_of_k ;

   _add_in_1 := Sx_of_k ;          { IF Ex(k-2) = 1 THEN       }
   IF Ex = ??????1?B THEN          {                           }
      _add_in_2 :=  1              {    Sx_of_k := Sx_of_k + 1 }
   ELSE                            { ELSE                      }
      _add_in_2 := -1 ;            {    Sx_of_k := Sx_of_k - 1;}
   Sx_of_k := _add_out ;           { (No over/under-flow check)}

   X_of_k := X_of_k + Sx_of_k ;    {  No over/under-flow check }

   Ex := Ex << 1 ;   { Shift signals left   i.e. one time step }
   IF PCM_input > X_of_k THEN Ex := Ex + 1 ;

   ADM_output := Ex ;
END.         { PCM_input must have remained valid all the time }
```

Figure 4.6. The Transmitter Program

After designing the transmitter and the receiver, it is possible to determine the minimum clock frequency of the codec chip in order to process 32 kHz ADM. From Appendix A (SPIL Codec Files), the transmitter requires 34 states during its worst case of input and state conditions to generate an ADM output.

$$32000\frac{\text{bits}}{\text{second}} \quad 34\frac{\text{states}}{\text{bit}} \quad = \quad 1.088\text{x}10^{6}\frac{\text{states}}{\text{second}}$$

Since one FSM state corresponds to one clock signal, the clock frequency of the chip will have to be 1.088 MHz, or higher.

4.3. EPAD Analysis

After SPIL generated the design example, the chip layout was completed by manually interconnecting the data path and the FSM using the layout editor, Caesar. This was done for both the receiver and the transmitter. EPAD was run on the layout which was generated by SPIL. The files for the EPAD run are in appendix B (EPAD Files).

This section about the EPAD analysis is divided into two parts. The first part is on logic simulation for the purpose of verifying the chip generated by SPIL and the algorithm which has been coded in the SPIL program. The second part is on critical path simulation for the purpose of determining how fast the chip can be clocked. Recall that these simulations are performed using SILOS and that the circuit description file for SILOS was generated automatically from

EPAD.

4.3.1. SILOS Logic Verification

The sample output shown in appendix C (SILOS Logic Files), file *output*, demonstrates part of the operation of the receiver: the ADM-to-PCM converter. Only a few salient simulation time points are shown in this sample output. The times in the simulation are in nanoseconds. The PHI1- and PHI2 clocks have periods in this simulation of 1000 nanoseconds.

At time 0, the receiver reset signal, NRRESET, is high. The reset signal is lowered at 875 after it is latched by the FSM's input latches. This is followed by the chip executing its reset procedure. The signal, NBREADY, which indicates that the reset procedure is finished, rises at time 4772. Both the ADM input signal, NADM_IN, and the go signal, NRGO, are high at time 5625 which causes the chip to constantly convert ADM values of 1 to PCM values. It takes between 20000 and 30000 nanoseconds to perform one conversion with the 1 MHz simulation clocks. Thus, at time 30772 when the ready signal rises again, the initial PCM value has been computed. The PCM output is shown by the signals RPCMOUT_H and RPCMOUT_L which are the most significant and least significant hexadecimal values. The PCM output at 30772 is FF (-1 decimal). The next PCM output at 55772 is FF (-1 decimal). The next PCM output at 76772

is 00 (0 decimal). The PCM outputs for ADM inputs are summarized in Table 4.2. These PCM outputs were verified using equations 4.1 to 4.4.

Table 4.2. Receiver Logic Verification Example

Simulation Time	Time Point	ADM Input $e_x(k)$	Step Size $S_x(k)$	PCM Output $\hat{x}(k)$
	-3			
	-2	-1		
	-1	-1	0	0
30772	0	1	-1	-1
55772	1	1	0	-1
76772	2	1	1	0
97772	3	1	2	2
118772	4	1	3	5
139772	5	1	4	9
160772	6	1	5	14
178772	7	-1	6	20
200772	8	-1	-5	15
226772	9	-1	-6	9
252772	10	-1	-7	2
278772	11	-1	-8	-6
304772	12	-1	-9	-15

The transmitter was verified in a similar manner to the receiver since they are very similar.

4.3.2. SILOS Critical Path Analysis

Appendix D (SILOS Critical Path Files), file *output*, shows how a sample propagation time was determined. The propagation time through the input latches of the adder and the adder itself, $t_{dlat,adder}$, was determined from the last column of the simulation output which shows the outputs of the adder settling at time `13574`. Since the input latches of the adder were enabled at time `13500`, it took 74 nanoseconds to propagate through the adder latches and the adder.

The propagation times in Table 4.3 were calculated in a similar manner. The terms used in Table 4.3 have already been described in equations 2.1 to 2.6. T_{min} was calculated using equation 29. The most important result of this section is the prediction that the chip will run faster than 1.088 MHz. The 1.088 MHz frequency is discussed at the end of section 4.3 (Algorithm Design). Thus, EPAD predicts that the clock frequency is high enough to permit the codec to compress and decompress 8 kHz PCM speech signals.

The transmitter and receiver contain 4996 transistors. The total power dissipations of the codec chip from EPAD-0, *kcapseries=0.0*, and from EPAD-1, *kcapseries=1.0*, are 27.0mW and 31.4mW when operating at 1 MHz. EPAD predictions for the codec chip are summarized in Table 4.4.

Table 4.3. Detailed Codec Propagation Delays (ns)

Times	Receiver		Transmitter	
	EPAD-0	EPAD-1	EPAD-0	EPAD-1
t_{tol}	0	0	0	0
t_p	6	6	6	6
t_{adder}	54	54	54	54
$t_{PLA,OL,src}$	73	155	76	177
t_c	6	6	6	6
$t_{PLA,OL}$	44	126	48	149
$t_{PLA,OL,dst}$	69	151	72	173
t_{dlat}	29	29	29	29
t_{IL}	66	66	70	70
T_{min}	145	227	152	253
f_{max}	6.90	4.40	6.58	3.95

Table 4.4. Design Summary for PCM-ADM Coder-Decoder

Parameter		EPAD	
		EPAD-0	EPAD-1
Area $(\mu m)^2$	Tx/Data Path	1633 x 1703	1633 x 1703
	Tx/Control Path	1601 x 2069	1601 x 2069
	Rx/Data Path	1633 x 1703	1633 x 1703
	Rx/Control Path	1460 x 1883	1460 x 1883
	Codec	4511 x 4511	4511 x 4511
Clock Frequency (MHz)	Tx	6.58	3.95
	Rx	6.90	4.40
	Codec	6.58	3.95
Power Dissipation (mW)	Tx	11.5	13.9
	Rx	15.5	17.9
	Codec	27.0	31.4

4.4. Test Plan

The architecture which SPIL generates does not contain any testing structures, such as a scan path [33]. Four methods of were considered.

The first method considered was the addition of test structures to test the basic parameters of the process. This test structure is known as a *test insert* and was placed in an unused area of the chip. If there is some question about the correctness of the fabrication process, then these test structures can be probed. The test structure is described by the schematic shown in Figure 4.7. This structure consists four sub-structures. The first sub-structure contains long poly lines and contact chains, the second an N-Channel transistor, the third a P-Channel transistor, and the fourth a CMOS inverter.

The second test consideration was to determine the correctness of the input and output pads. This test structure consists of an input cell connected directly to an output cell. Since the I/O cells were characterized at the CMC, the I/O cells on the three chips could be evaluated in terms of the CMC's previous results [34].

Third, the chip contains a test structure to observe the eight data bus values in the receiver. A single data bus line is observed by connecting the data bus line to the gate of an N-Channel transistor. The source of this transistor is grounded. The drain of the transistor is connected to a probe pad. This open-

drain structure can be used to observe changes in the data bus signals to a resolution of 0.2 ns with a negligible effect on the data bus [35].

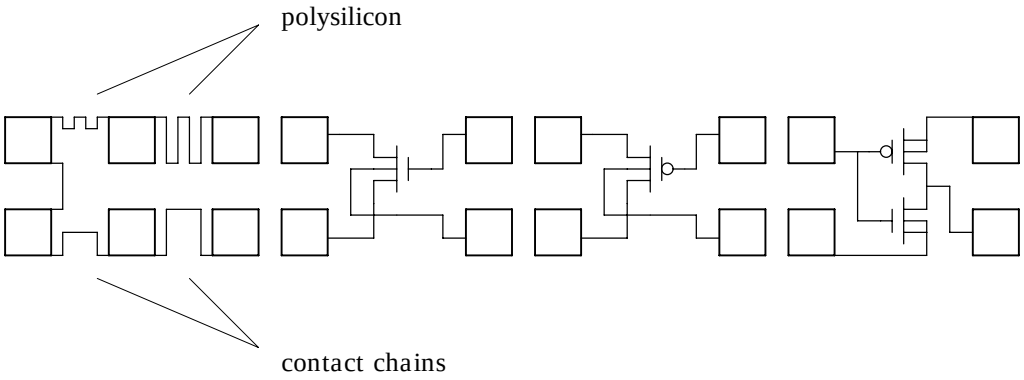


Figure 4.7. The Test Insert

The fourth test consideration was the development of a test pattern, and the single-stuck-at fault coverage of the test pattern is 80.1 percent. It was found that 9.0 percent of the undetected faults are redundancies in the SPIL architecture. Appendix E (SILOS Fault Simulation Files) contains the results of the SILOS fault simulation.

4.5. Submission for Fabrication

