M85/11
287 Pages

AUTOMATIC GENERATION OF SIGNAL PROCESSING INTEGRATED CIRCUITS

by

Stephen P. Pope

Memorandum No. UCB/ERL M85/11

22 February 1985

AUTOMATIC GENERATION OF SIGNAL PROCESSING INTEGRATED CIRCUITS

by

Stephen P. Pope

Memorandum No. UCB/ERL M85/11

22 February 1985

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Automatic Generation of Signal Processing Integrated Circuits

Ph.D. Stephen P. Pope E.E.C.S.

Chairman of Committee

A system for the automated design of signal processing integrated circuits is described in this thesis. The system is based on a library of circuit cells, and a software package which can configure the cells into complete integrated circuits. The architecture of the cell library is optimized for low and medium bandwidth digital signal processing applications. Circuits designed with the system use a multiprocessor architecture.

Input to the system is a design file written in a specialized programming language. Software emulation from the design file is used to verify performance. A two-pass silicon compiler is used to translate the design file into a mask-level description of an integrated circuit.

A major goal of the project is to make the system usable by those with little or no formal training in integrated circuits. A second goal is to reduce the time and cost associated with performing an integrated circuit design, while still producing designs which are reasonably efficient in their use of the technology.

Development of the system was guided by basic research on appropriate architectures and circuit constructs for signal processors. As part of this research an integrated circuit was designed which performs speech analysis and synthesis. This vocoder circuit is intended for use in low-bit-rate digital speech transmission systems.

Dedication

This thesis is dedicated to my wife, Kathleen.

Acknowledgements

The list of those who influenced this research is quite long, and it is only possible to mention a few individuals here. Among the contributors to the LPC Vocoder design, Ron Fellman, Thomas Glad and Bjorn Solberg played major roles. Jan Rabaey and Peter Reutz made large contributions to the silicon compiler work. I would also like to extend my thanks to Prof. Richard Newton and his students, without whose work on CAD tools little of this research would have been possible.

Most of all I would like to thank Prof. Robert Brodersen for his extensive support and management of this project.

Table of Contents

Chapter 1 – Digital Signal Processors

1.1 Dedicated Signal Processors

The ability to design complex, large-scale integrated (LSI) circuits has not kept up with the technological advances that allow such devices to be fabricated. This situation has led to an increasing use of computer automation in integrated circuit design, and a decreasing emphasis on minimization of circuit area. This dissertation is concerned with design methods specifically intended for digital signal processing applications. The goal of such methods is the ability to implement economically important circuit functions with reasonable circuit density, while keeping the design time short.

Despite numerous predictions over the past decade that digital signal processing LSI circuits would soon have a huge impact on large segments of the industry, such an effect has not yet materialized. To better understand why this is so, it is necessary to consider how signal processing functions differ from other applications.

A *signal* is a representation of a time-varying physical quantity. Common examples of signals are sounds, vibrations, and electromagnetic waves. A *signal processor* is a device or system which conditions, analyzes, synthesizes, or otherwise modifies or creates signals.

Signal processors may be subdivided into general purpose signal processors and dedicated signal processors. A general-purpose signal processor is one that may, through reconfiguration or reprogramming, perform a variety of fundamentally different algorithms. This dissertation is concerned with *dedicated* signal processors — ones that perform a single algorithm. There might be a few variables or parameters that may be modified, but such a system, once created, is pretty much devoted to a single task. Also, the systems studied here are those which work with sampled and digitized (quantized) signals. Digital signal processors such as

these have a number of advantages over their analog counterparts: reproducibility, control over dynamic range and other performance aspects, and the ability to be programmed to perform complex tasks.

From an abstract point of view, virtually any computation would satisfy the above definition of signal processing. After all, the inputs and outputs of any computing system can be viewed as time sequences. From a practical point of view, signal processors are distinguished by the following two properties:

(1) Because signals result from ongoing, real-world physical processes, certain types of processing tend to recur in different signal-processing situations. Filtering, modulation, correlation and convolution are all classical signal processing functions.

(2) The signals involved are processed in *real time*. This means that the rate of processing is sufficient to keep up with incoming and outgoing signals. Thus processing could proceed indefinitely without loss of continuity.

Central to the research described here is the utilization of integrated circuit (IC) technology in the implementation of signal processing systems. There are two aspects in which the use of this technology is significant. First is the potential for dramatic cost savings that arises when custom integrated circuits are designed into a system. Second is the fact that conventional design methods and system architectures are no longer suitable as the level of integration becomes very high.

The remaining sections of this chapter cover background material regarding digital signal processing systems, their architectures, and techniques for their implementation using IC technology.

Chapter 2 discusses the design of an integrated circuit which implements a

low-bit-rate speech analysis/synthesis system. This *LPC Vocoder* integrated circuit represents a fairly complex and sophisticated signal processing system which uses a variety of processing techniques in its implementation. The LPC Vocoder IC was designed in a *full custom* fashion -- the goal of the project was to create as efficient a vocoder implementation as possible, without solving the more general problem of implementing other, similar applications. The vocoder IC was fabricated and tested, with test results presented in Chapter 2.

Chapters 3, 4 and 5 represent a broadening of scope of the research. The premise is that similar implementation techniques can be applied to different signal processing applications with good results. To take advantage of this generality, a *design system* was created which allows the rapid generation of digital signal IC's. The IC's produced with this system are assembled from the cell library described in Chapter 3. Chapter 4 describes a software package which automates much of the design process. In Chapter 5 the results of this approach are presented.

1.2 Processing of Sampled Signals

Signal processing algorithms are often presented by means of a signal flow diagram. Fig 1.1 is a signal flow diagram for a second-order filter section. Note that the primitive operations involved are multiplication, addition, and delays. All linear filters, however complex, are composed of these same operations.

The filter section may be implemented by repeatedly executing the following sequence of operations. The period of repetition is one sample interval. The quantities A and B represent the state (memory) of the filter, while X is a temporary variable. ":=" is the assignment operator.

$X := input + kA + gB$

$output := B$

$B := A$

$A := X$

In order for this abstract program to operate properly, the timing of the system must be such that each time the program is repeated the values "input" and "output" refer to new input and output samples respectively. The four-instruction program is executed from top to bottom without branching, and repeats each sample.

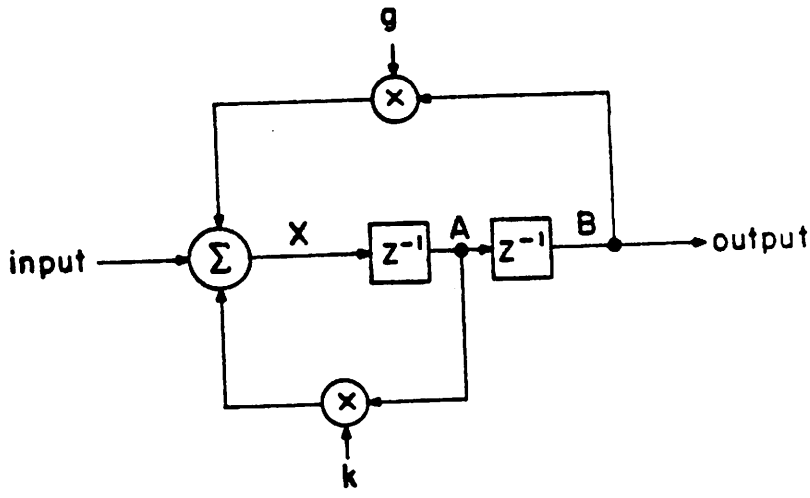


Fig.1.1 Flow diagram for a second-order filter

The example above is very simple; however, it suggests that all linear digital filters (and possibly other signal processing functions as well) may be implemented with by a processor with the following properties:

(1) A simple instruction set

(2) A program which is executed once per sample

(3) A lack of branches in the program flow

The research described in this dissertation demonstrates that this simple model serves as a basis for the design of reasonably powerful signal processors.

It should be noted that some system functions do not conform to the above model. Consider the Fast Fourier Transform [1], for example. In this case a block of input data (say 128 samples) is processed as a unit, resulting in an algorithm that is difficult to express by a program that repeats at the sample rate. Thus property (2) above would preclude efficient implementation of a Fast Fourier Transformer using the the approach described above.

Many signal processing algorithms make use of a frame. A frame is an interval many samples in length. The concept of a frame is significant because many interesting attributes of a signal, such as second moments and spectral characteristics, vary much less rapidly than the signal waveform itself. This being the case, it is possible to represent the desired attribute as being constant over an interval of considerably greater duration than the sample interval. This sort of data reduction is known as signal analysis and is widely used. The inverse operation, wherein a sampled-data signal is created from lower bandwidth data, is termed signal synthesis.

A typical signal processing system might consist of a front-end in which sample-rate processing is performed, an intermediate level in which frame-rate

processing is performed, and additional levels in which processing occurs at increasingly slower data rates. For those levels with a sufficiently slow data rate, conventional computer hardware may be programmed to economically perform the operations required. Other levels may require special-purpose hardware to meet performance goals.

The cell library of Chapter 3 allows a frame rate to be defined and used for the purpose of off-chip I/O. Semi-custom chips designed using the cell library can then serve as special-purpose front-end processors. I/O processing differs qualitatively from the other sorts of processing in a typical system, justifying separate hardware. Also motivating this focus is the perception that the I/O processing is often the major obstacle to a successful hardware realization.