The codec was sent for fabrication on January 7, 1987. The chip is identified by the logo WTBRP in the upper right corner. The pad frame for the codec was supplied by the CMC. The pad frame specified the maximum dimensions of the chip that could be submitted. The codec used the largest possible one, the *A* pad frame which is 4511 μm by 4511 μm. This pad frame is restricted to having 10 bonding pads per side of the chip. See the plot of the chip in appendix B (EPAD Files), file *codec.cif*.

The input and output pads that were employed were developed at CMC for use by researchers using the VLSI implementation service [25]. There are two criteria for selecting an I/O cell. One criteria is simply whether an input cell or an output cell is desired. The second criteria is the aspect ratio of the desired cell. There are two possible aspect ratios, narrow (type X) and square (type Y). Thus, there are four different I/O cells: XIN, XOUT, YIN and YOUT. The codec used the Y cells since they do not extend as far into the chip as the X cells. If Y cells are used, the working area of the *A* pad frame is reduced to 3911 μm by 3911 μm. The functions of the I/O cells are now described. The input cells pass the input signal directly to the inside of the chip, but this signal is clamped to the upper and lower power rail voltages by a PNP and an NPN transistor, respectively. The output cells could have been used as tri-state

buffers, but were not. The output cells were used as non-inverting buffers. These output cells are capable of driving loads such as 50 pF with propagation delays on the order of tens of nanoseconds.

By sharing only the V_{DD} and GND connections for the transmitter and receiver, the chip requires 32 external pins.

Figure 4.8 contains a photomicrograph of the ADM-PCM Coder-Decoder.

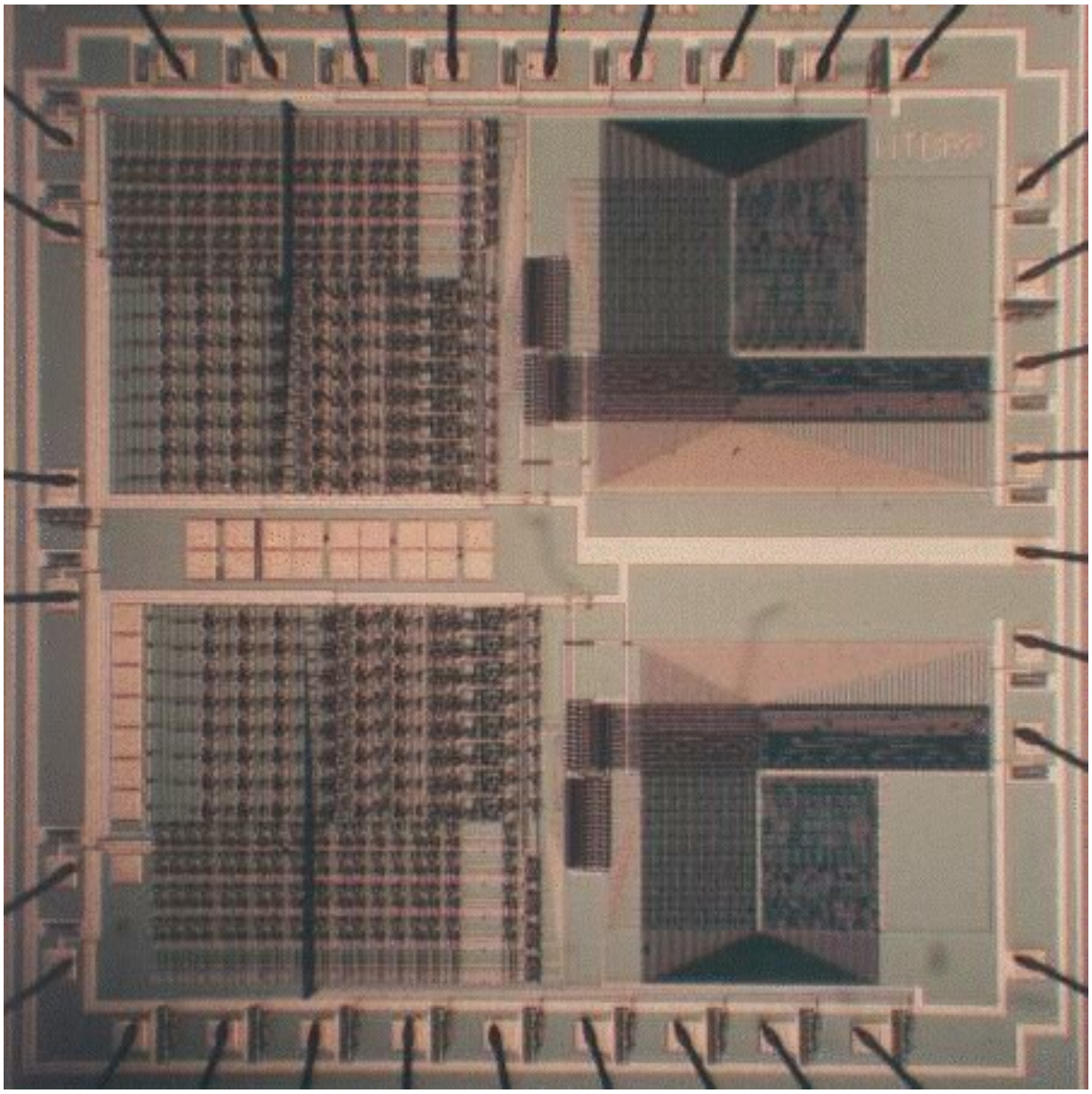


Figure 4.8. ADM-PCM Coder-Decoder Photomicrograph

CHAPTER 5

Test Results and Suggested Enhancements

This chapter describes three tests that were performed on the fabricated codec chips. These tests verified the logic, determined the maximum clock frequency and determined the power dissipation. The results are discussed, including the calibration of EPAD, and enhancements are suggested.

5.1. Logic Verification

A block diagram of the test apparatus for logically verifying the receiver is shown in Figure 5.1. The data generator was used to store and generate the test patterns for the **D**evice **U**nder **T**est (DUT). An ammeter was employed to obtain the static drain current of the DUT. The data analyser captured the DUT outputs and to compared the captured data with the expected data.

The following method was used to test the chips.

The data generator was set up as follows. The initial clock frequency for the test was chosen to be 1 MHz, since it was predicted that the chips would operate at speeds of at least that frequency. Because the chips require two-phase non-overlapping clocks, the width of each of the four clock phases was initially set to 250ns. The data generator permitted the width and duration of each clock

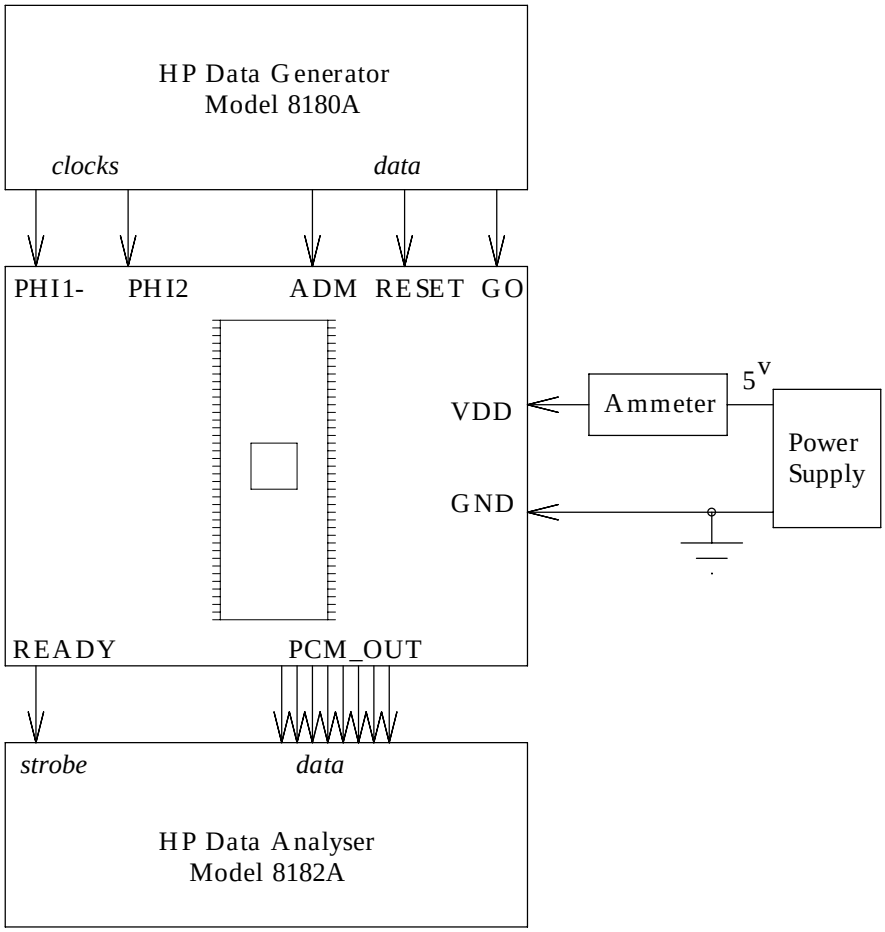


Figure 5.1. Test Apparatus

pulse to be specified. During testing the clock frequency was increased to the maximum possible frequency, as shown in Figure 5.2. The test patterns were entered into the data generator. There were 1024 words of data stored, where each word was three bits wide. The test data that was used is shown in Figures 5.3 and 5.4. Only the parts of the test data where changes occurred have been shown in these two figures. The three bits of data that were used are the last

three bits of the first column of four-bit words. These three bits correspond to the chip signals : ADM_IN, RESET and GO.

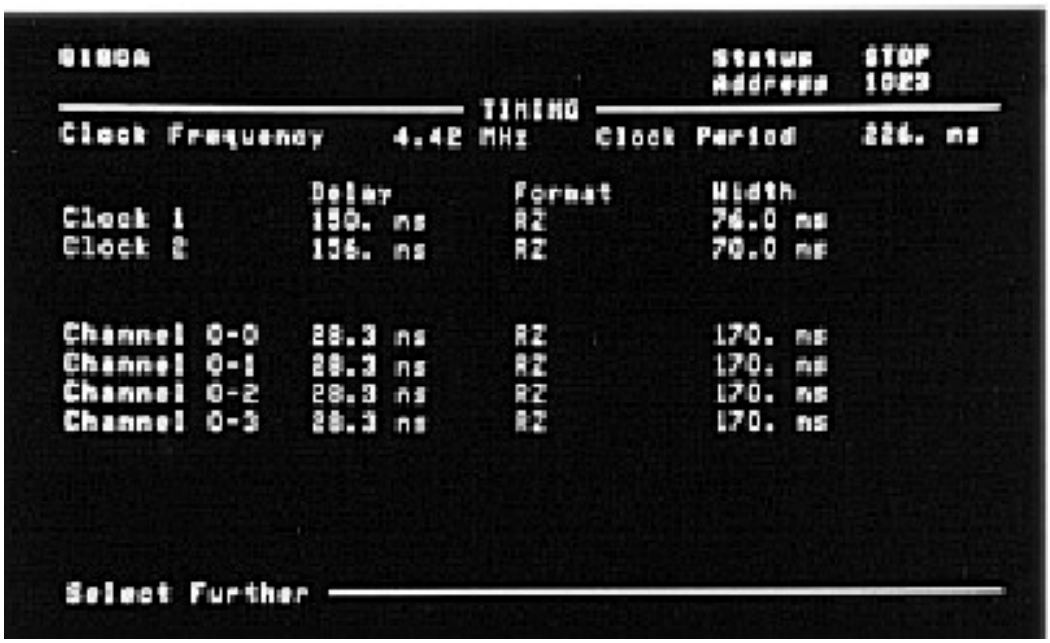


Figure 5.2. Data Generator - Clocks

As the 1024 test patterns were run through the chip, the data analyser captured the DUT outputs. The READY signal from the DUT is connected to the clock strobe of the data analyser. The PCM outputs of the DUT are connected to the data inputs of the data analyser. Each time the READY signal goes high, signifying a new PCM output, the data analyser captures it. Since it takes about 25 input patterns to generate one output pattern, there are 40 bytes stored in the data analyser. The captured data from the data analyser is shown in Figures 5.5, 5.6 and 5.7. This captured data was verified with the results of fault

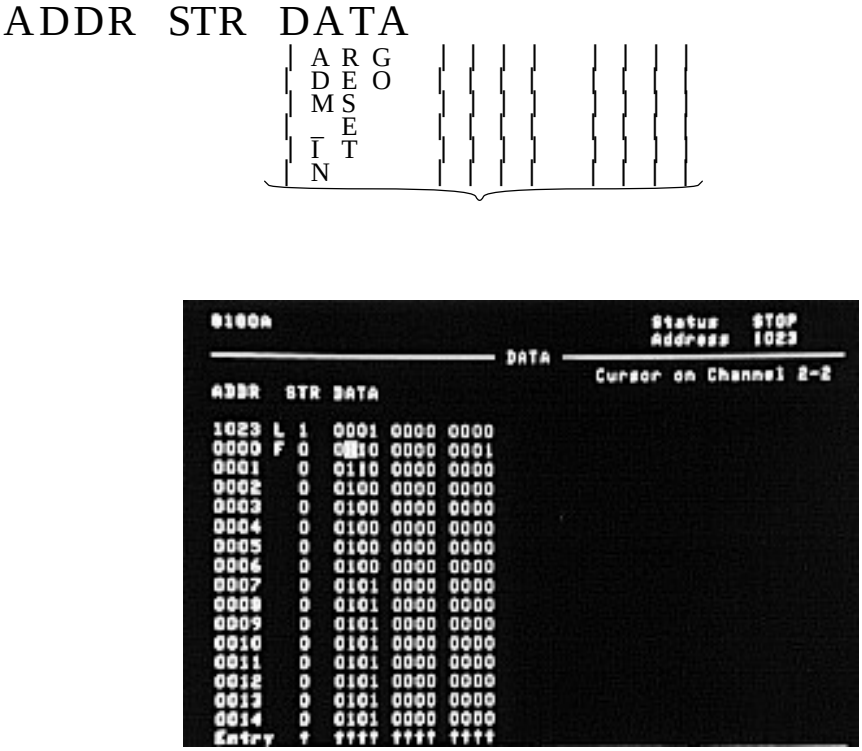


Figure 5.3. Data Generator - Data 1

simulation and showed that the chips performed functionally.

The test pattern that was used was run through a fault simulator, SILOS, and the fault coverage of the pattern is 80.1%. However, 9.0% of the faults cannot be detected by any pattern. This is because they are redundancies in the SPIL output which means that only 10.9% of the faults that are detectable have not been detected by the pattern.

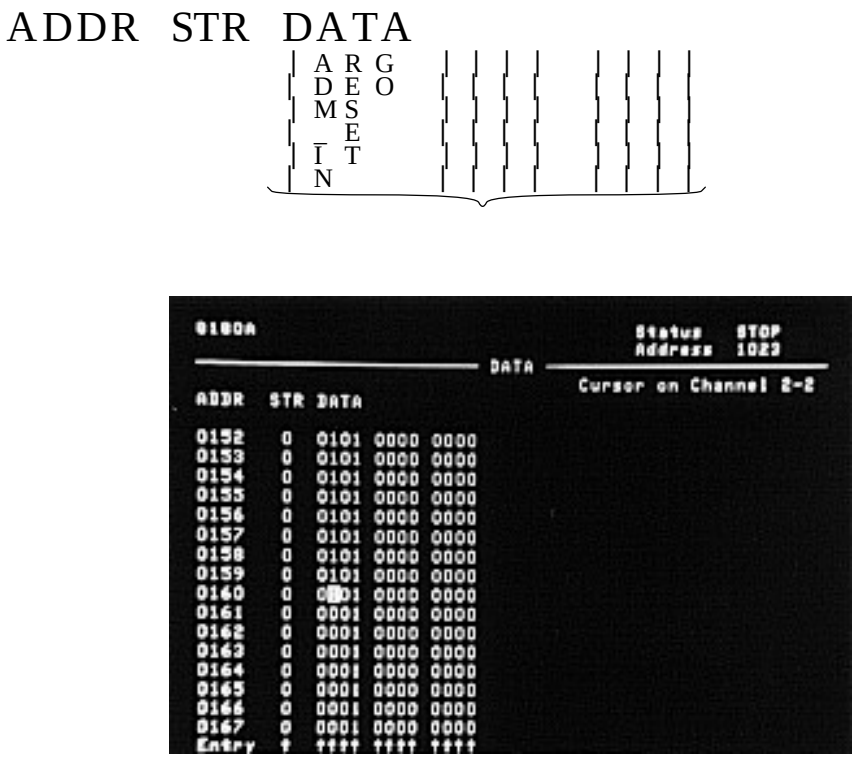


Figure 5.4. Data Generator - Data 2

The transmitter was logically verified by performing a *back-to-back* test with the receiver. The test was possible because the transmitter was built upon the same logic blocks present in the receiver. Note that this test was not a simple case of converting PCM to and from ADM and then verifying that the PCM out of the receiver was the same as the PCM put into the transmitter. The test is not simple is because the Song conversion algorithm introduces noise into the signal during the conversions so that the PCM output from the receiver is close, but not exactly equal to the PCM which is put into the transmitter. Instead, the *back-to-back* test made the input and state conditions the same on the predictor

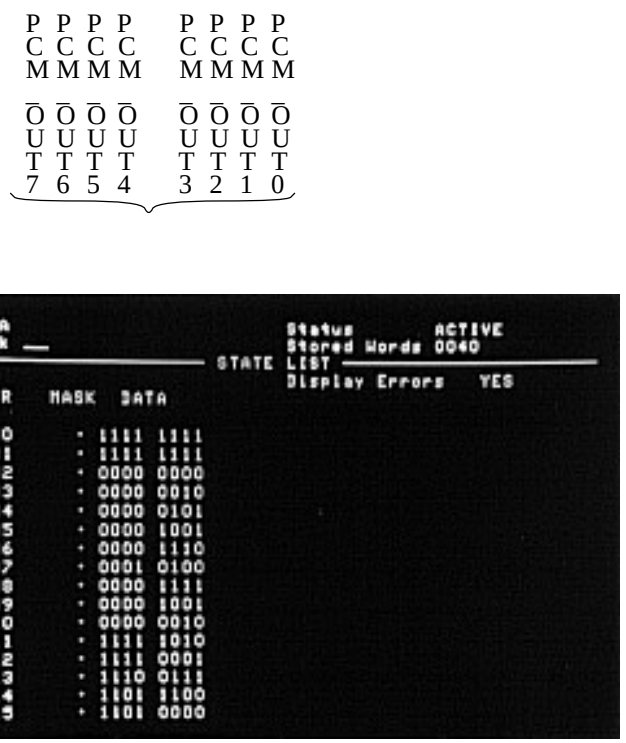


Figure 5.5. Data Analyser Observations 1

and summer blocks which exist in both the transmitter and the receiver (Figures 4.1 and 4.4).

The apparatus for the test is similar to the setup for the receiver; only the differences will be described here. Both the transmitter and receiver READY signals were ANDed together to form the one GO signal which was connected to both the transmitter and the receiver. The AND gate synchronized the operation of the transmitter and the receiver because the transmitter works slower than the receiver. The GO signal previously connected to the data generator

DATA

PCM OUT 7	PCM OUT 6	PCM OUT 5	PCM OUT 4	PCM OUT 3	PCM OUT 2	PCM OUT 1	PCM OUT 0

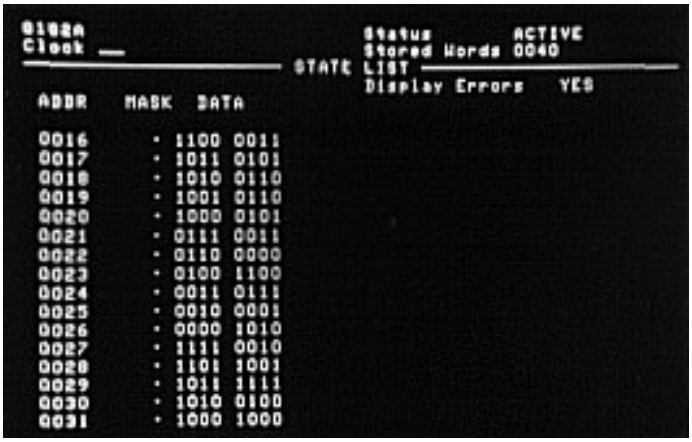


Figure 5.6. Data Analyser Observations 2

was discarded. The PHI1- and PHI2 clocks of the transmitter were connected to the same clocks of the receiver. The RESET signal from the data generator was also connected to the transmitter. The ADM_IN signal from the data generator was put into a network to convert it to a PCM signal for the transmitter. The input to output mapping of this network is shown:

DATA

PCM OUT 7	PCM OUT 6	PCM OUT 5	PCM OUT 4	PCM OUT 3	PCM OUT 2	PCM OUT 1	PCM OUT 0

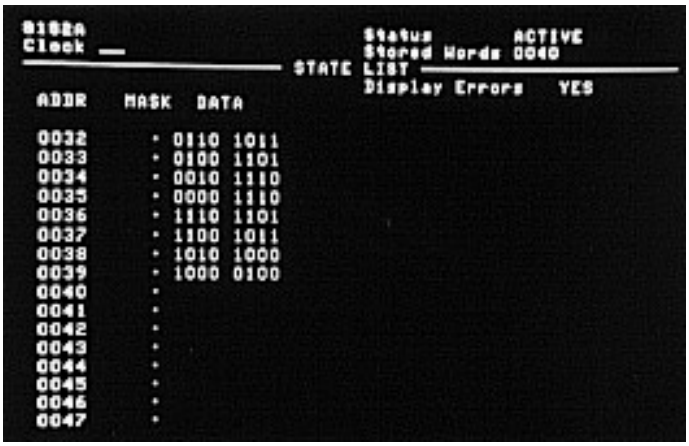


Figure 5.7. Data Analyser Observations 3

Input Receiver ADM_IN	Output Transmitter PCM_IN
0	(MSB) 10010101 (LSB)
1	01101010

The programming in the data generator underwent two changes. The first is that the ADM_IN column of data, which was previously 1 between pattern number 0 and 160 (Figures 5.3 and 5.4) was extended to be 1 between pattern

number 0 and 205 in order to maintain the same test as the one in the receiver verification. The receiver slows down because it must occasionally wait for the transmitter. To maintain the same logical test, the input data had to be extended. The second programming change was to make the last address 640 instead of 1023. It was not possible to use the entire receiver pattern to verify the transmitter. Beyond pattern number 640, the PCM output of the receiver underflows; that is, the receiver PCM output attempts to go below the range 127 to -128. This was intended with the receiver test pattern in order to achieve a higher fault coverage; however, it meant that the transmitter could not be tested beyond the underflow limit.

Figure 5.8 contains the data that was captured by the data analyser. TX means transmitter. RX means receiver. Note that the transmitter's ADM_OUT and the receiver's ADM_IN are the same.