1.3 Architectural Alternatives for Digital Signal Processing

Early digital signal processors were assembled from small- and medium-scale integrated circuits. Since this approach led to rather bulky systems, a considerable degree of specialization is evident in these early designs. Typical examples would be special purpose Fast Fourier Transform (FFT) machines [2], and a computer hard-wired to perform a speech pitch detection algorithm [3]. A considerable emphasis was often placed on fast array multipliers. In fact, much of the research on multiplier implementations was done by those pursuing signal processing applications [4]. Because of the throughput requirements and real-time constraints, pipeline architectures were popular as well.

As the 1980's approached, single-chip digital signal processors started to become available. A few of these were dedicated circuits intended for special applications. An early example would be a speech synthesis chip developed by Texas Instruments for use in a toy known as the "Speak-and-Spell" [5]. Of more

interest to signal processing researchers was the availability of general purpose, programmable DSP chips such as those of N.E.C. [6], Bell Laboratories [7], and Texas Instruments [8]. Architecturally, these circuits all followed a standard formula. The instruction set resembles that of a conventional microprocessor. Hardware organization is also conventional, with some pipelining added and a parallel multiplier attached.

Despite heavy promotion, these general purpose processors have failed thus far to achieve widespread use. One probable reason for this is that the general-purpose architecture is seldom a good match for any given application. Such a circuit offers a fixed combination of resources (program memory, data memory, arithmetic capacity) which can not be tailored to suit the designer's needs. The inclusion of a large parallel multiplier results in an expensive part in each case. Although there is a definite need for multiply-accumulate operations in a typical signal processor, it is difficult to make efficient use of the parallel multiplier when programming the circuit to perform a complex system function.

Because of the problems described above, a number of researchers have worked on the problem of designing semi-custom digital signal processing IC's [9-11]. The goal of this work is to combine the cost-effectiveness of a full custom design (such as the vocoder chip of Chapter 2) with the short design times possible using general purpose hardware. The underlying assumption of such work is that similar applications can be efficiently implemented using different configurations of the same hardware.

Semi-custom design approaches have a target architecture to which individual designs must conform. There are several types of target architectures which have been proposed. Three general categories may be identified: (1) processors built around a parallel array multiplier; (2) bit serial processors; and (3) bit-parallel processors without parallel array multipliers.

The parallel multiplier approach is often regarded as the best signal processing solution. This is only partially true. Parallel multipliers by necessity must be constructed with a fixed precision, e.g. a 16 by 16 multiplier. They are therefore best suited for algorithms which make heavy use of multiplies, all of which require about the same precision. Unfortunately, precision requirements in signal processing algorithms vary considerably. Many multiplies require only a low precision coefficient, such as a power of two. The parallel multiplier is in these cases underutilized. Also, there is the architectural problem of getting data to and from the parallel multiplier without bottlenecks occurring. The cost, in terms of busses and registers, is high. In summary, a parallel multiplier is a high cost, high throughput hardware item that fails to see good utilization in actual practice.

The second option, bit serial designs [12], are the opposite end of the architectural spectrum. In an effort to maximally utilize the logic elements of the circuit, bit-level pipelining is used. High clock rates are needed to reap the full advantage of this approach, creating a particularly difficult low-level design problem. An advantage is that bit-serial functional units can be combined in a modular fashion, a plus if the target system is a dedicated processor. One drawback of this approach is that a number of low level functions (such as division) are not compatible with the LSB-first bit-serial format. Due to the hard-wired nature of the systems, control and decision-making functions are not easily implemented. Also, a system-wide wordlength is generally established for all data, in an effort to avoid intractable timing problems. This is a disadvantage, since all data must be represented by words of the same precision.

The third option, bit-parallel processors without parallel multipliers, is the one used for the designs described in this dissertation. An arithmetic unit which can complement, shift and accumulate bit-parallel data is used. This unit may be microprogrammed to multiply signal data by either bit-serial or signed-digit

coefficients. (Signed-digit representation provides a low-cost method for fixed coefficient multiplies, as discussed in Section 3.5.) Because the arithmetic element is relatively compact, several such processing units may be included on a single IC. Thus the advantage of modularity exhibited by the bit-serial approach is retained here. This results in a better match between algorithm and hardware organization than is possible in designs using large parallel multipliers.

The bit-parallel processors are microprogrammed, resulting in a versatility lacking in hard-wired approaches. Although fully hard-wired architectures might be more efficient in the abstract, programmability is a virtual necessity in a complex system. Control, decision-making and input-output functions are all more readily implemented in a programmable environment.

1.4 Design Methods for Digital Integrated Circuits

Numerous approaches, systems and methodologies have been proposed or are in use for integrated circuit design. One reason for this is that, viewed in the abstract, integrated circuit design is a large process with a high level of complexity and many degrees of freedom. Many methodologies seek to constrain the design problem, to make it less complex while still achieving performance goals.

IC designs are ultimately specified at the geometric level on a number of mask layers. In a *full-custom* design, the designer has complete control over these low-level geometries. Standard cell and gate-array design methods involve specifying the design at a logical level, and creating the mask geometries from a predefined set of primitive elements. Thus standard-cell and gate-array approaches constrain the design problem to make it more tractable.