```
                        TTTT TTTT        TR RRRR RRRR
                        XXXX XXXX        XX XXXX XXXX

                        PPPP PPPP        AA PPPP PPPP
                        CCCC CCCC        DD CCCC CCCC
                        MMMM MMMM        MM MMMM MMMM

                        ____ ____        __ ____ ____
                        IIII IIII        OI OOOO OOOO
                        NNNN NNNN        UN UUUU UUUU
                        7654 3210        T  TTTT TTTT
                                            7654 3210

8182A                         Status       ACTIVE
Clock                         Stored Words 0020
------------------- STATE LIST -------------------
                              Display Errors   YES
ADDR   MASK  DATA

0000     . 0000 0000 0110 1010 0011 0011 1111 1111
0001     . 0000 0000 0110 1010 0011 0011 1111 1111
0002     . 0000 0000 0110 1010 0011 0011 0000 0000
0003     . 0000 0000 0110 1010 0011 0011 0000 0010
0004     . 0000 0000 0110 1010 0011 0011 0000 0101
0005     . 0000 0000 0110 1010 0011 0011 0000 1001
0006     . 0000 0000 0110 1010 0011 0011 0000 1110
0007     . 0000 0000 1001 0101 0011 0000 0001 0100
0008     . 0000 0000 1001 0101 0011 0000 0000 1111
0009     . 0000 0000 1001 0101 0011 0000 0000 1001
0010     . 0000 0000 1001 0101 0011 0000 0000 0010
0011     . 0000 0000 1001 0101 0011 0000 1111 1010
0012     . 0000 0000 1001 0101 0011 0000 1111 0001
0013     . 0000 0000 1001 0101 0011 0000 1110 0111
0014     . 0000 0000 1001 0101 0011 0000 1101 1100
0015     . 0000 0000 1001 0101 0011 0000 1101 0000
0016     . 0000 0000 1001 0101 0011 0000 1100 0011
0017     . 0000 0000 1001 0101 0011 0000 1011 0101
0018     . 0000 0000 1001 0101 0011 0000 1010 0110
0019     . 0000 0000 1001 0101 0011 0000 1001 0110
```

Figure 5.8. Transmitter Verification

5.2. Maximum Clock Frequency Determination

The test setups to determine the maximum clock frequency of the codec were similar to the test setups used to logically verify the the transmitter and the receiver. The only differences in the setups were the clock frequencies, which were programmed differently into the data generator.

The maximum clock frequency was determined by reducing each of the four clock phases one at a time until an error was detected in the observed output pattern. Errors were observed because the test equipment can be programmed to automatically highlight those bits which differ from what is expected. The expected data was obtained from the logic verification tests. Finding the minimum clock phase can be done efficiently in a manner similar to a binary search.

Tables 5.1 and 5.2 contain the results of testing. Note that EPAD predicted the clock frequency of the receiver to be between 4.40 MHz and 6.90 MHz. Also, EPAD predicted the clock frequency of the transmitter to be between 4.40 MHz and 6.90 MHz. See section 4.3 (EPAD Analysis) These bounds correspond to the worst and best case capacitive loading conditions on gates within the layout. From Tables 5.1 and 5.2, lines 1,3,4 and 5, the mean values of the maximum frequencies were determined. The maximum clock frequency of the receiver is 4.27 MHz. The maximum clock frequency of the transmitter

is 3.11 MHz. The accuracy of EPAD's worst-case-loading prediction for the receiver is :

$$\text{EPAD-1 Accuracy} = \frac{|4.40 - 4.27|}{4.27} \text{x} 100\%$$

$$= 3.0\%$$

The accuracy of EPAD's worst-case-loading prediction for the transmitter is :

$$\text{EPAD-1 Accuracy} = \frac{|3.95 - 3.11|}{3.11} \text{x} 100\%$$

$$= 27.0\%$$

Table 5.1. Maximum Clock Frequency of the Receiver

Chip Number	Minimum Clock Phases(ns)				Minimum Period (ns)	Maximum Clock Frequency (MHz)
	1	2	3	4		
1	149	6	70	0	225	**4.42**
2	222	61	70	0	352	2.83
3	140	29	70	0	239	**4.18**
4	149	14	70	0	233	**4.29**
5	150	20	70	0	240	**4.17**
					Mean	**4.27**

Chip 2 contains a yield error.

In Tables 5.1 and 5.2 the duration of clock phase 3 is always 70 ns. This is by choice. The sum of the durations of phases 1, 3 and 4 is a constant for any DUT. After the durations of phases 3 and 4 are chosen, the value of phase 1 must be obtained by measurement. During testing, some phases could be

Table 5.2. Maximum Clock Frequency of the Transmitter

Chip Number	Minimum Clock Phases(ns)				Minimum Period (ns)	Maximum Clock Frequency (MHz)
	1	2	3	4		
1	243	0	70	0	313	**3.19**
2	256	0	70	0	326	3.07
3	243	0	70	0	313	**3.19**
4	257	0	70	0	327	**3.06**
5	265	0	70	0	335	**2.99**
					Mean	**3.11**

Chip 2 contains a yield error.

reduced to zero and still maintain correct output results; however, this may not be a reliable way to operate the chip. No reliability tests were performed. The reason phase 2 for the receiver could not be reduced to zero is most likely due to the dominant term in the critical path being $t_{PLA,OL}$ instead of t_C (Figure 2.11). This dominant term would be due to a slower-settling READY signal compared to the transmitter. The idea of a slower-settling receiver READY signal is to some extent confirmed by chip number 2's yield error. Chip number 2 has a longer phase 2 and this chip's yield error was observed to be related to the READY signal which never went lower than one volt above ground. The data path's for the receiver and transmitter are similar, but the Boolean logic in the FSM's is not similar and this could affect the results.

5.3. Power Dissipation

The first test that was performed was to determine the static drain current. The only purpose of this test is to check for yield defects which cause large static drain currents. Both the receiver and the transmitter clocks were set so as to put them into phase 4; no actions occur in phase 4. The results are shown in Table 5.3. Chip number 2 shows a relatively large drain current. Further analysis of chip 2 showed that the output signal READY was at one volt when it should have been at zero volts. It is assumed that this problem is due to a yield defect. All of the other chip's static currents initially seemed high for 5000 transistors in a 3μm CMOS technology. This was due to having many N-channel or P-channel transistor transmission gates; that is, non-CMOS transmission gates. Since these gates are not complementary, they do not pass all voltages equally. For example, the output of an N-channel transmission gate will only go as high as $V_{DD} - V_{threshold}$. This will mean that the gate whose input is at the output of the transmission gate will not be completely switched off. This hypothesis about the cause of the high static drain current was verified by the observation of the current immediately after the test patterns were run through the DUT, during previous logic verification testing. The current slowly decayed over a period of minutes. The current decay corresponds to the outputs of single transistor transmission gates settling to final voltages as the transistors go into cutoff.

Table 5.3. Static Drain Current of the Codec

Chip Number	Current (mA)
1	0.316
2	9.036
3	0.356
4	0.345
5	0.371

The dynamic power dissipation of the transmitter, receiver and codec were determined. The test apparatus was identical to that of the corresponding logical verification, except for the following points. There was a 4.7 ohm resistor put in series with the ground of the chip and the system ground. An HP3400 True RMS Voltmeter with a 10 Hz to 10 MHz bandwidth was connected across this resistor. The codec power dissipation was measured using the *back-to-back* test apparatus. The transmitter power dissipation was measured using the *back-to-back* test apparatus, except that the receiver was disconnected and put into state 4 and the transmitter GO signal was obtained from the data generator instead of an AND gate.

This voltmeter was used to measure the RMS value of voltage across the resistor, and by Ohm's law, determine the RMS value of the AC component of the current. Since testing was done with clock frequencies in the kHz range, it was assumed that the AC component of the current below 10Hz was negligible. The HP3486A multimeter was still kept in series with V_{DD} in order to measure

the DC component of the chip current.

Measurements were taken for frequencies from 20 kHz to 500 kHz in 20 kHz increments. At each frequency, two measurements were taken: the DC power supply current and the AC RMS voltage across the 4.7 Ω resistor.

The total chip power at a given frequency was determined from the following equations.

$$I_{DD,RMS}^2 = I_{DD,DC,RMS}^2 + \left[\frac{V_{R,RMS}}{R}\right]^2 \quad (5.1)$$

$$P_{Total} = I_{DD,RMS} V_{DD} - R I_{DD,RMS}^2 \quad (5.2)$$

where:

$I_{DD,RMS}$ is the RMS value of the power supply current, the drain current.

$I_{DD,DC,RMS}$ is the RMS value of the DC component of the power supply current.

$V_{R,RMS}$ is the RMS value of the AC component of the voltage across the resistor R.

R is the resistor in series with the chip's ground. $R = 4.7\Omega$

V_{DD} is the power supply voltage. $V_{DD} = 4.989^{v}$

P_{Total} is the total power dissipation of the chip.

After the total powers were computed for each frequency, a linear regression was performed on power versus frequency. The line was extrapolated to 1 MHz to obtain the dynamic power dissipation at 1 MHz. The *y-intercept* of the line gave the static power. This analysis was performed for each of the four chips which did not have yield errors. It was performed for the transmitter, receiver and codec. The results are summarized in Tables 5.4, 5.5 and 5.6.

Table 5.4. Receiver Power Dissipation (mW) (Clocked at 1 MHz)

Chip Number	P_{Static}	$P_{Dynamic}$	P_{Total}
1	3.0	15.1	18.2
3	3.1	14.3	17.4
4	3.1	14.8	17.8
5	3.4	14.4	17.8
Mean	3.2	14.6	17.8

Table 5.5. Transmitter Power Dissipation (mW) (Clocked at 1 MHz)

Chip Number	P_{Static}	$P_{Dynamic}$	P_{Total}
1	2.3	9.9	12.3
3	2.3	10.3	12.5
4	2.2	10.2	12.4
5	2.6	9.5	12.1
Mean	2.3	10.0	12.3

Table 5.6. Codec Power Dissipation (mW) (Clocked 1 MHz)

Chip Number	P_{Static}	$P_{Dynamic}$	P_{Total}
1	4.0	27.0	31.0
3	4.0	26.9	30.9
4	3.1	29.5	32.6
5	4.0	26.9	30.9
Mean	3.8	27.6	31.3

5.4. ADM-PCM Codec Chip Summary

The results for the ADM-PCM codec are shown in Table 5.7. The results are a summary of Tables 4.4, 5.1, 5.2, 5.4, 5.5 and 5.6.

Table 5.7. Design Summary for PCM-ADM Coder-Decoder

Parameter		EPAD			Chip Test Results
		EPAD-0	EPAD-1	Calibrated EPAD-*kcapseries*	
AREA $(\mu m)^2$	Tx/Data Path	1633 x 1703	1633 x 1703	-	1633 x 1703
	Tx/Control Path	1601 x 2069	1601 x 2069	-	1601 x 2069
	Rx/Data Path	1633 x 1703	1633 x 1703	-	1633 x 1703
	Rx/Control Path	1460 x 1883	1460 x 1883	-	1460 x 1883
	Codec	4511 x 4511	4511 x 4511	-	4511 x 4511
CLOCK FREQUENCY (MHz)	Tx	6.58	3.95	1.32±0.02	3.11
	Rx	6.90	4.40	1.05±0.05	4.27
	Codec	6.58	3.95	1.19±0.15	3.11
POWER DISSIPATION (mW)	Tx	11.5	13.9	0.344±0.071	12.3
	Rx	15.5	17.9	0.958±0.136	17.4
	Codec	27.0	31.4	0.651±0.344	30.9

The area of the codec is the largest that CMC permits fabricating. The complexity of the chip in terms of the number of FSM states in the receiver and transmitter is 33 and 38, respectively.

The chip results for the maximum clock frequency show that the mean value of the chip clock frequencies are not bounded by the EPAD predictions of the clock frequencies. This result is discussed later in this section. The mean values and sample standard deviations of the clock frequencies are 4.27 ± 0.12 for the receiver and 3.11 ± 0.10 for the transmitter. The sample size is 4, the 4 working chips. Even though the sample size is small, it predicts the variations in

the overall effect of process parameters much better than an analysis of individual process parameter variations. The range of frequencies within one standard deviation is 4.39 to 4.15 MHz for the receiver and 3.21 to 3.01 MHz for the transmitter. For a normal distribution, 14 percent of fabricated receivers will have delays which are in the range predicted by EPAD. Essentially, none of the fabricated transmitters will ever be in the frequency range predicted by EPAD. The reason for the lower measured frequencies is most likely the effect of polysilicon resistance. In the data path, the power runs vertically in polysilicon. There are polysilicon links between metal parts of the data buses. Finally, in the FSM, there are long polysilicon lines for the minterms. The effect of polysilicon resistances is not taken into account in the EPAD delay models. This result advocates a redesign of the SPIL cell library and the control path in order to reduce the amount of polysilicon used. Recall that the transmitter and the receiver use identical data paths. The decrease in speed in the transmitter is solely due to the the effect of the control path having 5 more states. Even though there is no physical interpretation of *kcapseries* when it is beyond the range 0 to 1, this parameter can still be used to calibrate EPAD to the current version of the SPIL data path cell library and the control path generator. The mean and standard deviations of *kcapseries* averaged over the four receivers and four transmitters are shown in Table 5.7.

The chip results show that the mean value of the power dissipations is bounded by the EPAD predictions. This result is discussed later in this paragraph. The mean values and sample standard deviations of the power dissipations are 17.8 ± 0.3 for the receiver and 12.3 ± 0.2 for the transmitter. The sample size is 4, the 4 working chips. For a normal distribution, 54 percent of fabricated receivers will have power dissipations which are in the range predicted by EPAD. Essentially all of the fabricated transmitters will be in the power dissipation range predicted by EPAD. Since EPAD assumes that all gates are switching every clock cycle, the EPAD power dissipation predictions should have been higher than test results. That the chip power dissipation is too high is most likely due to three reasons. The first is the presence of non-CMOS transmission gates in latches of the FSM and data path. Gates connected to the outputs of non-CMOS transmission gates have higher static power dissipation, and this is not taken into account in the EPAD power models. Second, chip capacitances may be larger than nominal process values. This is confirmed by chip clock frequencies being lower than predicted values. Third, chip static and dynamic power dissipation may be higher due to leakage currents through input protection devices. All three of these effects could cause chip power disspation to be higher than initially expected, especially since the chip power dissipation is very low, $\frac{1}{20}^{\text{th}}$ of the maximum for the chip package. Since these effects should not

change radically from one SPIL design to the next, the single EPAD parameter *kcapseries* will be used as the basis for calibrating EPAD. The mean and standard deviations of *kcapseries* averaged over the four receivers and four transmitters are shown in Table 5.7.

Using SPIL reduces the design *turn-around* time. After the receiver was designed, the author was familiar with chip design using SPIL and EPAD. The transmitter was subsequently designed, verified and prepared for fabrication within nine days, compared to an estimated six months required using low-level layout design tools; an improvement of more than a factor of ten. This comparison is based on designers who have experience with their respective CAD tools. No comparison of times required for a designer to become familiar with the appropriate CAD tools is given. However, the learning time for SPIL will not be greater than learning time for low-level design tools, assuming a designer has no previous experience in chip design. A reasonable estimate of the time for a designer to become thoroughly familiar with chip design using SPIL is two months.

The designer can efficiently iterate designs using SPIL. To generate the CIF files for the receiver required 43 seconds of CPU time on a VAX 11/785. SPIL required 1 second, Busgen 7 seconds and PLAmate 35 seconds. To generate the CIF files for the transmitter required 52 seconds of CPU time on a

VAX 11/785. SPIL required 1 second, Busgen 8 seconds and PLAmate 43 seconds. Furthermore, the time complexity of SPIL and Busgen are linear, meaning the CPU time required grows linearly with the number of lines in the SPIL input file. The CPU time required by PLAmate grows non-linearly; however a finite state machine having about 65 states fills the largest CMC pad frame but only requires approximately five minutes of CPU time.

5.5. Suggested Enhancements and Future Work

Suggested enhancements to SPIL and EPAD are given in this section. The enhancements are listed in decreasing priority of importance; however, they are not ordered with regard to the amount of work required to implement the suggestions.

1 Have the data path and the control path connected automatically. This can be based upon the **T**erminal **AR**ray **CON**necting program (TARCON).

2 Create a SPIL algorithm simulator to verify the SPIL program. One might use the simulator N.2 [36] or have a translator which translates the SPIL program into Standard Pascal. The translator would be the easiest to implement.

3 Parameterize the SPIL data path cell library using the procedural layout languages ICEWATER or IGLOO. This would allow the designer to give power, area and delay specifications of the cells and provide an opportunity to change power lines from polysilicon into metal.

4 The speed of the control path should be increased. The critical path analysis indicates that the control path constitutes about 85 percent of the delay. One method of speeding up the control path is to use additional pipelining, such as putting a pipeline register after the AND plane of the PLA. This has the advantages of not causing a drastic change to the SPIL architecture and of cutting the delay of the PLA in half. Currently, SPIL is only a two-stage pipelined architecture; three stages might provide a significant speed up. This change could be coupled with the SPIL compiler optimizing branch instructions, as previously discussed.

5 PLAmate should label the signals in the output CIF file. This makes interfacing to CAD tools such as SILOS more efficient.

6 Some testability features must be added to the compiler [33]. The most obvious addition would seem to be scannable FSM latches. Furthermore, redundancies in the SPIL output should be removed to makes test pattern generation easier. Currently, these are the redundancies:

a The cell for most significant bit of the left shifter has redundant transistors which conditionally discharge an imaginary data bus.

b The destination decoder cells which are clocked by PHI2 contain unused inverters.

c Unconnected cells of input and output registers contain unused circuitry. Note that these unconnected cells sometimes cannot simply have their transistors removed; they may have to be replaced by cells which set values on the data bus. An example of cells which can have their transistors removed are bits 7 to 1 of the receiver variable `ADM_IN`.

d Unused bits of storage registers have extra circuitry. This is similar to the input or output cells. An example of this is the receiver variable `Ex_of_k`, bits 7 to 2.

7 Allow SPIL to specify *multiclock computational units* to reduce clock period and allow some computations over multiple clock periods. However, this only becomes practical after the speed of the control path (currently PLAmate's FSM) gets increased by at least a factor of two.

8 Investigate having address decoders made of alternating NAND and NOR gates instead of having a NAND gate and an inverter in every address decoder. This would be relatively easy to implement, compared to a customized address decoder. But, a customized address decoder would take less area and run faster. Address decoders might not even be necessary if the control path generated horizontal addresses instead of vertical ones. Currently address decoders are in the critical path.

9 Add more computational units, such as AND, OR, NAND, NOR, XOR and XNOR, to the SPIL architecture.

10 Add bidirectional off-chip latches, multiplexed chip inputs and outputs.

11 Add indirection in SPIL by adding a feature to PLAmate. Indirection could also be implemented by adding new SPIL cells which feed a value from the data bus to the address decoders.

12 The area required by off-chip lines from an output register can be reduced by having half the lines connect up and the other half connect down.

EPAD should be expanded to handle a CMOS technology which has two layers of metal. Interconnection modelling should also be further investigated, with consideration of extracting Penfield-Rubinstein-type data [37]. This modelling could be used to account for the effect of polysilicon in the layout. Temperature effects on mobilities and threshold voltages should be considered and

could be included by formulas similar to those used in SPICE [26,38].

The designer interface to EPAD could be improved to provide automatic summation of delays along circuit paths to provide a critical path analysis.

The feedback loop in SPIL should be completed. This would involve the EPAD estimates of power dissipation, area and delay, which can be analysed to obtain a critical path. This critical path information would be used to feedback changes to a parameterized SPIL library.

CHAPTER 6

Conclusions

The successful design, verification, fabrication and testing of a chip to process speech signals demonstrates that using SPIL and EPAD is an efficient design style. SPIL increases design automation in comparison to lower-level CAD tools. Since all SPIL layouts are very similar, the performance of the layout is consistent from one design to the next. Furthermore, calibrating EPAD using *kcapseries* allows the performance of future SPIL designs to be accurately predicted by EPAD and SILOS. The turn-around time of a SPIL design is less than low-level design tools because SPIL manages of all the intermediate levels of detail. With fewer details dependent on the designer, the probability that the design is correct is higher. SPIL designers do not require as much technology or circuit design experience. However, they are required to become familiar with the SPIL design environment; gaining familiarity requires approximately two months. With a design as complex as the receiver, an experienced SPIL designer would require approximately three weeks of design iterations to achieve a final layout. The predictability of the SPIL output, combined with the performance estimation from EPAD, allow the designer to determine if the generated chip meet the desired specifications. Thus, EPAD allows a design to meet a

chip's specifications. The cost of using SPIL is lower chip performance in terms of area and speed. However, the speed is increased by having covert concurrency present in the architecture. Thus, SPIL can be used to design chips for low-speed applications, such as speech processing. Since these applications will continue to exist, silicon compilers such as SPIL will become more prevalent as VLSI fabrication technology continues to improve.

Appendix A

SPIL Codec Files

Although a more detailed description of the meanings of these files may be found in the SPIL User's Manual [16], a brief description of the files will be included here for clarity.

Receiver Source File (rx.sp)

This file is the input to the SPIL program. The designer describes the chip's algorithm in this program. This file has been extensively discussed in section 4.2 (Algorithm Design).

```
PROGRAM ADM_to_PCM ;

CONST
   _data_width = 8 ;

VAR
    ADM_input  : input_port  CONNECT 0..0   UPWARD ;
    PCM_output : output_port CONNECT      DOWNWARD ;

    Ex      ,        { Starting from LSB(0) : Ex(k-1), Ex(k-2) }
    Sx_of_k ,                                  { Predicted PCM }
    X_of_k  : integer ;                       { Last PCM output }

PROCEDURE _reset ;             { Chip initialization procedure }
BEGIN
    Ex      := 0 ;               { Make Ex(k-1) = Ex(k-2) = 0 }
    X_of_k  := 0 ;
    Sx_of_k := 0 ;
END ;

BEGIN
```

```
   IF Sx_of_k < 0    THEN Sx_of_k := 0 - Sx_of_k ;

   IF Ex = ???????0B THEN Sx_of_k := 0 - Sx_of_k ;

   _add_in_1 := Sx_of_k ;          { IF Ex(k-2) = 1 THEN       }
   IF Ex = ??????1?B THEN          {                           }
      _add_in_2 :=  1              {    Sx_of_k := Sx_of_k + 1 }
   ELSE                            { ELSE                      }
      _add_in_2 := -1 ;            {    Sx_of_k := Sx_of_k - 1;}
   Sx_of_k := _add_out ;           { (No over/under-flow check)}

   X_of_k := X_of_k + Sx_of_k ;    {  No over/under-flow check }

   Ex := Ex << 1 ;   { Shift signals left   i.e. one time step }
   IF ADM_input = ???????1B THEN Ex := Ex + 1 ;

   PCM_output := X_of_k ;
END.         { ADM_input must have remained valid all the time }
```

Receiver Source File Listing (rx.spil_list)

This file is the listing output from the SPIL compiler after the compilation of the SPIL program (file rx.sp). The first part of this file, having lines numbered 1 to 39, is the same as the receiver source file (rx.sp). The next part of this file begins with `** Program Graph **`. This part indicates the width of the three buses. The width of the data bus is specified by the designer. The source and destination buses are determined by SPIL to be each four bits wide. That means there can only be up to 16 (2^4) source units and 16 destination units in the data path. This part of the file also contains a description of the states of the receiver; the states are numbered `0` to `32`. Consider state `0`. It indicates a destination unit numbered `10` and a source unit numbered `11`. The meanings of these numbers are shown in columns in the last part of this file, `** Bus Map **`.

Thus, state 0 performs a data transfer from constant 0 to register Ex. As indicated just above state 0, this data transfer implements the high-level statement numbered 16 in the first part of this file. The Moore_mask indicates what signals are the outputs of a given state. The left bit is the READY signal. The next bit is the least significant bit of the destination bus. The next bit is the least significant bit of the source bus, and so on. The source and destination buses are interleaved. State 0 indicates to transfer (Arc) to state 1, unconditionally. An example of a conditional test is shown in state 3. The construct ?1???????? refers to the matching condition of the FSM conditional test. The left signal is RESET. The next signal is GO. This is followed by the most to least significant bits of the data bus. Question marks in the condition represent *don't care* bits. Thus, the condition in state 3 means go to state 4 if the GO input to the FSM is set high. The last part of the file begins with ** Bus Map **. This part contains six columns of attributes. Each row describes one device in the data path. The first two columns contain source and destination address numbers. The third column describes the name of the cells used by Busgen to generate the layout. The sixth column contains the name of the device that can be used in the designer's program (rx.sp). The fourth and fifth columns sometimes contain additional information. For example, row 12 describes the variable ADM_IN and the fourth and fifth columns indicate the bit range that is connected off chip. This bit range was specified in the designer's program (rx.sp). Destination unit

7 is shown as -7; this negative sign means the off-chip outputs are connected at the bottom of the data path.

```
Finite-state-control description in file: rx.fsm
Bus map in file: rx.bm

 1  PROGRAM ADM_to_PCM ;
 2
 3  CONST
 4     _data_width = 8 ;
 5
 6  VAR
 7      ADM_input  : input_port  CONNECT 0..0   UPWARD ;
 8      PCM_output : output_port CONNECT      DOWNWARD ;
 9
10      Ex      ,        { Starting from LSB(0) : Ex(k-1), Ex(k-2) }
11      Sx_of_k ,                                  { Predicted PCM }
12      X_of_k  : integer ;                       { Last PCM output }
13
14  PROCEDURE _reset ;             { Chip initialization procedure }
15  BEGIN
16      Ex      := 0 ;               { Make Ex(k-1) = Ex(k-2) = 0 }
17      X_of_k  := 0 ;
18      Sx_of_k := 0 ;
19  END ;
20
21  BEGIN
22     IF Sx_of_k < 0    THEN Sx_of_k := 0 - Sx_of_k ;
23
24     IF Ex = ???????0B THEN Sx_of_k := 0 - Sx_of_k ;
25
26     _add_in_1 := Sx_of_k ;          { IF Ex(k-2) = 1 THEN       }
27     IF Ex = ??????1?B THEN          {                           }
28        _add_in_2 :=  1              {    Sx_of_k := Sx_of_k + 1 }
29     ELSE                            { ELSE                      }
30        _add_in_2 := -1 ;            {    Sx_of_k := Sx_of_k - 1;}
31     Sx_of_k := _add_out ;           { (No over/under-flow check)}
32
33     X_of_k := X_of_k + Sx_of_k ;    {  No over/under-flow check }
34
35     Ex := Ex << 1 ;   { Shift signals left   i.e. one time step }
36     IF ADM_input = ???????1B THEN Ex := Ex + 1 ;
37
38     PCM_output := X_of_k ;
39  END.         { ADM_input must have remained valid all the time }

** Program Graph **

  Data bus width =    8
```

```
     Dest addr width =   4
     Source addr width = 4

--- Source Line 16 ---
0) dest = 10,  source = 11 (const: 0),  Moore_mask: 001110011
     Arc to: #1  on condition:  default

--- Source Line 17 ---
1) dest = 8,  source = 11 (const: 0),  Moore_mask: 001010011
     Arc to: #2  on condition:  default

--- Source Line 18 ---
2) dest = 9,  source = 11 (const: 0),  Moore_mask: 011010011
     Arc to: #3  on condition:  default

--- Source Line 1 ---
3) dest = 0,  source = 0,  Moore_mask: 100000000
     Arc to: #4  on condition: ?1????????
     Arc to: #3  on condition:  default

--- Source Line 22 ---
4) dest = 0,  source = 9,  Moore_mask: 001000001
     Arc to: #5  on condition:  default

5) dest = 0,  source = 9,  Moore_mask: 001000001
     Arc to: #6  on condition: ??1???????
     Arc to: #10  on condition:  default

6) dest = 5 (cu),  source = 9,  Moore_mask: 011001001
     Arc to: #7  on condition:  default

7) dest = 1 (cu),  source = 12 (const: 1),  Moore_mask: 010000101
     Arc to: #8  on condition:  default

8) dest = 2 (cu),  source = 5 (cu),  Moore_mask: 001100100
     Arc to: #9  on condition:  default

9) dest = 9,  source = 2 (cu),  Moore_mask: 010010010
     Arc to: #10  on condition:  default

--- Source Line 24 ---
10) dest = 0,  source = 10,  Moore_mask: 000010001
     Arc to: #11  on condition:  default

11) dest = 0,  source = 10,  Moore_mask: 000010001
     Arc to: #12  on condition: ??????????0
     Arc to: #16  on condition:  default

12) dest = 5 (cu),  source = 9,  Moore_mask: 011001001
     Arc to: #13  on condition:  default

13) dest = 1 (cu),  source = 12 (const: 1),  Moore_mask: 010000101
     Arc to: #14  on condition:  default
```

```
14) dest = 2 (cu),  source = 5 (cu),  Moore_mask: 001100100
     Arc to: #15  on condition:  default

15) dest = 9,  source = 2 (cu),  Moore_mask: 010010010
     Arc to: #16  on condition:  default

--- Source Line 26 ---
16) dest = 1,  source = 9,  Moore_mask: 011000001
     Arc to: #17  on condition:  default

--- Source Line 27 ---
17) dest = 0,  source = 10,  Moore_mask: 000010001
     Arc to: #18  on condition:  default

18) dest = 0,  source = 10,  Moore_mask: 000010001
     Arc to: #19  on condition: ????????1?
     Arc to: #20  on condition:  default

--- Source Line 29 ---
19) dest = 2,  source = 12 (const: 1),  Moore_mask: 000100101
     Arc to: #21  on condition:  default

--- Source Line 30 ---
20) dest = 2,  source = 13 (const: -1),  Moore_mask: 001100101
     Arc to: #21  on condition:  default

--- Source Line 31 ---
21) dest = 9,  source = 2,  Moore_mask: 010010010
     Arc to: #22  on condition:  default

--- Source Line 33 ---
22) dest = 1 (cu),  source = 8,  Moore_mask: 010000001
     Arc to: #23  on condition:  default

23) dest = 2 (cu),  source = 9,  Moore_mask: 001100001
     Arc to: #24  on condition:  default

24) dest = 8,  source = 2 (cu),  Moore_mask: 000010010
     Arc to: #25  on condition:  default

--- Source Line 35 ---
25) dest = 3 (cu),  source = 10,  Moore_mask: 010110001
     Arc to: #26  on condition:  default

26) dest = 10,  source = 3 (cu),  Moore_mask: 001110010
     Arc to: #27  on condition:  default

--- Source Line 36 ---
27) dest = 0,  source = 6,  Moore_mask: 000010100
     Arc to: #28  on condition:  default

28) dest = 0,  source = 6,  Moore_mask: 000010100
     Arc to: #29  on condition: ??????????1
     Arc to: #32  on condition:  default
```

```
29) dest = 1 (cu),  source = 10,  Moore_mask: 010010001
     Arc to: #30  on condition:  default

30) dest = 2 (cu),  source = 12 (const: 1),  Moore_mask: 000100101
     Arc to: #31  on condition:  default

31) dest = 10,  source = 2 (cu),  Moore_mask: 000110010
     Arc to: #32  on condition:  default

--- Source Line 38 ---
32) dest = 7,  source = 8,  Moore_mask: 010101001
     Arc to: #3  on condition:  default
```

** Bus Map **

dest addr	source addr	device.	info1	info2	symbol
0	0	data_prech	*	*	*
0	2	add_out	*	*	_ADD_OUT
0	0	adder	*	*	*
2	0	add_latch_b	*	*	_ADD_IN_2
1	0	add_latch_a	*	*	_ADD_IN_1
0	3	shift_left	*	*	_SHFL_OUT
3	0	dlatch	*	*	_SHFL_IN
0	4	shift_right	*	*	_SHFR_OUT
4	0	dlatch	*	*	_SHFR_IN
0	5	compl	*	*	_COMPL_OUT
5	0	dlatch	*	*	_COMPL_IN
0	6	oc_in_latch	0	0	ADM_INPUT
-7	0	oc_out_latch	0	7	PCM_OUTPUT
0	8	out_enable	*	*	X_OF_K
8	0	dlatch	*	*	"
0	9	out_enable	*	*	SX_OF_K
9	0	dlatch	*	*	"
0	10	out_enable	*	*	EX
10	0	dlatch	*	*	"
0	11	const	0	*	*
0	12	const	1	*	*
0	13	const	-1	*	*

Receiver Busgen File (rx.bm)

This file was generated by SPIL. It is a description of devices in the data path. Each line of the file describes one or more columns of items in the data path. The first line indicates the number of bits in the data bus, followed on the same line by the number of bits in the source and destination address buses. The remaining lines map on a one-to-one basis with the last part (** Bus Map **) of the listing file (rx.spil_list), excluding the symbol column in the listing file. This file could be input to Busgen. The actual file which was given as input to Busgen is shown in the next section.

```
8    4    4
   0        0     9          0     0
   0        2     2          0     0
   0        0     3          0     0
   2        0     1          0     0
   1        0     0          0     0
   0        3    23          0     0
   3        0    10          0     0
   0        4    25          0     0
   4        0    10          0     0
   0        5     5          0     0
   5        0    10          0     0
   0        6    11          0     0
  -7        0    12          0     7
   0        8    22          0     0
   8        0    10          0     0
   0        9    22          0     0
   9        0    10          0     0
   0       10    22          0     0
  10        0    10          0     0
   0       11     6          0     0
   0       12     6          1     0
   0       13     6         -1     0
```

Receiver Busgen File (rx_no_right_shifter.bm)

This is file is the same as rx.bm except that the lines

```
  0        4    25          0     0
  4        0    10          0     0
```

which describe the input and output ports of the right shifter and the right shifter have been removed. The file shown here had the right shifter removed because it is not used in the receiver algorithm. The version of SPIL that was used did not automatically remove unused devices.

```
8    4    4
   0        0     9          0     0
   0        2     2          0     0
   0        0     3          0     0
   2        0     1          0     0
   1        0     0          0     0
   0        3    23          0     0
   3        0    10          0     0
   0        5     5          0     0
   5        0    10          0     0
   0        6    11          0     0
  -7        0    12          0     7
   0        8    22          0     0
   8        0    10          0     0
   0        9    22          0     0
   9        0    10          0     0
   0       10    22          0     0
  10        0    10          0     0
   0       11     6          0     0
   0       12     6          1     0
   0       13     6         -1     0
```

Receiver FSM File (rx.fsm)

This file was generated by SPIL for input to PLAmate. This file's syntax is acceptable to PLAmate; it's semantics are similar to the state-by-state description of the FSM given in the SPIL listing file (rx.spil_list). The second line of this file (rx.fsm) shows the inputs to the FSM. Signal x00 is the RESET signal. Signal x01 is the GO signal. Signal x02 is the least significant bit of the data bus and so on up to the most significant bit of the data bus, x09. The third line shows the outputs. Signals y00 to y08 correspond exactly to the Moore mask in the SPIL listing file (rx.spil_list). This is followed by a list of the symbolic states, such as s000, and the corresponding actual FSM state numbers, such as 0. The line beginning with RESET indicates that when signal x00 is raised high, the FSM should enter state s000, the reset state. The RESET line is followed by the state description. Consider state s005. It indicates two output signals, y02, y08, which are set high during state s005. If input x02 is high, then the next FSM state will be s008, otherwise s010 will be the next FSM state.

```
FSM ADM_TO_PCM;
    INPUTS  x00, x01, x02, x03, x04, x05, x06, x07, x08, x09;
    OUTPUTS  y00, y01, y02, y03, y04, y05, y06, y07, y08;

    STATES
        s000 = 0,
        s001 = 1,
        s002 = 2,
        s003 = 3,
        s004 = 4,
        s005 = 5,
        s006 = 6,
```

```
    s007 = 7,
    s008 = 8,
    s009 = 9,
    s010 = 10,
    s011 = 11,
    s012 = 12,
    s013 = 13,
    s014 = 14,
    s015 = 15,
    s016 = 16,
    s017 = 17,
    s018 = 18,
    s019 = 19,
    s020 = 20,
    s021 = 21,
    s022 = 22,
    s023 = 23,
    s024 = 24,
    s025 = 25,
    s026 = 26,
    s027 = 27,
    s028 = 28,
    s029 = 29,
    s030 = 30,
    s031 = 31,
    s032 = 32;

RESET  x00 : s000;

STATE  s000  > y02, y03, y04, y07, y08;
    OTHERWISE  : s001;

STATE  s001  > y02, y04, y07, y08;
    OTHERWISE  : s002;

STATE  s002  > y01, y02, y04, y07, y08;
    OTHERWISE  : s003;

STATE  s003  > y00;
    x01  : s004;
    OTHERWISE  : s003;

STATE  s004  > y02, y08;
    OTHERWISE  : s005;

STATE  s005  > y02, y08;
    x02  : s006;
    OTHERWISE  : s010;

STATE  s006  > y01, y02, y05, y08;
    OTHERWISE  : s007;

STATE  s007  > y01, y06, y08;
    OTHERWISE  : s008;
```

```
STATE  s008  > y02, y03, y06;
    OTHERWISE  : s009;

STATE  s009  > y01, y04, y07;
    OTHERWISE  : s010;

STATE  s010  > y04, y08;
    OTHERWISE  : s011;

STATE  s011  > y04, y08;
    x09'  : s012;
    OTHERWISE  : s016;

STATE  s012  > y01, y02, y05, y08;
    OTHERWISE  : s013;

STATE  s013  > y01, y06, y08;
    OTHERWISE  : s014;

STATE  s014  > y02, y03, y06;
    OTHERWISE  : s015;

STATE  s015  > y01, y04, y07;
    OTHERWISE  : s016;

STATE  s016  > y01, y02, y08;
    OTHERWISE  : s017;

STATE  s017  > y04, y08;
    OTHERWISE  : s018;

STATE  s018  > y04, y08;
    x08  : s019;
    OTHERWISE  : s020;

STATE  s019  > y03, y06, y08;
    OTHERWISE  : s021;

STATE  s020  > y02, y03, y06, y08;
    OTHERWISE  : s021;

STATE  s021  > y01, y04, y07;
    OTHERWISE  : s022;

STATE  s022  > y01, y08;
    OTHERWISE  : s023;

STATE  s023  > y02, y03, y08;
    OTHERWISE  : s024;

STATE  s024  > y04, y07;
    OTHERWISE  : s025;

STATE  s025  > y01, y03, y04, y08;
```

```
        OTHERWISE  : s026;

    STATE  s026  > y02, y03, y04, y07;
        OTHERWISE  : s027;

    STATE  s027  > y04, y06;
        OTHERWISE  : s028;

    STATE  s028  > y04, y06;
        x09  : s029;
        OTHERWISE  : s032;

    STATE  s029  > y01, y04, y08;
        OTHERWISE  : s030;

    STATE  s030  > y03, y06, y08;
        OTHERWISE  : s031;

    STATE  s031  > y03, y04, y07;
        OTHERWISE  : s032;

    STATE  s032  > y01, y03, y05, y08;
        OTHERWISE  : s003;

END.
```

Receiver Busgen Listing File (rx.bm_list)

This file is the listing file from Busgen and simply indicates the size of the data path. The units of microns which are shown are design scale microns [18,19].

```
Dimensions of device array:
Height =  2721 microns
Width  =  2839 microns
```

Receiver FSM Listing File (rx.fsm_list)

This file is the listing output of PLAmate. It contains a description of the AND as well as the OR planes of the PLA. The rows of the AND plane represent the FSM inputs x00 to x09 followed by the six state lines from the least to the most significant bits. The rows extended beyond 80 characters per line and wrap around to the beginning of the line so that one row of the AND plane spans two rows of text. The rows are numbered just before the character |. The numbers 1 to 16 correspond to the signals by the order that the rows of the AND plane were described above. If a negative number is shown, it refers to the inverted version of the signal. The columns of the AND plane represent the minterms. In the OR plane output, the rows represent the minterms and the columns represent the FSM outputs. The order of the FSM outputs from left to right is y08 to y00 and then the least significant bit to the most significant bit. Above each of the AND and OR plane columns is the number of ones in each column. The end of this file indicates the size of the FSM, even though the word PLA is shown. Units of size are design scale microns.

```
---    P L A M A T E     -    University of Waterloo PLA/FSM generator    ---
                PLA/FSM "ADM_TO_PCM" generated on Tue Dec  2 18:48:52 1986


SYMBOLS:
    INPUT   1 = x00
```

```
INPUT   2 = x01
INPUT   3 = x02
INPUT   4 = x03
INPUT   5 = x04
INPUT   6 = x05
INPUT   7 = x06
INPUT   8 = x07
INPUT   9 = x08
INPUT  10 = x09

OUTPUT  1 = y00
OUTPUT  2 = y01
OUTPUT  3 = y02
OUTPUT  4 = y03
OUTPUT  5 = y04
OUTPUT  6 = y05
OUTPUT  7 = y06
OUTPUT  8 = y07
OUTPUT  9 = y08

STATE   0 = s000        (explicit)
STATE   1 = s001        (explicit)
STATE   2 = s002        (explicit)
STATE   3 = s003        (explicit)
STATE   4 = s004        (explicit)
STATE   5 = s005        (explicit)
STATE   6 = s006        (explicit)
STATE   7 = s007        (explicit)
STATE   8 = s008        (explicit)
STATE   9 = s009        (explicit)
STATE  10 = s010        (explicit)
STATE  11 = s011        (explicit)
STATE  12 = s012        (explicit)
STATE  13 = s013        (explicit)
STATE  14 = s014        (explicit)
STATE  15 = s015        (explicit)
STATE  16 = s016        (explicit)
STATE  17 = s017        (explicit)
STATE  18 = s018        (explicit)
STATE  19 = s019        (explicit)
STATE  20 = s020        (explicit)
STATE  21 = s021        (explicit)
STATE  22 = s022        (explicit)
STATE  23 = s023        (explicit)
STATE  24 = s024        (explicit)
STATE  25 = s025        (explicit)
STATE  26 = s026        (explicit)
STATE  27 = s027        (explicit)
STATE  28 = s028        (explicit)
STATE  29 = s029        (explicit)
STATE  30 = s030        (explicit)
STATE  31 = s031        (explicit)
STATE  32 = s032        (explicit)
```

Before minimization:

Number of inputs is 10.
Number of outputs is 9.
Number of state lines is 6.
Number of minterms is 39.

There were 0 errors in this description.

After minimization, number of minterms is 38.

THE AND PLANE:

```
       8  8  8  8  7  7  7  7  7  8  7  8  7  7  7  8  7  7  6  7  7  7  7  7  7
  7  8  7  8  7  7  6  6  7  7  7  7  8
     ------------------------------------------------------------------------------
-------------------------------------
 -1 |  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  1  1  1  1  1  1
 -2 |  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0
  2 |  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0
 -3 |  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0
  3 |  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0
 -9 |  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  1  0  0  0  0  0  0  0  0  0  0  0
  9 |  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  1
-10 |  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  1  0  0  0  0  0  0  0  0  0
 10 |  0  0  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0
-11 |  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  1  1  1
  1  1  1  1  1  1  1  1  1  1  1  1  1
 11 |  1  1  1  0  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0
-12 |  0  1  0  1  0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1  0  0  0  0  0
  0  0  1  1  1  0  0  1  1  1  1  0
 12 |  1  0  1  0  1  1  1  1  1  1  1  1  0  0  0  0  0  0  0  0  1  1  1  1  1
  1  1  1  0  0  0  0  0  0  0  0  0  1
-13 |  1  0  1  0  0  0  0  0  1  1  1  1  0  0  0  0  1  1  1  1  0  0  0  0  1
  1  1  1  0  0  0  0  1  1  1  1  1  1
 13 |  0  1  0  1  1  1  1  1  0  0  0  0  1  1  1  1  0  0  0  0  1  1  1  1  0
  0  0  1  1  1  1  0  0  0  0  0  0
-14 |  1  1  0  0  0  0  1  1  0  0  1  1  0  0  1  1  0  0  0  1  0  0  1  1  0
  0  1  1  0  0  1  1  0  0  1  1  1  1
 14 |  0  0  1  1  1  1  0  0  1  1  0  0  1  1  0  0  1  1  0  0  1  1  0  0  1
  1  0  0  1  1  0  0  1  1  0  0  0  0
-15 |  1  1  1  0  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0
```

```
  1  0  1  0  1  0  1  0  1  0  1  1  0
 15 |  0  0  0  1  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1
  0  1  0  1  0  1  0  1  0  1  0  0  1
-16 |  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  1  1  1  0  1  1
 16 |  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  1  0  0
```

THE OR PLANE:

	2	13	14	11	20	3	10	9	25	19	19	17	17	18	2
1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0
2	0	0	1	0	0	0	0	0	1	0	1	1	0	0	0
3	0	0	0	0	1	0	0	0	1	0	0	0	0	1	0
4	0	0	0	0	1	0	1	0	0	1	0	1	1	1	0
5	0	0	0	1	1	0	0	1	0	0	0	0	0	0	1
6	0	1	0	0	1	0	0	1	0	0	0	0	0	1	0
7	0	0	1	1	0	0	0	0	1	0	0	0	1	1	0
8	0	1	0	0	0	0	1	0	1	0	0	0	1	0	0
9	0	0	0	0	1	0	1	0	0	0	0	1	1	1	0
10	0	0	0	0	1	0	0	0	1	0	0	1	1	0	0
11	0	0	0	1	0	0	1	0	1	1	0	1	0	1	0
12	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0
13	0	1	0	0	1	0	0	0	1	0	1	1	1	1	0
14	0	1	0	0	0	0	1	0	1	0	1	1	1	0	0
15	0	1	0	0	1	0	0	1	0	0	1	1	0	1	0
16	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0
17	0	1	0	1	1	0	0	0	1	0	1	0	1	1	0
18	0	1	0	0	1	0	0	1	0	0	1	0	1	0	0
19	0	0	0	0	1	0	0	0	1	0	1	0	0	1	0
20	0	0	1	0	1	0	0	1	1	0	1	0	0	0	0
21	0	0	0	1	0	0	1	0	1	1	1	1	1	1	0
22	0	0	1	1	0	0	1	0	0	1	1	1	1	0	0
23	0	1	0	0	0	0	0	0	1	1	1	1	0	1	0
24	0	1	1	0	0	1	0	0	1	1	1	1	0	0	0
25	0	0	1	1	1	0	0	1	0	1	1	0	1	1	0
26	0	0	0	0	1	0	0	0	1	1	1	0	1	0	0
27	0	0	0	0	1	0	0	0	1	0	0	1	0	1	0
28	0	1	1	0	1	0	0	1	1	1	1	0	0	0	0
29	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1
30	0	1	1	0	0	1	0	0	1	1	0	1	1	0	0
31	0	0	1	1	0	0	1	0	1	1	0	1	0	1	0
32	0	0	1	0	0	0	0	0	1	1	0	1	0	0	0
33	0	0	0	0	1	0	0	1	0	1	0	0	1	1	0
34	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0
35	0	1	1	0	0	0	0	0	1	1	0	0	0	1	0
36	0	1	0	1	0	1	0	0	1	1	1	0	0	0	0
37	0	0	1	1	1	0	0	1	1	1	0	0	0	0	0
38	0	0	0	0	1	0	0	0	1	1	1	0	0	1	0

PLA HEIGHT: 2433 Microns

```
PLA WIDTH:   3137 Microns
```

Transmitter Source File (tx.sp)

The transmitter files have meanings similar to those of the receiver files. A description will not be repeated.