As a general rule, the more constrained the design problem is made, the more easily a design may be produced. However, this design can not be optimized to the

same extent as a full custom design.

Many IC design systems are intended to fill a certain market need. For example, a standard-cell system might be intended to produce circuits which are replacements for systems designed from a standard logic family (for example, the 74-series TTL family). Such systems have the severe drawback that the designer is using logic designs and architectures appropriate for one technology (TTL-MSI) while producing a circuit in a different technology (MOS-LSI).

In addition to the degree of customization, a distinction may be drawn between graphical and procedural methods of specifying IC layouts. In the graphical approach, the designer specifies the mask geometries directly, usually with the aid of a computerized graphics editor. In the procedural approach, the specification is at a higher, functional level; software is used to translate the higher-level specification to the mask level.

The LPC vocoder circuit described in Chapter 2 was designed with purely graphical methods. This approach resulted in a high degree of customization, and many opportunities to optimize the design. Since no procedural methods were employed, the design time was high, and the overall approach would not have been suitable for applications with much greater complexity. Architecturally, however, the methods used in the vocoder IC are applicable to even larger, more complex designs. This conclusion is the motivation for the macrocell-based design system described in Chapters 3, 4 and 5.

The term *macrocell* has been assigned various meanings in the past. It is now generally agreed that a macrocell is a large block of circuitry containing perhaps several thousand transistors. The approach used in assembling the macorcells is to tile (array) smaller rectangular cells in two dimensions. The tiling process is sufficiently flexible that the macrocells can be customized for specific tasks.

One premise of the macrocell technique is that a target application range has

been identified. This dissertation is concerned solely with digital signal processors. Typical applications would be speech processing, telecommunications and digital audio. However, the macrocell approach may be applied to any group of applications. The more narrow the application range, the more the macrocell-based designs will resemble full-custom designs in terms of architectural efficiency.

Since MOS processes evolve rather rapidly, a desirable property of an IC design system is that it can be applied to a number of different processes. Macrocell based systems can be designed such that only the underlying cell library is technology-dependent.

In summary, the macrocell approach has the following advantages:

(1) Large blocks of circuitry are generated by procedural methods.

(2) The underlying cell library is designed by graphical methods.

(3) Architectures can resemble those of full-custom designs.

(4) The design system may be ported to different technologies.

Chapter 2 -- A Single-Chip LPC Vocoder

2.1. Introduction

A number of applications benefit from the fact that, with sufficient processing, speech may be analyzed, encoded at a low data rate, and then resynthesized. For example, the available bandwidth of a telephone line or microwave channel may be better utilized if data compression methods are employed. In "voice mail" networks, where spoken messages are stored electronically and subsequently forwarded to their destination, data reduction reduces the storage cost.

In many applications, digital encoding of speech is needed to allow the use of encryption algorithms (for security) or error-correction techniques (to compensate for a noisy channel). Often, the available bandwidth is that of a voice-grade analog channel -- a few kilohertz. In situations such as these, it is necessary to digitally encode speech at a low enough data rate that the digital data may be transmitted in the available bandwidth.

Low-bit-rate coding systems, where the transmission rate is 2400 bits/second or lower, cannot attempt to reproduce the speech waveform itself. Instead, a set of slowly-varying parameters is extracted from the speech. Synthetic speech may be generated from these parameters. A vocoder is a system of this sort based on a model for speech synthesis similar to that shown in Fig.2.1.

This model, roughly analogous to the human vocal tract, consists of an excitation source driving a time-varying filter. The excitation source is described by three parameters: voicing, pitch, and amplitude. The voicing parameter represents a binary decision between a voiced, vowel-like sound and an unvoiced sibilant. The pitch represents the fundamental period of a voiced sound.