```
PROGRAM PCM_to_ADM ;

CONST
   _data_width = 8 ;

VAR
    PCM_input  : input_port  CONNECT      DOWNWARD ;
    ADM_output : output_port CONNECT 0..0   UPWARD ;

    Ex      ,        { Starting from LSB(0) : Ex(k-1), Ex(k-2) }
    Sx_of_k ,                    { Step to next predicted PCM }
    X_of_k  : integer ;                     { Last PCM output }

PROCEDURE _reset ;             { Chip initialization procedure }
BEGIN
    Ex      := 0 ;                { Make Ex(k-1) = Ex(k-2) = 0 }
    X_of_k  := 0 ;
    Sx_of_k := 0 ;
END ;

BEGIN
   IF Sx_of_k < 0    THEN Sx_of_k := 0 - Sx_of_k ;

   IF Ex = ???????0B THEN Sx_of_k := 0 - Sx_of_k ;

   _add_in_1 := Sx_of_k ;          { IF Ex(k-2) = 1 THEN       }
   IF Ex = ??????1?B THEN          {                           }
      _add_in_2 :=  1              {    Sx_of_k := Sx_of_k + 1 }
   ELSE                            { ELSE                      }
      _add_in_2 := -1 ;            {    Sx_of_k := Sx_of_k - 1;}
   Sx_of_k := _add_out ;           { (No over/under-flow check)}

   X_of_k := X_of_k + Sx_of_k ;    {  No over/under-flow check }

   Ex := Ex << 1 ;   { Shift signals left   i.e. one time step }
   IF PCM_input > X_of_k THEN Ex := Ex + 1 ;

   ADM_output := Ex ;
END.         { PCM_input must have remained valid all the time }
```

Transmitter Source File Listing (tx.spil_list)

```
Finite-state-control description in file: tx.fsm
Bus map in file: tx.bm

     1  PROGRAM PCM_to_ADM ;
     2
     3  CONST
     4     _data_width = 8 ;
     5
     6  VAR
     7      PCM_input  : input_port  CONNECT      DOWNWARD ;
     8      ADM_output : output_port CONNECT 0..0   UPWARD ;
     9
    10      Ex      ,        { Starting from LSB(0) : Ex(k-1), Ex(k-2) }
    11      Sx_of_k ,                    { Step to next predicted PCM }
    12      X_of_k  : integer ;                    { Last PCM output }
    13
    14  PROCEDURE _reset ;             { Chip initialization procedure }
    15  BEGIN
    16      Ex      := 0 ;               { Make Ex(k-1) = Ex(k-2) = 0 }
    17      X_of_k  := 0 ;
    18      Sx_of_k := 0 ;
    19  END ;
    20
    21  BEGIN
    22     IF Sx_of_k < 0    THEN Sx_of_k := 0 - Sx_of_k ;
    23
    24     IF Ex = ???????0B THEN Sx_of_k := 0 - Sx_of_k ;
    25
    26     _add_in_1 := Sx_of_k ;          { IF Ex(k-2) = 1 THEN       }
    27     IF Ex = ??????1?B THEN          {                           }
    28        _add_in_2 :=  1              {    Sx_of_k := Sx_of_k + 1 }
    29     ELSE                            { ELSE                      }
    30        _add_in_2 := -1 ;            {    Sx_of_k := Sx_of_k - 1;}
    31     Sx_of_k := _add_out ;           { (No over/under-flow check)}
    32
    33     X_of_k := X_of_k + Sx_of_k ;    {  No over/under-flow check }
    34
    35     Ex := Ex << 1 ;   { Shift signals left   i.e. one time step }
    36     IF PCM_input > X_of_k THEN Ex := Ex + 1 ;
    37
    38     ADM_output := Ex ;
    39  END.         { PCM_input must have remained valid all the time }

  ** Program Graph **

     Data bus width =    8
     Dest addr width =   4
     Source addr width = 4
```

```
--- Source Line 16 ---
0) dest = 10,  source = 11 (const: 0),  Moore_mask: 001110011
     Arc to: #1  on condition:  default

--- Source Line 17 ---
1) dest = 8,  source = 11 (const: 0),  Moore_mask: 001010011
     Arc to: #2  on condition:  default

--- Source Line 18 ---
2) dest = 9,  source = 11 (const: 0),  Moore_mask: 011010011
     Arc to: #3  on condition:  default

--- Source Line 1 ---
3) dest = 0,  source = 0,  Moore_mask: 100000000
     Arc to: #4  on condition: ?1????????
     Arc to: #3  on condition:  default

--- Source Line 22 ---
4) dest = 0,  source = 9,  Moore_mask: 001000001
     Arc to: #5  on condition:  default

5) dest = 0,  source = 9,  Moore_mask: 001000001
     Arc to: #6  on condition: ??1???????
     Arc to: #10  on condition:  default

6) dest = 5 (cu),  source = 9,  Moore_mask: 011001001
     Arc to: #7  on condition:  default

7) dest = 1 (cu),  source = 12 (const: 1),  Moore_mask: 010000101
     Arc to: #8  on condition:  default

8) dest = 2 (cu),  source = 5 (cu),  Moore_mask: 001100100
     Arc to: #9  on condition:  default

9) dest = 9,  source = 2 (cu),  Moore_mask: 010010010
     Arc to: #10  on condition:  default

--- Source Line 24 ---
10) dest = 0,  source = 10,  Moore_mask: 000010001
     Arc to: #11  on condition:  default

11) dest = 0,  source = 10,  Moore_mask: 000010001
     Arc to: #12  on condition: ??????????0
     Arc to: #16  on condition:  default

12) dest = 5 (cu),  source = 9,  Moore_mask: 011001001
     Arc to: #13  on condition:  default

13) dest = 1 (cu),  source = 12 (const: 1),  Moore_mask: 010000101
     Arc to: #14  on condition:  default

14) dest = 2 (cu),  source = 5 (cu),  Moore_mask: 001100100
     Arc to: #15  on condition:  default
```

```
15) dest = 9,  source = 2 (cu),  Moore_mask: 010010010
     Arc to: #16  on condition:  default

--- Source Line 26 ---
16) dest = 1,  source = 9,  Moore_mask: 011000001
     Arc to: #17  on condition:  default

--- Source Line 27 ---
17) dest = 0,  source = 10,  Moore_mask: 000010001
     Arc to: #18  on condition:  default

18) dest = 0,  source = 10,  Moore_mask: 000010001
     Arc to: #19  on condition: ????????1?
     Arc to: #20  on condition:  default

--- Source Line 29 ---
19) dest = 2,  source = 12 (const: 1),  Moore_mask: 000100101
     Arc to: #21  on condition:  default

--- Source Line 30 ---
20) dest = 2,  source = 13 (const: -1),  Moore_mask: 001100101
     Arc to: #21  on condition:  default

--- Source Line 31 ---
21) dest = 9,  source = 2,  Moore_mask: 010010010
     Arc to: #22  on condition:  default

--- Source Line 33 ---
22) dest = 1 (cu),  source = 8,  Moore_mask: 010000001
     Arc to: #23  on condition:  default

23) dest = 2 (cu),  source = 9,  Moore_mask: 001100001
     Arc to: #24  on condition:  default

24) dest = 8,  source = 2 (cu),  Moore_mask: 000010010
     Arc to: #25  on condition:  default

--- Source Line 35 ---
25) dest = 3 (cu),  source = 10,  Moore_mask: 010110001
     Arc to: #26  on condition:  default

26) dest = 10,  source = 3 (cu),  Moore_mask: 001110010
     Arc to: #27  on condition:  default

--- Source Line 36 ---
27) dest = 1 (cu),  source = 13 (const: -1),  Moore_mask: 011000101
     Arc to: #28  on condition:  default

28) dest = 2 (cu),  source = 6,  Moore_mask: 000110100
     Arc to: #29  on condition:  default

29) dest = 5 (cu),  source = 2 (cu),  Moore_mask: 010011000
     Arc to: #30  on condition:  default
```

```
30) dest = 1 (cu),  source = 8,  Moore_mask: 010000001
     Arc to: #31  on condition:  default

31) dest = 2 (cu),  source = 5 (cu),  Moore_mask: 001100100
     Arc to: #32  on condition:  default

32) dest = 0,  source = 2 (cu),  Moore_mask: 000010000
     Arc to: #33  on condition:  default

33) dest = 0,  source = 2 (cu),  Moore_mask: 000010000
     Arc to: #34  on condition: ??1???????
     Arc to: #37  on condition:  default

34) dest = 1 (cu),  source = 10,  Moore_mask: 010010001
     Arc to: #35  on condition:  default

35) dest = 2 (cu),  source = 12 (const: 1),  Moore_mask: 000100101
     Arc to: #36  on condition:  default

36) dest = 10,  source = 2 (cu),  Moore_mask: 000110010
     Arc to: #37  on condition:  default

--- Source Line 38 ---
37) dest = 7,  source = 10,  Moore_mask: 010111001
     Arc to: #3  on condition:  default

           ** Bus Map **

   dest   source
   addr    addr    device.        info1     info2   symbol
   ----   ------   -------        -----     -----   ------

     0       0      data_prech      *         *     *
     0       2      add_out         *         *     _ADD_OUT
     0       0      adder           *         *     *
     2       0      add_latch_b     *         *     _ADD_IN_2
     1       0      add_latch_a     *         *     _ADD_IN_1

     0       3      shift_left      *         *     _SHFL_OUT
     3       0      dlatch          *         *     _SHFL_IN
     0       4      shift_right     *         *     _SHFR_OUT
     4       0      dlatch          *         *     _SHFR_IN
     0       5      compl           *         *     _COMPL_OUT

     5       0      dlatch          *         *     _COMPL_IN
     0      -6      oc_in_latch     0         7     PCM_INPUT
     7       0      oc_out_latch    0         0     ADM_OUTPUT
     0       8      out_enable      *         *     X_OF_K
     8       0      dlatch          *         *     "

     0       9      out_enable      *         *     SX_OF_K
     9       0      dlatch          *         *     "
     0      10      out_enable      *         *     EX
    10       0      dlatch          *         *     "
```

```
 0      11      const          0        *     *
 0      12      const          1        *     *
 0      13      const          -1       *     *
```

Transmitter Busgen File (tx.bm)

```
8    4    4
   0        0     9          0     0
   0        2     2          0     0
   0        0     3          0     0
   2        0     1          0     0
   1        0     0          0     0
   0        3    23          0     0
   3        0    10          0     0
   0        4    25          0     0
   4        0    10          0     0
   0        5     5          0     0
   5        0    10          0     0
   0       -6    11          0     7
   7        0    12          0     0
   0        8    22          0     0
   8        0    10          0     0
   0        9    22          0     0
   9        0    10          0     0
   0       10    22          0     0
  10        0    10          0     0
   0       11     6          0     0
   0       12     6          1     0
   0       13     6         -1     0
```

Transmitter Busgen File (tx_no_right_shifter.bm)

```
8    4    4
   0        0     9          0     0
   0        2     2          0     0
   0        0     3          0     0
   2        0     1          0     0
   1        0     0          0     0
   0        3    23          0     0
   3        0    10          0     0
   0        5     5          0     0
   5        0    10          0     0
   0       -6    11          0     7
   7        0    12          0     0
   0        8    22          0     0
   8        0    10          0     0
   0        9    22          0     0
   9        0    10          0     0
   0       10    22          0     0
  10        0    10          0     0
   0       11     6          0     0
   0       12     6          1     0
   0       13     6         -1     0
```

Transmitter FSM File (tx.fsm)

```
FSM PCM_TO_ADM;
    INPUTS  x00, x01, x02, x03, x04, x05, x06, x07, x08, x09;
    OUTPUTS  y00, y01, y02, y03, y04, y05, y06, y07, y08;

    STATES
        s000 = 0,
        s001 = 1,
        s002 = 2,
        s003 = 3,
        s004 = 4,
        s005 = 5,
        s006 = 6,
        s007 = 7,
        s008 = 8,
        s009 = 9,
        s010 = 10,
        s011 = 11,
        s012 = 12,
        s013 = 13,
        s014 = 14,
        s015 = 15,
        s016 = 16,
        s017 = 17,
        s018 = 18,
        s019 = 19,
        s020 = 20,
        s021 = 21,
        s022 = 22,
        s023 = 23,
        s024 = 24,
        s025 = 25,
        s026 = 26,
        s027 = 27,
        s028 = 28,
        s029 = 29,
        s030 = 30,
        s031 = 31,
        s032 = 32,
        s033 = 33,
        s034 = 34,
        s035 = 35,
        s036 = 36,
        s037 = 37;

    RESET  x00 : s000;

    STATE  s000  > y02, y03, y04, y07, y08;
        OTHERWISE  : s001;
```

```
STATE  s001  > y02, y04, y07, y08;
    OTHERWISE  : s002;

STATE  s002  > y01, y02, y04, y07, y08;
    OTHERWISE  : s003;

STATE  s003  > y00;
    x01  : s004;
    OTHERWISE  : s003;

STATE  s004  > y02, y08;
    OTHERWISE  : s005;

STATE  s005  > y02, y08;
    x02  : s006;
    OTHERWISE  : s010;

STATE  s006  > y01, y02, y05, y08;
    OTHERWISE  : s007;

STATE  s007  > y01, y06, y08;
    OTHERWISE  : s008;

STATE  s008  > y02, y03, y06;
    OTHERWISE  : s009;

STATE  s009  > y01, y04, y07;
    OTHERWISE  : s010;

STATE  s010  > y04, y08;
    OTHERWISE  : s011;

STATE  s011  > y04, y08;
    x09'  : s012;
    OTHERWISE  : s016;

STATE  s012  > y01, y02, y05, y08;
    OTHERWISE  : s013;

STATE  s013  > y01, y06, y08;
    OTHERWISE  : s014;

STATE  s014  > y02, y03, y06;
    OTHERWISE  : s015;

STATE  s015  > y01, y04, y07;
    OTHERWISE  : s016;

STATE  s016  > y01, y02, y08;
    OTHERWISE  : s017;

STATE  s017  > y04, y08;
    OTHERWISE  : s018;
```

```
STATE  s018  > y04, y08;
    x08  : s019;
    OTHERWISE  : s020;

STATE  s019  > y03, y06, y08;
    OTHERWISE  : s021;

STATE  s020  > y02, y03, y06, y08;
    OTHERWISE  : s021;

STATE  s021  > y01, y04, y07;
    OTHERWISE  : s022;

STATE  s022  > y01, y08;
    OTHERWISE  : s023;

STATE  s023  > y02, y03, y08;
    OTHERWISE  : s024;

STATE  s024  > y04, y07;
    OTHERWISE  : s025;

STATE  s025  > y01, y03, y04, y08;
    OTHERWISE  : s026;

STATE  s026  > y02, y03, y04, y07;
    OTHERWISE  : s027;

STATE  s027  > y01, y02, y06, y08;
    OTHERWISE  : s028;

STATE  s028  > y03, y04, y06;
    OTHERWISE  : s029;

STATE  s029  > y01, y04, y05;
    OTHERWISE  : s030;

STATE  s030  > y01, y08;
    OTHERWISE  : s031;

STATE  s031  > y02, y03, y06;
    OTHERWISE  : s032;

STATE  s032  > y04;
    OTHERWISE  : s033;

STATE  s033  > y04;
    x02  : s034;
    OTHERWISE  : s037;

STATE  s034  > y01, y04, y08;
    OTHERWISE  : s035;

STATE  s035  > y03, y06, y08;
```

```
        OTHERWISE  : s036;

    STATE  s036  > y03, y04, y07;
        OTHERWISE  : s037;

    STATE  s037  > y01, y03, y04, y05, y08;
        OTHERWISE  : s003;

END.
```

Transmitter Busgen Listing (tx.bm_list)

```
 Dimensions of device array:
 Height =  2721 microns
 Width  =  2839 microns
```

Transmitter FSM Listing (tx.fsm_list)

```
---    P L A M A T E     -    University of Waterloo PLA/FSM generator    ---

               PLA/FSM "PCM_TO_ADM" generated on Sat Dec 13 05:19:00 1986


SYMBOLS:

    INPUT   1 = x00
    INPUT   2 = x01
    INPUT   3 = x02
    INPUT   4 = x03
    INPUT   5 = x04
    INPUT   6 = x05
    INPUT   7 = x06
    INPUT   8 = x07
    INPUT   9 = x08
    INPUT  10 = x09

    OUTPUT  1 = y00
    OUTPUT  2 = y01
    OUTPUT  3 = y02
```

```
    OUTPUT  4 = y03
    OUTPUT  5 = y04
    OUTPUT  6 = y05
    OUTPUT  7 = y06
    OUTPUT  8 = y07
    OUTPUT  9 = y08

    STATE   0 = s000        (explicit)
    STATE   1 = s001        (explicit)
    STATE   2 = s002        (explicit)
    STATE   3 = s003        (explicit)
    STATE   4 = s004        (explicit)
    STATE   5 = s005        (explicit)
    STATE   6 = s006        (explicit)
    STATE   7 = s007        (explicit)
    STATE   8 = s008        (explicit)
    STATE   9 = s009        (explicit)
    STATE  10 = s010        (explicit)
    STATE  11 = s011        (explicit)
    STATE  12 = s012        (explicit)
    STATE  13 = s013        (explicit)
    STATE  14 = s014        (explicit)
    STATE  15 = s015        (explicit)
    STATE  16 = s016        (explicit)
    STATE  17 = s017        (explicit)
    STATE  18 = s018        (explicit)
    STATE  19 = s019        (explicit)
    STATE  20 = s020        (explicit)
    STATE  21 = s021        (explicit)
    STATE  22 = s022        (explicit)
    STATE  23 = s023        (explicit)
    STATE  24 = s024        (explicit)
    STATE  25 = s025        (explicit)
    STATE  26 = s026        (explicit)
    STATE  27 = s027        (explicit)
    STATE  28 = s028        (explicit)
    STATE  29 = s029        (explicit)
    STATE  30 = s030        (explicit)
    STATE  31 = s031        (explicit)
    STATE  32 = s032        (explicit)
    STATE  33 = s033        (explicit)
    STATE  34 = s034        (explicit)
    STATE  35 = s035        (explicit)
    STATE  36 = s036        (explicit)
    STATE  37 = s037        (explicit)


Before minimization:

    Number of inputs is      10.
    Number of outputs is      9.
    Number of state lines is  6.
    Number of minterms is    44.
```

There were 0 errors in this description.

After minimization, number of minterms is 43.

THE AND PLANE:

```
       8  8  8  8  7  7  7  7  7  8  7  7  8  7  7  7  7  8  7  7  6  8  7  7  7
  6  7  7  7  8  7  7  7  7  7  7  6  6  7  7  6  7  8
     ------------------------------------------------------------------------------
-------------------------------------------------------
 -1 |  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
 -2 |  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  2 |  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 -3 |  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  1  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  3 |  0  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 -9 |  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  9 |  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1
-10 |  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 10 |  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
-11 |  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  1
  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
 11 |  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
-12 |  0  1  1  0  0  0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1  1  1  0  0
  0  0  0  0  0  0  0  1  1  1  1  0  0  1  1  0  1  0
 12 |  1  0  0  1  1  1  1  1  1  1  1  1  1  1  0  0  0  0  0  0  0  0  0  1  1
  1  1  1  1  1  1  1  0  0  0  0  0  0  0  0  0  0  1
-13 |  1  0  1  1  0  0  0  0  1  1  1  1  1  0  0  0  0  0  1  1  1  1  1  0  0
  0  0  1  1  1  1  1  0  0  0  0  0  1  1  1  1  1  1
 13 |  0  1  0  0  1  1  1  1  0  0  0  0  0  1  1  1  1  1  0  0  0  0  0  1  1
  1  1  0  0  0  0  0  1  1  1  1  1  0  0  0  0  0  0
-14 |  1  1  1  0  0  0  1  1  0  0  1  1  1  0  0  1  1  1  0  0  0  1  1  0  0
  0  1  0  0  1  1  1  0  0  1  1  1  0  0  1  1  1  1
 14 |  0  0  0  1  1  1  0  0  1  1  0  0  0  1  1  0  0  0  1  1  0  0  0  1  1
  0  0  1  1  0  0  0  1  1  0  0  0  1  1  0  0  0  0
-15 |  1  1  1  1  0  1  0  1  0  1  0  1  1  0  1  0  1  1  0  1  0  1  1  0  1
  0  1  0  1  0  1  1  0  1  0  1  1  0  1  0  1  1  0
 15 |  0  0  0  0  1  0  1  0  1  0  1  0  0  1  0  1  0  0  1  0  1  0  0  1  0
  1  0  1  0  1  0  0  1  0  1  0  0  1  0  1  0  0  1
-16 |  1  1  0  1  1  1  1  1  1  1  1  1  0  1  1  1  1  0  1  1  1  1  0  1  1  1
  1  1  1  1  1  0  1  1  1  1  0  1  1  1  1  0  1  1
 16 |  0  0  1  0  0  0  0  0  0  0  0  0  1  0  0  0  0  1  0  0  0  0  1  0  0  0
  0  0  0  0  0  1  0  0  0  0  1  0  0  0  0  1  0  0
```

THE OR PLANE:

	2	16	16	13	23	4	10	9	27	23	21	20	17	18	7
1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0
2	0	0	1	0	0	0	0	0	1	0	1	1	0	0	0
3	0	0	0	0	1	0	0	0	0	0	1	0	0	0	1
4	0	0	0	0	1	0	0	0	1	0	0	0	0	1	0
5	0	0	1	1	0	0	1	0	0	0	0	0	0	0	1
6	0	1	0	0	1	0	0	1	0	0	0	0	0	1	0
7	0	0	1	1	0	0	0	0	1	0	0	0	1	1	0
8	0	1	0	0	0	0	1	0	1	0	0	0	1	0	0
9	0	1	1	0	0	0	1	0	1	0	0	1	1	1	0
10	0	0	0	0	1	0	0	0	1	0	0	1	1	0	0
11	0	0	0	1	0	0	1	0	1	1	0	1	0	1	0
12	0	0	0	1	0	0	1	0	1	0	0	1	0	0	1
13	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0
14	0	1	0	0	1	1	0	0	0	0	1	1	1	1	0
15	0	1	0	0	0	0	1	0	1	0	1	1	1	0	0
16	0	1	0	0	1	0	0	1	0	0	1	1	0	1	0
17	0	1	0	1	1	1	0	0	1	1	1	0	0	0	0
18	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0
19	0	1	0	1	1	0	0	0	1	0	1	0	1	1	0
20	0	1	0	0	1	0	0	1	0	0	1	0	1	0	0
21	0	0	0	0	1	0	0	0	1	0	1	0	0	1	0
22	0	0	0	0	1	0	0	0	0	1	0	1	0	0	1
23	0	0	1	0	1	0	0	1	1	0	1	0	0	0	0
24	0	1	0	0	0	0	0	0	1	1	1	1	1	1	0
25	0	0	1	1	0	0	1	0	0	1	1	1	1	0	0
26	0	1	0	0	0	0	0	0	1	1	1	1	0	1	0
27	0	1	1	0	0	1	0	0	1	1	1	1	0	0	0
28	0	0	1	1	1	0	0	1	0	1	1	0	1	1	0
29	0	0	0	0	1	0	0	0	1	1	1	0	1	0	0
30	0	0	0	0	1	0	0	0	1	0	0	1	0	1	0
31	0	1	0	0	1	0	0	0	1	1	1	0	0	0	1
32	0	1	1	0	1	0	0	1	1	1	1	0	0	0	0
33	0	0	0	1	1	0	1	0	0	1	0	1	1	1	0
34	0	1	1	0	0	1	0	0	1	1	0	1	1	0	0
35	0	0	1	1	0	0	1	0	1	1	0	1	0	1	0
36	0	0	0	1	1	0	0	1	0	1	0	1	0	0	1
37	0	0	1	0	0	0	0	0	1	1	0	1	0	0	0
38	0	0	0	0	1	0	0	1	0	1	0	0	1	1	0
39	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0
40	0	1	1	0	0	0	0	0	1	1	0	0	0	1	0
41	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1
42	0	0	1	1	1	0	0	1	1	1	0	0	0	0	0
43	0	0	0	0	1	0	0	0	1	1	1	0	0	1	0

PLA HEIGHT: 2668 Microns
PLA WIDTH: 3447 Microns

Appendix B

EPAD Files

EPAD CMOS Technology File (epad.analysis)

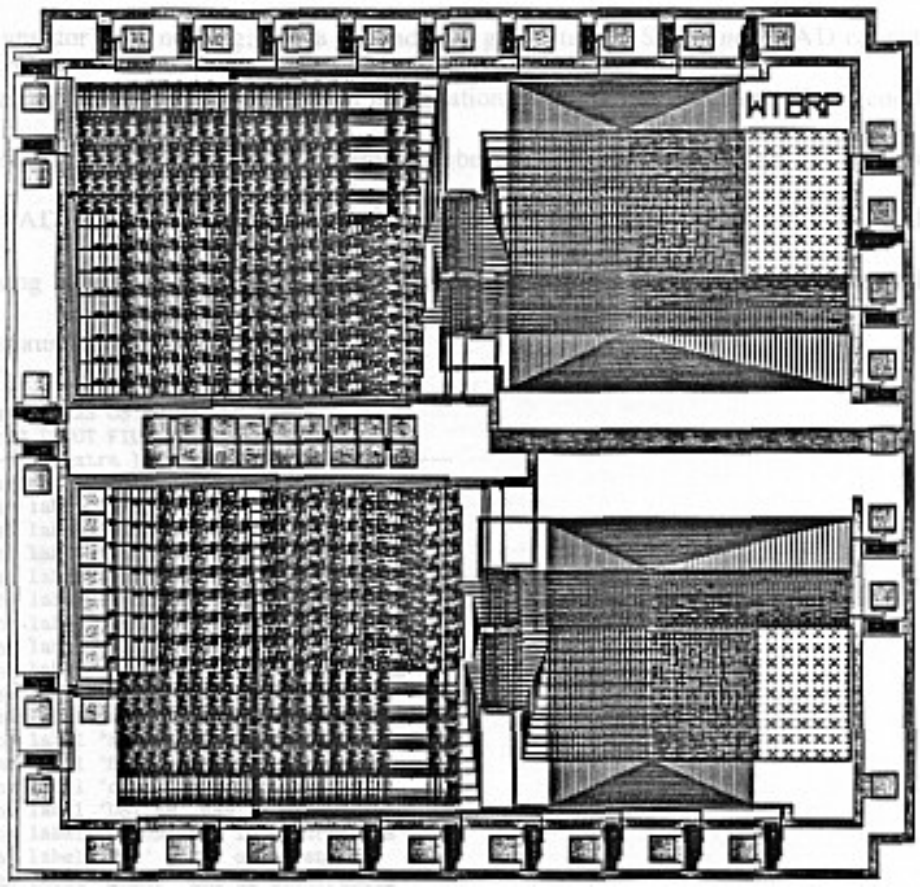
This file contains the CMOS technology parameters that were used to run EPAD. The technology file contains the *(kcapseries=1.0*. This parameter was explained in section 3.1.4 (Delay Models). This file is self-explanatory.

```
tech cmos-nt    # CMOS3 technology Jan 1986
#
# capacitance parameters (as .cadrc)
#
areatocap metal 27         # areatocap is cap per unit area in
areatocap poly 60          #  aF/micron*micron
areatocap diff 100
areatocap poly/diff 690
perimtocap poly 20         # perimtocap is cap per unit perimeter in
perimtocap diff 800        #  aF/micron
perimtocap metal 40
perimtocap poly/diff 50
#
# other capacitance parameters
#
coxn 690    # gate capacitance (NMOS) in aF/micron*micron
coxp 690    # gate capacitance (PMOS) in aF/micron*micron
covn 300    # overlap capacitance (NMOS) in aF/micron
covp 250    # overlap capacitance (PMOS) in aF/micron
#
# additional delay and power calculation factors
#
un  775.0    # electron surface mobility in cm*cm/(V-s)
up  250.0    # hole surface mobility in cm*cm/(V-s)
vtn 0.7      # zero bias threshold voltage NMOS in V
vtp 0.8      # zero bias threshold voltage (absolute value) PMOS in V
gamn 1.1     # bulk threshold parameter NMOS in V**0.5 (gamma)
gamp 0.6     # bulk threshold parameter PMOS in V**0.5
phin 0.6     # surface potential NMOS in V  (2*phi-f)
```

```
phip 0.6     # surface potential PMOS in V
#
# INPUT PARAMETERS
#
kcapseries 1.0    # factor for series transistor capacitance combination
#                   normally between 0.0 and 1.0
vdd 5.0      # power supply voltage in V
fsw 1.0      # switching frequency in MHz
```

Layout Input File (codec.cif)

The layout file is much too big to be shown here. Instead, a diagram of the layout is shown.



EPAD Output Log File (codec.log)

This file displays any errors that EPAD detected. Some transmission gate directionality is unresolved. It is a minor annoyance and is due to two points. This first point is one transistor at the most significant bit of the left shifter. This transistor does nothing; it is a redundancy generated by SPIL and EPAD cannot determine the direction of signal propagation through the transistor. The second point is test inserts. Error messages labelled with the name `tins` are due to EPAD errors in identifying the test inserts. Test inserts were not simulated using EPAD and SILOS. No noticeable effect on any codec results occurred because of these error messages.

```
Tue Dec 23 03:50:23 EST 1986
epad INPUT FILE =    codec.cif
----- mextra log ------------------------
Window: -2255.4 2255.4 -2255.4 2255.4 @ u=100
the label 'add_co' has 16 occurrences
the label 'add_cin' has 14 occurrences
the label 'lat_com' has 80 occurrences
the label 'src_add' has 8 occurrences
the label 'src_com' has 8 occurrences
the label 'dst_add' has 8 occurrences
the label 'SRC_EN' has 20 occurrences
the label 'dst_com' has 8 occurrences
the label 'add_a' has 16 occurrences
the label 'add_b' has 16 occurrences
the label 'add_s' has 16 occurrences
the label 'D_IN' has 16 occurrences
the label 'dst' has 8 occurrences
the label 'DST_EN' has 16 occurrences
the label 'D_BUS' has 16 occurrences
the label 'src' has 8 occurrences
the label 'D_OUT' has 16 occurrences
the label 'PHI1-' has 2 occurrences
the label 'PHI2' has 2 occurrences
    2584 neg enhancement
    2412 pos enhancement
```

```
3291 nodes
---------- epad.analysis parameters used ---------------
tech cmos-nt
areatocap metal 27
areatocap poly 60
areatocap diff 100
areatocap poly/diff 690
perimtocap metal 40
perimtocap poly 20
perimtocap diff 800
perimtocap poly/diff 50
coxn 690
coxp 690
covn 300
covp 250
GATES WITH > 10 INPUTS - ASSUMED COMPLEMENTARY
un 775.0
up 250.0
vtn 0.7
vtp 0.8
gamn 1.1
gamp 0.6
phin 0.6
phip 0.6
------- INPUT PARAMETERS -------
kcapseries 1.0
vdd 5.0
fsw 1.0
ERROR: Some transmission gate directionality unresolved
ERROR: Unresolved transfer gate directionality in SILOS input
8126 NPASS.1 1.70319 0.309967 1.3177 1.0 8145 sel_3 -
8145 NPASS.2 1.4354 0.261232 1.11052 1.0 8126 sel_3 -
tins2a NPASS.1 33.6633 6.12645 13.0221 1.0 tins2c tins2b -
tins2c NPASS.2 34.3378 6.24921 13.283 1.0 tins2a tins2b -
tins4c NPASS.1 34.3134 6.24476 13.2736 1.0 tins4d tins4a -
tins4d NPASS.2 36.1015 6.57018 13.9653 1.0 tins4c tins4a -
8494 NPASS.1 1.4353 0.261214 1.11045 1.0 8739 tsel_3 -
8739 NPASS.2 1.70309 0.309949 1.31763 1.0 8494 tsel_3 -
tins3a PPASS.1 6.79251 39.6306 13.2257 1.0 tins3c - tins3b
tins3c PPASS.2 6.87809 40.1299 13.3923 1.0 tins3a - tins3b
tins4b PPASS.1 6.9188 40.3674 13.4716 1.0 tins4d - tins4a
tins4d PPASS.2 7.17234 41.8467 13.9653 1.0 tins4b - tins4a
```

EPAD SILOS-Input File (codec.dat)

This file contains a logic description of the codec chip. It is too big to be shown in its entirety. Each line shown represents one capacitance from a node to ground. The nodes names are in the first column and are automatically generated by mextra, when EPAD calls mextra. The capacitances in femto-Farads are shown in the third column. This file contains other circuit elements such as CMOS gates, N-channel transmission gates and P-channel transmission gates, but they are not shown.

```
.TITLE   SILOS INPUT FOR   codec.cif
#
# Add clock/pattern specification for input nodes
#
n10059 .CAP     596
n10080 .CAP     130
n10082 .CAP     608
n10135 .CAP     665
n10138 .CAP     127
n10140 .CAP     127
n10142 .CAP     127
n10144 .CAP     126
n10146 .CAP     127
n10148 .CAP     127

 ...
```

EPAD Output File (codec.epad)

This file is the output from EPAD and contains power dissipation, area and delay predictions. This file is too big to be shown in its entirety. The first part of the file describes the areas of cells in the CIF layout. Only two cells are shown. The units of length and area for the cells are in physical μm, not design scale microns. The gate delays and power disspations are grouped by gate types. CMOS gate delays are listed first. This is followed by D-latch delays. D-latches are the level-sensitive latches used in the FSM latches and in data path registers. Finally complementary and non-complementary transmission gate data are shown. These tables are self-explanatory. The last three parts of the EPAD output file contain the circuit connectivity, again segregated by gate types.

```
Tue Dec 23 03:31:20 EST 1986
epad INPUT FILE =    codec.cif

-------------------------------------------------------
  SILICON AREA :

NAME/SYMBOL :     padouty;    1
HEIGHT (microns)    300
WIDTH (microns)     312
AREA (sq. microns)  93600
ASPECT RATIO        0.96
------------------
NAME/SYMBOL :     padiny;    2
HEIGHT (microns)    300
WIDTH (microns)     312
AREA (sq. microns)  93600
ASPECT RATIO        0.96
------------------

 ...

-------------------------------------------------------
```

```
CMOS GATE DELAYS , POWER :
Output node       Gate  PMOS (rise)  NMOS (fall)      Power  Frequency
                  type  delay (ns)   delay (ns)         uW      Mhz
================= ==== ============ ============ ========== ==========
10171             .INV         2.393        1.326       3.64      1.000
10226             .INV         0.847        0.782       2.15      1.000
10238             .INV         4.043        2.240       6.15      1.000

 ...

tsel_8            .NAND        4.255        3.274      17.98      1.000
tsel_9            .INV         4.329        3.997      10.98      1.000
                                      Total Power   11647.83

-------------------------------------------------------
D LATCH DELAYS , POWER :
 Output node     Input node      Rise input Fall input    Power  Freq.  Output
  Q or QBAR          D           delay(ns) delay(ns)       uW      Mhz   type
================ ================ ========= ========= ========= ======= ======
2237             1127                5.354     2.582      0.00   1.000 QBAR
src_0            1127                8.951     4.021     17.56   1.000 Q
12702            12567               5.354     2.582      0.00   1.000 QBAR

 ...

12082            tst_in_5           59.187    46.313      0.00   1.000 QBAR
tst_out5         tst_in_5           70.322    50.768    100.66   1.000 Q
                                        Total Power   4096.21

-------------------------------------------------------
TRANSMISSION GATE DELAYS , POWER :
*
  Output           Input          Rising   Falling       Power  Freq.  Type
   node             node         delay(ns) delay(ns)       uW      Mhz
================ ================ ========= ========= ========= ======= =====
10277            GND                 1.746     0.318      1.35   1.000 NPASS
10320            add_a#11            4.123     2.316      4.28   1.000 CPASS
10320            add_b#11           11.062     2.013      4.28   1.000 NPASS

 ...

tready           Vdd                 5.887    34.347     45.09   1.000 PPASS
tready           Vdd                 5.887    34.347     45.09   1.000 PPASS
tready           Vdd                 5.887    34.347     45.09   1.000 PPASS
                                   Total Power   15617.71
**
** Some transmission gate directionality is unresolved
** and therefore double counted in power summation
-------------------------------------------------------
CMOS GATE NETWORK CONNECTIONS :
Output node      Type Input nodes
================ ===== ===========
10171            .INV  add_co#12
10226            .INV  add_s#11
```

```
10238            .INV  add_b#11

 ...

tsel_7           .INV  14380
tsel_8           .NAND 14675,tPHI2
tsel_9           .INV  14376
-----------------------------------------------------
D LATCH NETWORK CONNECTIONS :
Output node      Input node      Clock Signal    Output
 Q or QBAR          D                            Type
================ ================ =============== ======
2237             1127            1885            QBAR
src_0            1127            1885            Q
12702            12567           12420           QBAR

 ...

tst_out4         tst_in_4        tPHI2           Q
12082            tst_in_5        tPHI2           QBAR
tst_out5         tst_in_5        tPHI2           Q
-----------------------------------------------------
TRANSMISSION GATE NETWORK CONNECTIONS :
*
Output node      Input node      NMOS Gate       PMOS Gate        Type
================ ================ =============== =============== =====
10277            GND             10256           -               NPASS
10320            add_a#11        add_b#11        10238           CPASS
10320            add_b#11        add_a#11        -               NPASS

 ...

tready           Vdd             -               16789           PPASS
tready           Vdd             -               16789           PPASS
tready           Vdd             -               16789           PPASS
**
** Some transmission gate directionality is unresolved
** and therefore both directions appear
```

Appendix C

SILOS Logic Simulation

This appendix contains the files associated with running a SILOS logic simulation.

Batch Command File (batchfile)

This file contains the UNIX™ command the was used to start the SILOS simulation. The file called `commands` contains the SILOS commands. The file called `output` contains the output of SILOS.

```
cat commands | silos > output
```

SILOS Commands File (commands)

This file contains the SILOS commands. The first command reads the three .dat files for SILOS. The second command runs a logic simulation from 0 nanoseconds to 400000 nanoseconds. The third command shows any errors if they existed. The fourth command exits SILOS and saves the simulation results.

```
input top.dat codec.dat bot.dat
sim 0 to 400000
ty err
exit save
```

Circuit Description Part 1 of 3 (top.dat)

This file contains the top of the SILOS circuit description file. It just shows the title.

```
.TITLE   Silos input from epad_1 for codec.dat
#
# The circuit description :
#
```

Circuit Description Part 2 of 3 (codec.dat)

This file contains a description of circuit elements of the codec. Recall the it was generated by EPAD. See appendix B (EPAD Files). It will not be repeated here.

Circuit Description Part 3 of 3 (bot.dat)

This file describes the logic test to be performed on the receiver. This file contains a description of the signals which are input to the circuit to perform a logic verification. It also describes the signals which were observed, the outputs of the circuit. Input signals are shown by the `.CLK` symbol in the second column of any line. The first column contains the signal name. The transmitter signals which are shown are: RESET, GO, PHI1-, PHI2 and the eight PCM input signals. The receiver signals are: RESET, GO, PHI1-, PHI2, the ADM input signal and the seven unused signals of the ADM_IN register which had to be grounded. Test structures were given input signals to prevent SILOS error messages, but test structure outputs were not analysed. Consider the signal `ntreset`, at time `0` it is defined to be `D1` (driving-strength, logic 1). At time `875` (nanoseconds), it is defined to be `D0`. The `Output data definitions` begin with the `.sym` statement. This statement indicates the meaning of charac-

ters in the output file. For example, SILOS shows a supply-strength signal (s0) by the symbol 0. Logic values may be 1, 0 or X (unknown). Signal strengths may be supply, driving, resistive or high-impedance. The statement .hex indicates groups of four logic outputs which are to have an equivalent hexadecimal name. For example, the most significant bit of the hexadecimal signal rdata_h is nD_7. More hexadecimal signals are defined than are used, but that did not affect the simulation. The final statement, .mon, indicates all the signals which are to displayed in the output file. Whenever any one of the signals changes, all signals will be displayed at the time that the change occurred. The signals in the .mon statement are: PHI1-, PHI2, RESET, GO, ADM input, READY, PCM output (high and low hexadecimal values), current FSM state (high and low hexadecimal values), data bus values (high and low hexadecimal values), source address, and destination address.