The excitation signal presented to the filter has a flat overall spectrum. The

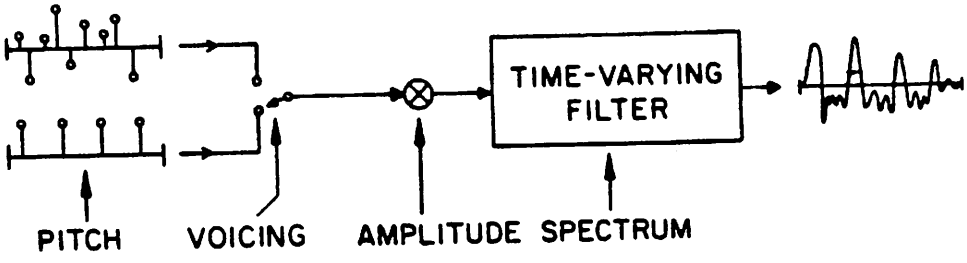


Fig.2.1 Vocoder speech model

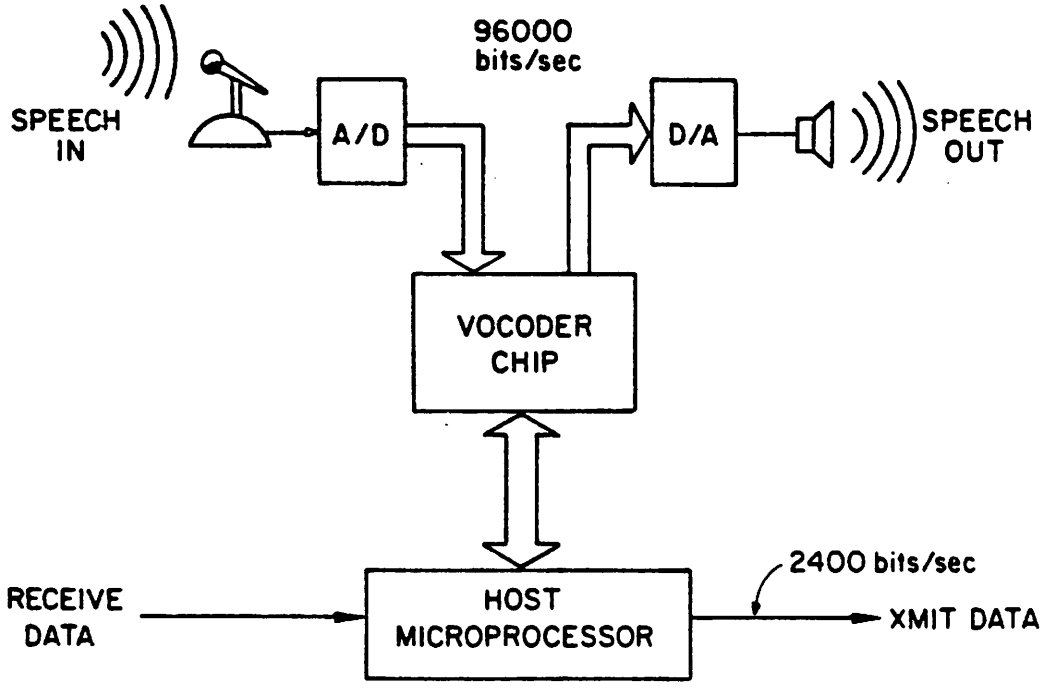


Fig.2.2 Target system for the vocoder I.C.

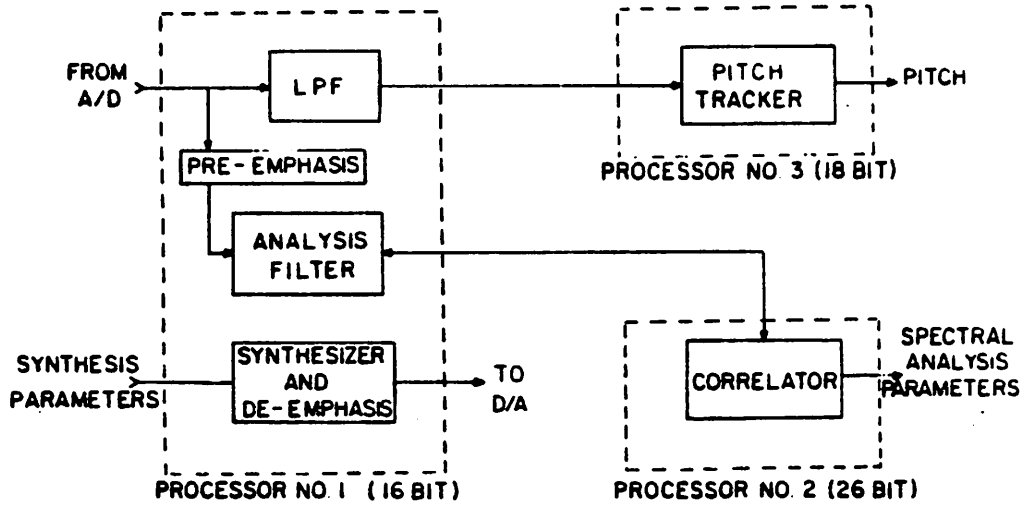


Fig.2.3 Multiprocessor organization

filter is programmed to impart the desired time-varying spectral shape to the signal. Since a representation of this spectrum is part of the parameter set, vocoders must perform some form of spectral analysis as part of their processing. Accurate preservation of the spectral shape of the speech signal is important in maintaining intelligibility.

Because of the importance of spectral representation, vocoders are often classified according to the spectral analysis methods used. *Channel vocoders* use either filter-bank or transform techniques to represent the energy in different parts of the spectrum. This report describes an *LPC vocoder* circuit using linear predictive coding (LPC) to represent the spectrum.

Fig.2.2 shows how the vocoder circuit would be used in a typical application. Digitized speech (from an A/D converter) is used as input to the device. The input speech is analyzed and the resulting parameters are transferred off-chip to a host microcomputer. The host formats the data for transmission (typically at 2400 bits/second). Simultaneously, an incoming data stream is transferred to the vocoder circuit, where it is used to generate a synthetic speech signal. The vocoder thus operates in full-duplex mode, allowing two-way low-bit-rate speech transmission.

To handle the complexity needed for a complete LPC vocoder algorithm, a monolithic multiprocessor approach was used. Three processors are included on the single I.C. Each processor performs a specific part of the vocoder algorithm, as shown in Fig.2.3.

Processor No. 1 performs most of the linear filtering of the speech signals that is required. Included are lattice filters for both spectral analysis and synthesis; pre-emphasis and de-emphasis networks; and a low-pass-filter whose output is used for speech pitch detection.

Processor No. 2 computes correlation values from signals in the analysis lattice

filter. These correlation values are used for spectral estimation as described in Section 2.4.

Processor No. 3 extracts pitch and voicing information from the input speech.

The algorithm is partitioned so that each processor is fairly self-contained. Communication among the processors is performed over a small number of bit-serial data lines. This avoids the need for large parallel busses between processors. The use of multiple, dedicated processors and bit-serial communication contributes to the area efficiency of the circuit.

2.2 Processor Architecture

Vocoder algorithms require a fairly diverse set of operations in their realization. Because of this a microprogrammed approach was used in the vocoder IC. Each processor contains a read/write memory and a simple arithmetic unit which, under control of a microsequencer, is programmed to perform the operations required. Among the operations used are multiply-accumulate, division, absolute value (rectification), and comparison. These operations may be combined to create complex functions such as filtering, correlation, and pitch extraction.

Multiply-accumulate operations are a common feature of signal processing algorithms. Examples are the programmable lattice filters used in both the spectral analysis and speech synthesis portions of the vocoder. The multiply-accumulate operations are implemented by programming the arithmetic unit shown in Fig.2.4. This involves multiplying signal data which is internal to the processor's read-write memory by externally available coefficients. The sample delays are implemented by providing the appropriate address sequence to the memory.

Data in the read-write memory and the arithmetic unit are represented as parallel two's complement words. The word length is optimized for each of the

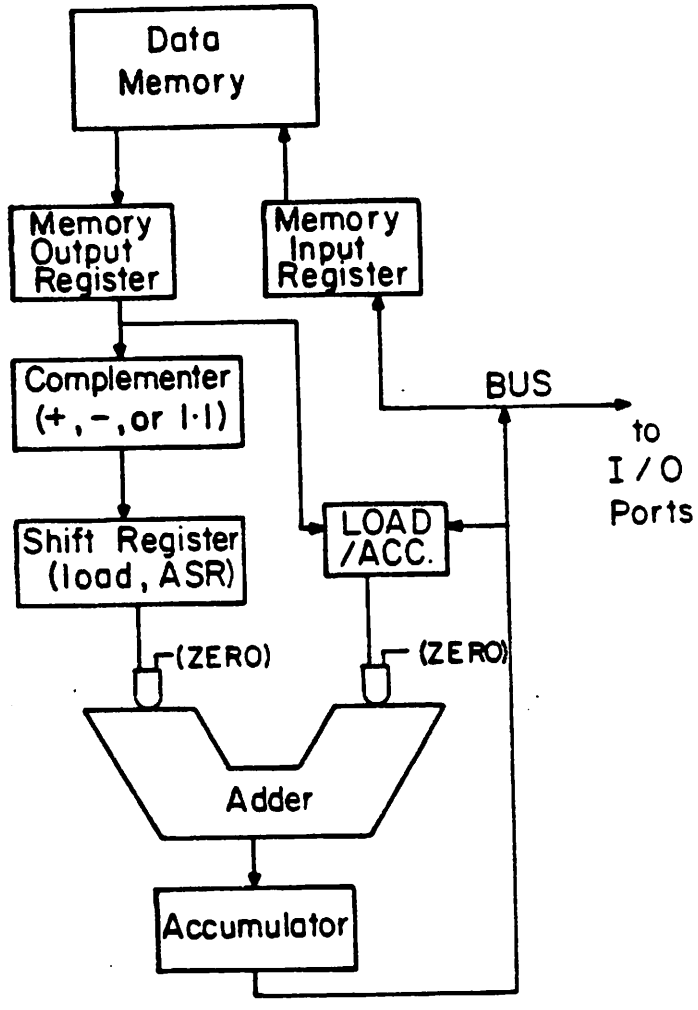


Fig.2.4 Arithmetic unit block diagram

three processors. Processor No.1, used for linear filtering of the speech signal, has a 16-bit word.

The arithmetic unit contains three pipeline registers (MOR, SR, and ACC) and a transparent latch (MIR). Pipelining allows concurrent memory access, invert/shift, and accumulate operations, resulting in good throughput for a relatively compact circuit. Operation is best understood by the action of microinstructions on the contents of these registers.