```
#
# Input waveforms :
#

# transmitter

ntreset    .CLK  0 D1  875 D0
ntgo       .CLK  0 D0 5625 D1
ntphi1-    .CLK  0 D0  250 D1                  1000 D0  .REP 0
ntphi2     .CLK  0 D0          500 D1  750 D0  1000 D0  .REP 0
#
nD_IN#8    .CLK  0 D0  192875 D1
nD_IN#9    .CLK  0 D1  192875 D1
ntpcmin5   .CLK  0 D0  192875 D0
ntpcmin4   .CLK  0 D0  192875 D0
ntpcmin3   .CLK  0 D0  192875 D0
ntpcmin2   .CLK  0 D0  192875 D0
ntpcmin1   .CLK  0 D0  192875 D0
```

```
ntpcmin0   .CLK  0 D0  192875 D0

# receiver

nrreset   .CLK  0 D1  875 D0
nrgo      .CLK  0 D0 5625 D1
nrphi1-   .CLK  0 D0  250 D1                  1000 D0  .REP 0
nrphi2    .CLK  0 D0          500 D1  750 D0  1000 D0  .REP 0
nADM_IN   .CLK  0 D1  160875 D0
#
nD_IN#1   .CLK  0 D1
nD_IN#2   .CLK  0 D1
nD_IN#3   .CLK  0 D1
nD_IN#4   .CLK  0 D1
nD_IN#5   .CLK  0 D1
nD_IN#6   .CLK  0 D1
nD_IN#7   .CLK  0 D1

# test structures

ntestin   .CLK  0 D0  100 D1  200 D0
ntins2a   .CLK  0 D0
ntins2b   .CLK  0 D0
ntins2c   .CLK  0 D0
ntins3a   .CLK  0 D1
ntins3b   .CLK  0 D1
ntins3c   .CLK  0 D1
ntins4a   .CLK  0 D0
ntins4b   .CLK  0 D1
ntins4c   .CLK  0 D0
ntins4d   .CLK  0 D1

#
# Output data definitions :
#

.sym s0=0 s*=X s1=1  d0=0 d*=X d1=1  r0=0 r*=X r1=1
+    z0=- z*=X z1=+  0d=v *d=X 1d=^  *s=S

.hex rdata_h=nD_7,nD_6,nD_5,nD_4 rdata_l=nD_3,nD_2,nD_1,nD_0
+    rstate_h=.GND,.GND,nst_out_5,nst_out_4
+    rstate_l=nst_out_3,nst_out_2,nst_out_1,nst_out_0
+    rinlat=n5943,n6154,n6288,n6366
+    routlat_1=.GND,.GND,nst_in_5,nst_in_4
+    routlat_2=nst_in_3,nst_in_2,nst_in_1,nst_in_0
+    rsrc=nsrc_3,nsrc_2,nsrc_1,nsrc_0
+    rdst=ndst_3,ndst_2,ndst_1,ndst_0
+    rdecsrc_1=n2151,n2145,n2141,n2137
+    rdecsrc_2=n2133,n2129,n2125,n2121
+    rdecsrc_3=.GND,.GND,n2119,n2117
+    rdecdst_1=n2149,n2147,n2143,n2139
+    rdecdst_2=n2135,n2131,n2127,n2123
+    raddlatb_h=nadd_b#7,nadd_b#6,nadd_b#5,nadd_b#4
+    raddlatb_l=nadd_b#3,nadd_b#2,nadd_b#1,nadd_b#0
```

```
+    raddlata_h=nadd_a#7,nadd_a#6,nadd_a#5,nadd_a#4
+    raddlata_l=nadd_a#3,nadd_a#2,nadd_a#1,nadd_a#0
+    rshfl_h=n7949,n7519,n7085,n6479
+    rshfl_l=n5757,n4513,n3774,n3087
+    rcompl_h=n7948,n7518,n7084,n6478
+    rcompl_l=n5756,n4512,n3773,n3086
+    rpcmout_h=nrpcmout7,nrpcmout6,nrpcmout5,nrpcmout4
+    rpcmout_l=nrpcmout3,nrpcmout2,nrpcmout1,nrpcmout0
+    rx_of_k_h=n7947,n7517,n7083,n6477
+    rx_of_k_l=n5755,n4511,n3772,n3085
+    rsx_of_k_h=n7946,n7516,n7082,n6476
+    rsx_of_k_l=n5754,n4510,n3771,n3084
+    rex_h=n7945,n7515,n7081,n6475
+    rex_l=n5753,n4509,n3770,n3083
+    roli_1=n894,n943,n1127,n1301
+    roli_2=n1462,n1709,n1729,n1886
+    roli_3=n2070,n2502,n2803,n3155
+    roli_4=.GND,n3506,n3736,n4044
+    raddout_h=n8147,n7718,n7293,n6861
+    raddout_l=n6147,n5013,n4050,n3381

#
# Output data :
#

.mon nrphi1- nrphi2 ;
+    nrreset nrgo nADM_IN ;
+    nbrready ;
+    rpcmout_h rpcmout_l ;
+    rstate_h rstate_l ;
+    rdata_h rdata_l ;
+    rsrc rdst ;
```

Circuit Description of the Transmitter (bot_tx.dat)

If this file replaces the file bot.dat, then logic verification tests will be performed on the transmitter. This file is the same as the receiver file (bot.dat) except for the corresponding transmitter signals in the .mon statement.

```
#
# Input waveforms :
#

# transmitter

ntreset    .CLK  0 D1  875 D0
ntgo       .CLK  0 D0 5625 D1
ntphi1-    .CLK  0 D0  250 D1                  1000 D0  .REP 0
ntphi2     .CLK  0 D0          500 D1  750 D0  1000 D0  .REP 0
#
nD_IN#8    .CLK  0 D0  192875 D1
nD_IN#9    .CLK  0 D1  192875 D1
ntpcmin5   .CLK  0 D0  192875 D0
ntpcmin4   .CLK  0 D0  192875 D0
ntpcmin3   .CLK  0 D0  192875 D0
ntpcmin2   .CLK  0 D0  192875 D0
ntpcmin1   .CLK  0 D0  192875 D0
ntpcmin0   .CLK  0 D0  192875 D0

# receiver

nrreset   .CLK  0 D1  875 D0
nrgo      .CLK  0 D0 5625 D1
nrphi1-   .CLK  0 D0  250 D1                  1000 D0  .REP 0
nrphi2    .CLK  0 D0          500 D1  750 D0  1000 D0  .REP 0
nADM_IN   .CLK  0 D1  160875 D0
#
nD_IN#1   .CLK  0 D1
nD_IN#2   .CLK  0 D1
nD_IN#3   .CLK  0 D1
nD_IN#4   .CLK  0 D1
nD_IN#5   .CLK  0 D1
nD_IN#6   .CLK  0 D1
nD_IN#7   .CLK  0 D1

# test structures

ntestin   .CLK  0 D0  100 D1  200 D0
ntins2a   .CLK  0 D0
ntins2b   .CLK  0 D0
```

```
ntins2c   .CLK  0 D0
ntins3a   .CLK  0 D1
ntins3b   .CLK  0 D1
ntins3c   .CLK  0 D1
ntins4a   .CLK  0 D0
ntins4b   .CLK  0 D1
ntins4c   .CLK  0 D0
ntins4d   .CLK  0 D1

#
# Output data definitions :
#

.sym s0=0 s*=X s1=1  d0=0 d*=X d1=1  r0=0 r*=X r1=1
+    z0=- z*=X z1=+  0d=v *d=X 1d=^  *s=S

.hex tdata_h=ntD_7,ntD_6,ntD_5,ntD_4 tdata_l=ntD_3,ntD_2,ntD_1,ntD_0
+    tstate_h=.GND,.GND,ntst_out5,ntst_out4
+    tstate_l=ntst_out3,ntst_out2,ntst_out1,ntst_out0
+    tinlat=n10525,n10377,n10209,n10082
+    toutlat_1=.GND,.GND,ntst_in_5,ntst_in_4
+    toutlat_2=ntst_in_3,ntst_in_2,ntst_in_1,ntst_in_0
+    tsrc=ntsrc_3,ntsrc_2,ntsrc_1,ntsrc_0
+    tdst=ntdst_3,ntdst_2,ntdst_1,ntdst_0
+    tdecsrc_1=n14691,n14685,n14681,n14677
+    tdecsrc_2=n14673,n14669,n14665,n14661
+    tdecsrc_3=.GND,.GND,n14659,n14657
+    tdecdst_1=n14689,n14687,n14683,n14679
+    tdecdst_2=n14675,n14671,n14667,n14663
+    taddlatb_h=nadd_b#15,nadd_b#14,nadd_b#13,nadd_b#12
+    taddlatb_l=nadd_b#11,nadd_b#10,nadd_b#9,nadd_b#8
+    taddlata_h=nadd_a#15,nadd_a#14,nadd_a#13,nadd_a#12
+    taddlata_l=nadd_a#11,nadd_a#10,nadd_a#9,nadd_a#8
+    tshfl_h=n8689,n9130,n9562,n10336
+    tshfl_l=n11438,n12612,n13274,n13959
+    tcompl_h=n8687,n9128,n9560,n10334
+    tcompl_l=n11436,n12610,n13272,n13957
+    tx_of_k_h=n8685,n9126,n9558,n10332
+    tx_of_k_l=n11434,n12608,n13270,n13955
+    tsx_of_k_h=n8683,n9124,n9556,n10330
+    tsx_of_k_l=n11432,n12606,n13268,n13953
+    tex_h=n8681,n9122,n9554,n10328
+    tex_l=n11430,n12604,n13266,n13951
+    toli_1=n14782,n14620,n14499,n14325
+    toli_2=n14205,n14018,n13852,n13710
+    toli_3=n13652,n13472,n13253,n13115
+    toli_4=.GND,n13001,n12751,n12567
+    taddout_h=n8607,n9049,n9480,n10226
+    taddout_l=n11158,n12421,n13150,n13830

#
# Output data :
#
```

```
.mon ntphi1- ntphi2 ;
+    ntreset ntgo
+    nD_IN#8 nD_IN#9 ntpcmin5 ntpcmin4 ntpcmin3 ntpcmin2 ntpcmin1 ntpcmin0 ;
+    nbtready ;
+    nadm_out ;
+    tstate_h tstate_l ;
+    tdata_h tdata_l ;
+    tsrc tdst
```

SILOS Output File (output)

This file contains the output of SILOS. Only some of the output file is shown. Signals are shown in columns and times are shown in rows. The part of the file which is shown indicates the PCM outputs of the receiver (PCMOUT_H and PCMOUT_L) when the READY signal rises. The meaning of this test is described in section 4.3.1 (SILOS Logic Verification).

```
- SILOS 2D.1-   MONITOR     20:22:20   Mar 25 1987
 SILOS INPUT FOR   CODEC.CIF


            NN NNN N RR RR RR RR
            RR RRA B PP SS DD SD
            PP RGD R CC TT AA RS
            HH EOM R MM AA TT CT
            II S _ E OO TT AA
            12 E I A UU EE __
            -  T N D TT __ HL
                   Y __ HL
                     HL
      TIME
         0  00 101 ? ** ** FF **
       ...
       763  10 101 0 00 ** 00 00
       875  10 001 0 00 ** 00 00
       ...
      4772  10 001 1 00 03 00 00
       ...
      5625  11 011 1 00 03 FF 00
       ...
```

```
  6763  10 011 0 00 04 FF 00
   ...
 30760  10 011 0 FF 03 FF 00
 30772  10 011 1 FF 03 FF 00
   ...
 55760  10 011 0 FF 03 FF 00
 55772  10 011 1 FF 03 FF 00
   ...
 76760  10 011 0 00 03 00 00
 76772  10 011 1 00 03 00 00
   ...
 97760  10 011 0 02 03 02 00
 97772  10 011 1 02 03 02 00
   ...
118760  10 011 0 05 03 05 00
118772  10 011 1 05 03 05 00
   ...
139760  10 011 0 09 03 09 00
139772  10 011 1 09 03 09 00
```

Appendix D

SILOS Critical Path Simulation

The execution of SILOS for the critical path analysis was the same as the execution described in the SILOS Logic Files appendix, except for the following different files.

Circuit Description File Part 3 of 3 (bot.dat)

This file describes the receiver analysis. The only difference from Appendix C (SILOS Logic Simulation) is the .mon statement. The .mon statement lists every signal that was necessary to perform a critical path analysis. The .mon signals are shown by lines. The first line gives PHI1-, PHI2 and PHI2'. The second line gives all the outputs of the FSM input latches. The third line gives all the outputs of the FSM output latches. The fourth line gives the data bus. The fifth line gives all the outputs of the address decoders. The sixth line gives the outputs of the adder's input ports. The seventh line gives the outputs of the left shifter's and complementer's input ports. The eighth line gives outputs all other register's input ports. The ninth line gives the inputs of all FSM output latches. The last line gives the inputs of the adder's output port.

```
#
# Input waveforms :
#

# transmitter

ntreset    .CLK  0 D1  875 D0
ntgo       .CLK  0 D0 5625 D1
ntphi1-    .CLK  0 D0  250 D1                  1000 D0  .REP 0
ntphi2     .CLK  0 D0          500 D1  750 D0  1000 D0  .REP 0
#
nD_IN#8    .CLK  0 D0  192875 D1
nD_IN#9    .CLK  0 D1  192875 D1
ntpcmin5   .CLK  0 D0  192875 D0
ntpcmin4   .CLK  0 D0  192875 D0
ntpcmin3   .CLK  0 D0  192875 D0
ntpcmin2   .CLK  0 D0  192875 D0
ntpcmin1   .CLK  0 D0  192875 D0
ntpcmin0   .CLK  0 D0  192875 D0

# receiver

nrreset   .CLK  0 D1  875 D0
nrgo      .CLK  0 D0 5625 D1
nrphi1-   .CLK  0 D0  250 D1                 1000 D0  .REP 0
nrphi2    .CLK  0 D0          500 D1  750 D0  1000 D0  .REP 0
nADM_IN   .CLK  0 D1  160875 D0
#
nD_IN#1   .CLK  0 D1
nD_IN#2   .CLK  0 D1
nD_IN#3   .CLK  0 D1
nD_IN#4   .CLK  0 D1
nD_IN#5   .CLK  0 D1
nD_IN#6   .CLK  0 D1
nD_IN#7   .CLK  0 D1

# test structures

ntestin   .CLK  0 D0  100 D1  200 D0
ntins2a   .CLK  0 D0
ntins2b   .CLK  0 D0
ntins2c   .CLK  0 D0
ntins3a   .CLK  0 D1
ntins3b   .CLK  0 D1
ntins3c   .CLK  0 D1
ntins4a   .CLK  0 D0
ntins4b   .CLK  0 D1
ntins4c   .CLK  0 D0
ntins4d   .CLK  0 D1

#
# Output data definitions :
#
```

```
.sym s0=0 s*=X s1=1  d0=0 d*=X d1=1  r0=0 r*=X r1=1
+    z0=- z*=X z1=+  0d=v *d=X 1d=^  *s=S

.hex rdata_h=nD_7,nD_6,nD_5,nD_4 rdata_l=nD_3,nD_2,nD_1,nD_0
+    rstate_h=.GND,.GND,nst_out_5,nst_out_4
+    rstate_l=nst_out_3,nst_out_2,nst_out_1,nst_out_0
+    rinlat=n5943,n6154,n6288,n6366
+    routlat_1=.GND,.GND,nst_in_5,nst_in_4
+    routlat_2=nst_in_3,nst_in_2,nst_in_1,nst_in_0
+    rsrc=nsrc_3,nsrc_2,nsrc_1,nsrc_0
+    rdst=ndst_3,ndst_2,ndst_1,ndst_0
+    rdecsrc_1=n2151,n2145,n2141,n2137
+    rdecsrc_2=n2133,n2129,n2125,n2121
+    rdecsrc_3=.GND,.GND,n2119,n2117
+    rdecdst_1=n2149,n2147,n2143,n2139
+    rdecdst_2=n2135,n2131,n2127,n2123
+    raddlatb_h=nadd_b#7,nadd_b#6,nadd_b#5,nadd_b#4
+    raddlatb_l=nadd_b#3,nadd_b#2,nadd_b#1,nadd_b#0
+    raddlata_h=nadd_a#7,nadd_a#6,nadd_a#5,nadd_a#4
+    raddlata_l=nadd_a#3,nadd_a#2,nadd_a#1,nadd_a#0
+    rshfl_h=n7949,n7519,n7085,n6479
+    rshfl_l=n5757,n4513,n3774,n3087
+    rcompl_h=n7948,n7518,n7084,n6478
+    rcompl_l=n5756,n4512,n3773,n3086
+    rpcmout_h=nrpcmout7,nrpcmout6,nrpcmout5,nrpcmout4
+    rpcmout_l=nrpcmout3,nrpcmout2,nrpcmout1,nrpcmout0
+    rx_of_k_h=n7947,n7517,n7083,n6477
+    rx_of_k_l=n5755,n4511,n3772,n3085
+    rsx_of_k_h=n7946,n7516,n7082,n6476
+    rsx_of_k_l=n5754,n4510,n3771,n3084
+    rex_h=n7945,n7515,n7081,n6475
+    rex_l=n5753,n4509,n3770,n3083
+    roli_1=n894,n943,n1127,n1301
+    roli_2=n1462,n1709,n1729,n1886
+    roli_3=n2070,n2502,n2803,n3155
+    roli_4=.GND,n3506,n3736,n4044
+    raddout_h=n8147,n7718,n7293,n6861
+    raddout_l=n6147,n5013,n4050,n3381

#
# Output data :
#

.mon nrphi1- nrphi2 ; n1885 ;
+    rstate_h rstate_l rinlat n6487;
+    routlat_1 routlat_2 rsrc rdst nbrready ;
+    rdata_h rdata_l ;
+    rdecsrc_1 rdecsrc_2 rdecsrc_3 rdecdst_1 rdecdst_2 ;
+    raddlatb_h raddlatb_l raddlata_h raddlata_l
+    rshfl_h rshfl_l rcompl_h rcompl_l
+    rpcmout_h rpcmout_l rx_of_k_h rx_of_k_l rsx_of_k_h rsx_of_k_l rex_h rex_l ;
+    roli_1 roli_2 roli_3 roli_4 ;
+    raddout_h raddout_l
```

Circuit Description File Part 3 of 3 (bot_tx.dat)

This file describes the transmitter analysis. It is identical to the receiver file (bot.dat) except for equivalent transmitter .mon signals.