The memory output register (MOR) is loaded each cycle with the result of a memory access. The accessed data is available the following cycle. There are no control options.

The shift register (SR) may be either loaded or shifted right arithmetically. When loading the shift register, the MOR is used as input, and may be optionally complemented. Options include true, complement, or absolute-value data; and complementing by the sign bit of an externally available coefficient bit. The following symbolic microinstructions describe these actions:

```
sr := sr/2
sr := mor
sr := -mor
sr := |mor|
sr := mor.coef
```

A number of different options are available as input for the accumulator. The B-inputs of the adder may be set equal to zero, ACC, or MOR. The A-inputs may be set equal to zero, SR, or the contents of SR gated with a bit from an external coefficient. These options are described by the following microinstructions:

```
acc := 0
acc := mor
acc := acc

acc := sr
acc := sr + mor
acc := sr + acc

acc := coef*sr
acc := coef*sr + mor
acc := coef*sr + acc
```

The memory input register (MIR) is a transparent latch which has as input either ACC or an external signal from the I/O bus. (The external signal may be bit-serial data originating from one of the other two processors on the chip which has been converted to bit-parallel form.) The MIR may be loaded or held under program control. When loaded, the new data is available the same cycle. Loading is indicated by the "load enable" microinstruction:

```
load_en
```

The purpose of the MIR is to hold temporarily a value which needs to be stored in the read/write memory. By holding such a value until the next free memory cycle, memory access conflicts are avoided.

Microinstructions are also used to address and control the read/write memory. All told, a horizontal control word of approximately 24 bits is used for controlling each processor.

In much of the processing needed for the vocoder algorithm, the most prevalent operations are multiply-accumulates by a variable coefficient. To allow this, the coefficient is made available in a bit-serial, sign-magnitude, MSB-first format. The multiply is performed by first complementing the data with the sign bit of the coefficient, then accumulating a sequence of right-shifted partial products which are gated with the magnitude bits of the coefficient.

The following microcode fragment performs a multiply-accumulate. The coefficient is eight bits. The data is first read into MOR, then microinstructions are executed as shown below. Because of the pipelining, microinstructions controlling SR and ACC are given on the same line and occur on the same cycle.

```
sr := mor.coef      acc := acc
sr := sr/2          acc := sr*coef + acc
sr := sr/2          acc := sr*coef + acc
sr := sr/2          acc := sr*coef + acc
sr := sr/2          acc := sr*coef + acc
sr := sr/2          acc := sr*coef + acc
sr := sr/2          acc := sr*coef + acc
                  acc := sr*coef + acc
load_en
```

At the end of this sequence, MIR contains the result of the multiply added to the prior contents of ACC. The "single-accumulator" architecture allows the same accumulator register to be used for summing partial products and accumulating the results of a series of multiplies. This parallel-serial approach to multiplication results in a low-cost implementation of the multiplies required for the lattice filters used in the vocoder.

The adder element of the arithmetic unit employs saturation logic, which limits the result of an addition to the maximum positive or negative value upon positive or negative overflow respectively. Among other advantages, this rules out the possibility of large-amplitude limit cycles in recursive filters.

The processor arithmetic unit described above is based on a single accumulator, performing multiply/ accumulate operations by a microprogrammed, "shift and add" method. The small circuit area occupied by this arithmetic unit is a key feature allowing multiple processors and high throughput in the vocoder IC.

2.3 Processor Implementation

In order to easily implement processor arithmetic units with different word lengths, a bit-slice organization for the arithmetic unit was used. Due to the fact that the arithmetic unit bit-slice is replicated many times in the three processors that comprise the vocoder circuit, considerable effort was spent making this element compact and high-performance. The schematic for a single bit-slice is shown in Fig.2.5. (This schematic is slightly simplified; part of the memory interface, and the adder saturation logic, are not shown.)

Each bit-slice contains 65 transistors and occupies an area of about 100 sq. mils. This small size makes possible the inclusion of multiple processors in the vocoder I.C. By way of contrast, a single 16 x 16 array multiplier (often thought of as an essential element of a digital signal processor) would exceed in area all three of the arithmetic units on the vocoder circuit.