```
#
# Input waveforms :
#

# transmitter

ntreset    .CLK  0 D1  875 D0
ntgo       .CLK  0 D0 5625 D1
ntphi1-    .CLK  0 D0  250 D1                  1000 D0  .REP 0
ntphi2     .CLK  0 D0          500 D1  750 D0  1000 D0  .REP 0
#
nD_IN#8    .CLK  0 D0  192875 D1
nD_IN#9    .CLK  0 D1  192875 D1
ntpcmin5   .CLK  0 D0  192875 D0
ntpcmin4   .CLK  0 D0  192875 D0
ntpcmin3   .CLK  0 D0  192875 D0
ntpcmin2   .CLK  0 D0  192875 D0
ntpcmin1   .CLK  0 D0  192875 D0
ntpcmin0   .CLK  0 D0  192875 D0

# receiver

nrreset   .CLK  0 D1  875 D0
nrgo      .CLK  0 D0 5625 D1
nrphi1-   .CLK  0 D0  250 D1                 1000 D0  .REP 0
nrphi2    .CLK  0 D0          500 D1  750 D0  1000 D0  .REP 0
nADM_IN   .CLK  0 D1  160875 D0
#
nD_IN#1   .CLK  0 D1
nD_IN#2   .CLK  0 D1
nD_IN#3   .CLK  0 D1
nD_IN#4   .CLK  0 D1
nD_IN#5   .CLK  0 D1
nD_IN#6   .CLK  0 D1
nD_IN#7   .CLK  0 D1

# test structures

ntestin   .CLK  0 D0  100 D1  200 D0
ntins2a   .CLK  0 D0
ntins2b   .CLK  0 D0
ntins2c   .CLK  0 D0
ntins3a   .CLK  0 D1
```

```
ntins3b   .CLK  0 D1
ntins3c   .CLK  0 D1
ntins4a   .CLK  0 D0
ntins4b   .CLK  0 D1
ntins4c   .CLK  0 D0
ntins4d   .CLK  0 D1

#
# Output data definitions :
#

.sym s0=0 s*=X s1=1  d0=0 d*=X d1=1  r0=0 r*=X r1=1
+    z0=- z*=X z1=+  0d=v *d=X 1d=^  *s=S

.hex tdata_h=ntD_7,ntD_6,ntD_5,ntD_4 tdata_l=ntD_3,ntD_2,ntD_1,ntD_0
+    tstate_h=.GND,.GND,ntst_out5,ntst_out4
+    tstate_l=ntst_out3,ntst_out2,ntst_out1,ntst_out0
+    tinlat=n10525,n10377,n10209,n10082
+    toutlat_1=.GND,.GND,ntst_in_5,ntst_in_4
+    toutlat_2=ntst_in_3,ntst_in_2,ntst_in_1,ntst_in_0
+    tsrc=ntsrc_3,ntsrc_2,ntsrc_1,ntsrc_0
+    tdst=ntdst_3,ntdst_2,ntdst_1,ntdst_0
+    tdecsrc_1=n14691,n14685,n14681,n14677
+    tdecsrc_2=n14673,n14669,n14665,n14661
+    tdecsrc_3=.GND,.GND,n14659,n14657
+    tdecdst_1=n14689,n14687,n14683,n14679
+    tdecdst_2=n14675,n14671,n14667,n14663
+    taddlatb_h=nadd_b#15,nadd_b#14,nadd_b#13,nadd_b#12
+    taddlatb_l=nadd_b#11,nadd_b#10,nadd_b#9,nadd_b#8
+    taddlata_h=nadd_a#15,nadd_a#14,nadd_a#13,nadd_a#12
+    taddlata_l=nadd_a#11,nadd_a#10,nadd_a#9,nadd_a#8
+    tshfl_h=n8689,n9130,n9562,n10336
+    tshfl_l=n11438,n12612,n13274,n13959
+    tcompl_h=n8687,n9128,n9560,n10334
+    tcompl_l=n11436,n12610,n13272,n13957
+    tx_of_k_h=n8685,n9126,n9558,n10332
+    tx_of_k_l=n11434,n12608,n13270,n13955
+    tsx_of_k_h=n8683,n9124,n9556,n10330
+    tsx_of_k_l=n11432,n12606,n13268,n13953
+    tex_h=n8681,n9122,n9554,n10328
+    tex_l=n11430,n12604,n13266,n13951
+    toli_1=n14782,n14620,n14499,n14325
+    toli_2=n14205,n14018,n13852,n13710
+    toli_3=n13652,n13472,n13253,n13115
+    toli_4=.GND,n13001,n12751,n12567
+    taddout_h=n8607,n9049,n9480,n10226
+    taddout_l=n11158,n12421,n13150,n13830

#
# Output data :
#

.mon ntphi1- ntphi2 ; n12420 ;
+    tstate_h tstate_l tinlat n9880;
```

```
+    toutlat_1 toutlat_2 tsrc tdst nbtready ;
+    tdata_h tdata_l ;
+    tdecsrc_1 tdecsrc_2 tdecsrc_3 tdecdst_1 tdecdst_2 ;
+    taddlatb_h taddlatb_l taddlata_h taddlata_l
+    tshfl_h tshfl_l tcompl_h tcompl_l
+    nadm_out tx_of_k_h tx_of_k_l tsx_of_k_h tsx_of_k_l tex_h tex_l ;
+    toli_1 toli_2 toli_3 toli_4 ;
+    taddout_h taddout_l
```

SILOS Output File (output)

This file contains the output of SILOS. Not all of the file is shown. The file has been described in section 4.3.2 (SILOS Critical Path Analysis).

```
- SILOS 2D.1-   MONITOR     18:44:57   Mar 25 1987
SILOS INPUT FOR   CODEC.CIF


          NN N RRRN RRRRN RR RRRRR RRRRRRRRRRRRRRR RRRR RR
          RR 1 SSI6 OOSDB DD DDDDD AAAASSCCPPXXSSEE OOOO AA
          PP 8 TTN4 UURSR AA EEEEE DDDDHHOOCC__XXXX LLLL DD
          HH 8 AAL8 TTCTR TT CCCCC DDDDFFMMMMOO____ IIII DD
          II 5 TTA7 LL  E AA SSSDD LLLLLLPPOOFFOOHL ____ OO
          12   EET  AA  A __ RRRSS AAAA__LLUU__FF   1234 UU
          -    __   TT  D HL CCCTT TTTTHL__TTKK__        TT
               HL   __  Y    _____ BBAA  HL___KK         __
                    12      12312 _____    HLHL__        HL
                                  HLHL       HL
     TIME
        0  00 1 ***? ****? FF ***** ***************** **** **
      250  10 1 ***? ****? FF ***** ***************** **** **
       ...
    13520  11 0 0EF0 0F520 FF 20080 FF01000000000000 3274 0*
    13522  11 0 0EF0 0F520 FF 20080 FF01000000000000 3274 01
    13526  11 0 0EF0 0F520 FF 20080 FF01000000000000 3274 03
    13534  11 0 0EF0 0F520 FF 20080 FF01000000000000 3274 07
    13542  11 0 0EF0 0F520 FF 20080 FF01000000000000 3274 0F
    13550  11 0 0EF0 0F520 FF 20080 FF01000000000000 3274 1F
    13556  11 0 0FF0 0F520 FF 20080 FF01000000000000 3274 1F
    13558  11 0 0FF0 0F520 FF 20080 FF01000000000000 3274 3F
    13566  11 0 0FF0 0F520 FF 20080 FF01000000000000 3274 7F
    13574  11 0 0FF0 0F520 FF 20080 FF01000000000000 3274 FF
    13583  11 0 0FF0 0F520 FF 20080 FF01000000000000 3074 FF
       ...
```

Appendix E

SILOS Fault Simulation

This appendix contains the files associated with running a SILOS fault simulation on the receiver. Fault simulation was not performed on the transmitter since it was verified in a *back-to-back* test with the receiver.

Batch Command File (batchfile)

This file contains the command the was used to start the SILOS simulation.

```
cat commands | silos > output
```

SILOS Commands File (commands)

This file contains the SILOS commands. This file is similar to the file which was described in Appendix C, with the following exceptions. A file called `inst.dat` has been included as SILOS input in order to specify fault simulation commands. The statement `ty clocks` displays information about all the `.CLK` signals. The statement `ty noconv` displays information about all the signals which did not converge during simulation; all simulation signals did converge. The next statement specifies fault simulation to be performed between 950 ns and 1023000 ns with circuit outputs analysed every 1000 ns. The next statement prints out an activity summary of circuit nodes. The next statement prints out all detected and undetected faults.

```
input top.dat receiver.dat bot.dat inst.dat
sim 0 to 1023000
ty err
ty clocks
ty noconv
fsim 950 to 1023000 step 1000
ty activity
ty faults / det und
exit save
```

Circuit Description Part 1 of 3 (top.dat)

This file contains the top of the SILOS circuit description file.

```
.TITLE   Silos input from epad_1 for receiver.dat
#
# The circuit description :
#
```

Circuit Description Part 2 of 3 (receiver.dat)

This file contains a description of circuit elements of the receiver. Recall the it was generated by EPAD. This file is too large to be shown in its entirety. It is similar to the codec.dat file shown in Appendix C (SILOS Logic Verification).

Circuit Description Part 3 of 3 (bot.dat)

This file describes the logic test to be performed on the receiver. This file contains a description of the signals which are input to the circuit to perform a logic verification. It also describes the signals which were observed, the outputs of the circuit. Additionally, this file contains the test pattern for the receiver.

```
#
# Input waveforms :
#

nRESET    .CLK  0 D1  875 D0
nGO       .CLK  0 D0 5625 D1
nPHI1-    .CLK  0 D0  250 D1                  1000 D0  .REP 0
nPHI2     .CLK  0 D0          500 D1  750 D0  1000 D0  .REP 0
nADM_IN   .CLK  0 D1  160875 D0
#
nD_IN#1   .CLK  0 D1
nD_IN#2   .CLK  0 D1
nD_IN#3   .CLK  0 D1
nD_IN#4   .CLK  0 D1
nD_IN#5   .CLK  0 D1
nD_IN#6   .CLK  0 D1
nD_IN#7   .CLK  0 D1

#
# Output data :
#

.sym s0=0 s*=X s1=1  d0=0 d*=X d1=1  r0=0 r*=X r1=1
+    z0=- z*=X z1=+  0d=v *d=X 1d=^  *s=S

.mon nRESET nGO nPHI1- nPHI2 ; nADM_IN ;;
+    nREADY ; npcm_7 npcm_6 npcm_5 npcm_4 npcm_3 npcm_2 npcm_1 npcm_0 ;;
+    nD_7 nD_6 nD_5 nD_4 nD_3 nD_2 nD_1 nD_0 ;;
+    nst_out_5 nst_out_4 nst_out_3 nst_out_2 nst_out_1 nst_out_0 ;;
+    nsrc_3 nsrc_2 nsrc_1 nsrc_0 ; ndst_3 ndst_2 ndst_1 ndst_0 ;;
```

Fault Simulation Method File (inst.dat)

This file contains a description of the fault simulation method. Essentially it shows that a thorough non-statistical fault simulation was performed. The observable output signals are given by the .tnode statement. The statements .slow to .ishigh specify that 100 percent of all gate-output stuck-low, gate-output stuck-high, gate-input stuck-low and gate-input stuck-high faults are to be simulated. The .fmon indicates signals to be displayed during fault simulation. The last statement indicates the any possible detections are counted.

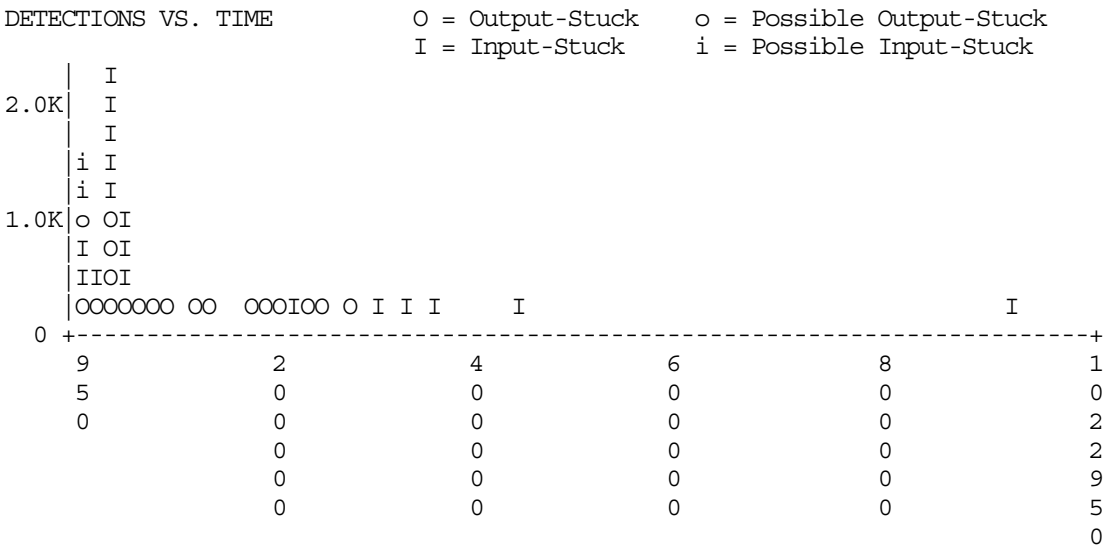
```
.tnode nready npcm_7 npcm_6 npcm_5 npcm_4 npcm_3 npcm_2 npcm_1 npcm_0
.slow   .pct=100%
.shigh  .pct=100%
.islow  .pct=100%
.ishigh .pct=100%
.fmon
.fcontrol .npdet=0 .fitr=500
```

SILOS Output File (output)

This file contains the output of SILOS. Only some of the output file is shown, that part relevant to the fault simulation summary. The OVERALL FAULT DETECTION is also called the fault coverage. It is the sum of total hard detections and total possible detections. The end of the file shows a rough plot of the number of detections versus time in nanoseconds.

	NUMBER FAULTED		HARD DETECTION		POSSIBLE DETECTION		UNDETECTED FAULTS	
	Count	%	Count	%	Count	%	Count	%
.SLOW	1031	100.0	766	74.3	49	4.8	216	21.0
.SHIGH	1031	100.0	697	67.6	210	20.4	124	12.0
.ISLOW	2115	100.0	1595	75.4	68	3.2	452	21.4
.ISHIGH	2174	100.0	1385	63.7	316	14.5	473	21.8
TOTAL	6351	100.0	4443	70.0	643	10.1	1265	19.9

```
 -------------------------------------
|                                     |
|   OVERALL FAULT DETECTION:  80.1%   |
|                                     |
 -------------------------------------
```


DETECTIONS VS. TIME
O = Output-Stuck
o = Possible Output-Stuck
I = Input-Stuck
i = Possible Input-Stuck
2.0K
1.0K
0
9 2 4 6 8 1
5 0 0 0 0 0
0 0 0 0 0 2
0 0 0 0 2
0 0 0 0 9
0 0 0 0 5
0

REFERENCES

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

[2] P.A.D.Powell, M.I.Elmasry, The ICEWATER Language and Interpreter, *21st Design Automation Conference*, IEEE, June, 1984, pp 98-102.

[3] M.Pulver, M.I.Elmasry, Using Igloo: A Constraint Based Layout Language for VLSI Design, *1987 Canadian Conference on Very Large Scale Integration*, University of Manitoba, Winnipeg, Manitoba, Oct., 1987.

[4] UW/NW VLSI Consortium, *VLSI Design Tools Reference Manual Release 1.0*, University of Washington, Seattle, Washington, Oct, 1983.

[5] W.S.Scott, R.N.Mayo, G.Hamachi, J.K.Ousterhout and editors, *1986 VLSI Tools: Still More Works by the Original Artists*, Report Number UCB/CSD 86/272, University of California, Berkeley, California, Dec., 1985.

[6] Mohamed I. Elmasry, *Digital VLSI Systems*, IEEE Press, New York, New York, 1985.

[7] D.E.Krekelberg, G.E.Sobelman and C.S.Jhon, *Yet Another Silicon Compiler*, Proceedings of the 22nd Design Automation Conference, Jun. 1985, pp 176-182.

[8] N.Bergmann, *Software Support for FIRST*, Technical Report, Edinburg University, Jun. 1982.

[9] R.Jamier and A.A.Jerraya, *APOLLON, A Data-Path Silicon Compiler*, IEEE Circuits and Devices Magazine, Vol. 1, No. 3, May 1985.

[10] A.V.Goldberg, S.S.Hirschhorn, K.J.Lieberherr, *Approaches Toward Silicon Compilation*, IEEE Circuits and Devices Magazine, Vol. 1, No. 3, May 1985.

[11] J.R.Southard, *MacPitts: An Approach to Silicon Compilation*, Computer, Vol. 16, Dec. 1983, pp74-82.

[12] D.J.Salomon, S.Sadler and M.I.Elmasry, A VLSI Architecture and a Silicon Compiler for Designing Numerical Processors, *VLSI DESIGN*, pp. 62-70, Feb., 1985.

[13] B.R.Petersen, B.A.White, D.J.Salomon, and M.I.Elmasry, SPIL: A Silicon Compiler with Performance Evaluation, *Proceedings of the International Conference on Computer Aided Design (ICCAD-86)*, Santa Clara, California, Nov., 1986.

[14] ANSI/IEEE770X3.97-198x *American National Standard Programming Language Pascal*, Prepared by the Joint ANSI/X2J9 IEEE Pascal Standards Committee X3J9/82-151, JPC/82-151, Oct. 19, 1982.

[15] J.L.LoCicero and D.L.Schilling, *An All-Digital Technique for ADM to PCM Conversion*, 1976 National Telecommunications Conference, Pt.II, Dallas, Texas, U.S.A, pp. 29.2/1-5, Nov. 29 - Dec. 1, 1976.

[16] B.A.White, B.R.Petersen, D.J.Salomon, and M.I.Elmasry, *Chip Design using SPIL : A Silicon Compiler with Performance Evaluation*, VLSI Group, University of Waterloo, Waterloo, Ontario, May 1987, also available as ICR Report No. UW/ICR 87-07 (Includes: The SPIL User's Manual).

[17] J.M.Leask, P.M.Gaboury and M.I.Elmasry, PLAmate, A PLA/FSM Compiler for MOS Technologies, *1984 Canadian VLSI Conference*, Edmonton, Alberta.

[18] Canadian Microelectronics Corporation, *CMC Guide for Designers Using the Northern Telecom CMOS3 Process*, Queen's University, Jun., 1985.

[19] Canadian Microelectronics Corporation, *Guide to the Integrated Circuit Implementation Services of the Canadian Microelectronics Corporation*, Queen's University, Jun. 4, 1986.

[20] B.R.Mears, A Modular Method for Designing Custom Signal Processing Integrated Circuits, *Proceedings of the IEEE Conference on Digital Signal Processing*, San Diego, California, 1983.

[21] K.Hwang, F.A.Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill Book Company, New York, New York, 1987.

[22] M.G.H.Katevenis, *Reduced Instruction Set Computer Architectures for VLSI*, The MIT Press, Cambridge, Mass., 1985.

[23] SIMUCAD, *SILOS Logic and Switch-Level Simulator User's Manual*, Rev. 4, Incline Village, Nevada, Sep. 1986.

[24] A.V.Aho, B.W.Kernighan and P.J.Weinberger, *A Pattern Scanning and Processing Language (Second Edition)*, Bell Laboratories, Murray Hill, New Jersey, Sep, 1978.

[25] B.A.White, M.I.Elmasry, *Performance Estimation in CMOS VLSI Circuits*, VLSI Group, University of Waterloo, Waterloo, Ontario, Feb. 1986, also available as ICR Report No. UW/ICR 87-06.

[26] A.Vladimirescu, A.R.Newton and D.O.Pederson, *SPICE Version 2G6 User's Guide*, University of California, Berkeley, Calif., U.S.A., Oct., 1980.

[27] J.R.Burns, Switching Response of Complementary-Symmetry MOS Transistor Logic Circuits, *RCA Review*, pp. 627-661, Dec., 1964.

[28] S.M.Kang, Accurate Simulation of Power Dissipation in VLSI Circuits, *IEEE Journal of Solid-State Circuits*, Vol. SC-21, NO. 5, Oct, 1986.

[29] J.Ousterhout, *Editing VLSI Circuits with Caesar*, University of California, Berkeley, Cal., 1984.

[30] L.R.Rabiner, R.W.Schafer, *Digital Processing of Speech Signals*, Prentice-Hall Inc., U.S.A, 1978.

[31] A.V.Oppenheim, R.W.Schafer, *Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, New Jersey, 1975.

[32] W.D.Stanley, G.R.Dougherty, R.Dougherty, *Digital Signal Processing*, Reston Publishing Company Inc., A Prentice-Hall Company, Reston, Virginia, 22090, 1984.

[33] T.W.Williams, K.P.Parker, *Design for Testability-A Survey*, Proceedings of the IEEE, Vol.71, No. 1, Jan. 1983.

[34] Paul DeMone, *CMOS3 I/O Cells*, The Canadian Microelectronics Corporation, Oct. 14, 1986.

[35] L.A.Glasser, D.W.Dobberpuhl, *The Design and Analysis of VLSI Circuits*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1985.

[36] ENDOT Inc., *N.2 Simulator User's Manual*, Document # 106, Version 1.12 - 12/2/85, 1985.

[37] P.Penfield, J.Rubinstein, Signal Delay in RC Tree Networks, *ACM IEEE Design Automation Conference Proceedings*, Nashville, Tenn., 1981, pp613-617.

[38] A.Vladimirescu and S.Liu, *The Simulation of MOS Integrated Circuits Using SPICE2*, University of California, Berkeley, Feb, 1980.