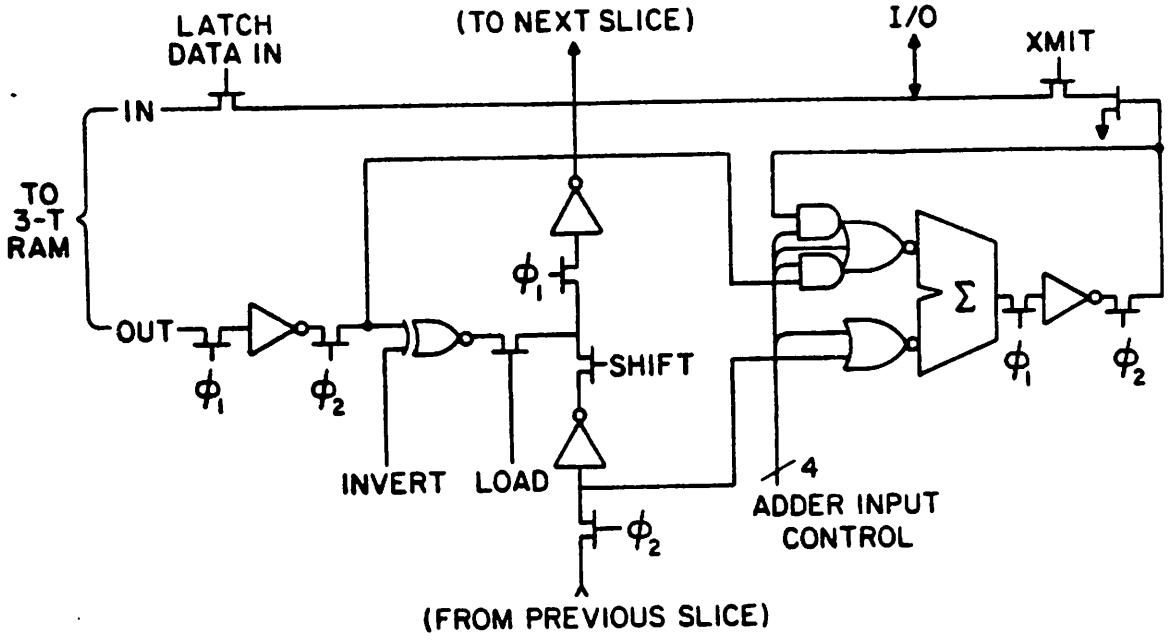


Fig.2.5 Arithmetic unit bit-slice schematic

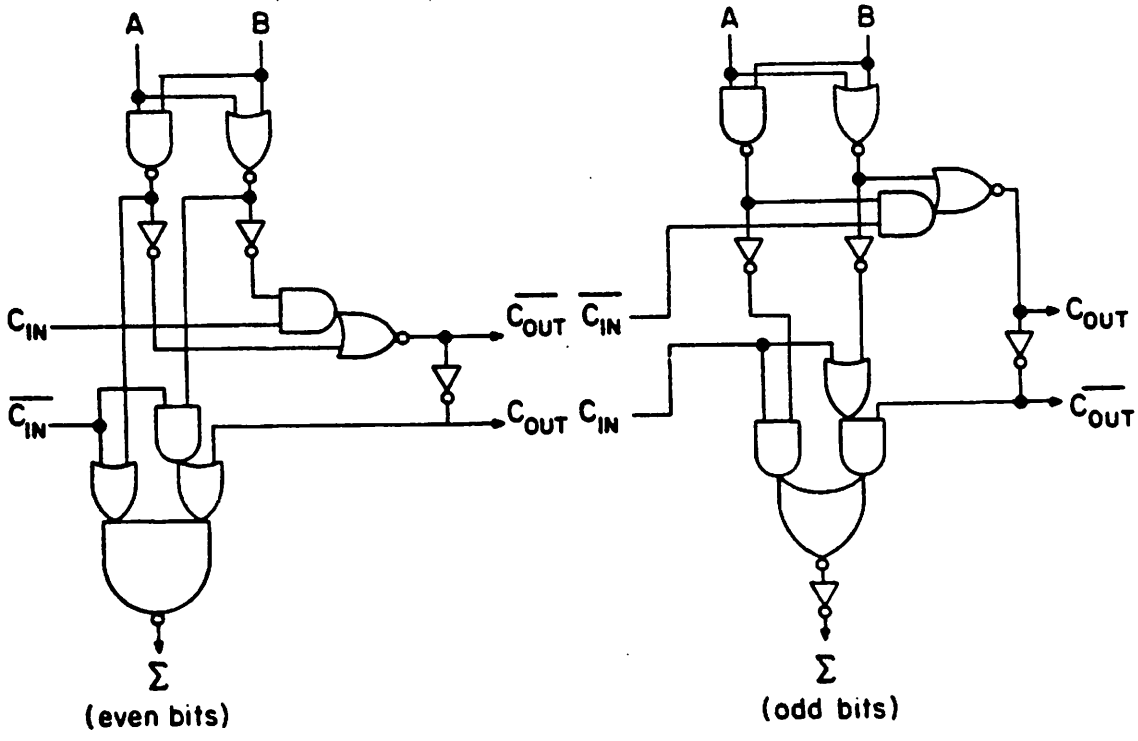


Fig.2.6 Adder schematic

Because of the bit-slice organization, the adder circuit used in the arithmetic unit is a ripple-carry design. Separate circuits were designed for the odd-numbered and even-numbered bit slices, as shown in Fig.2.6. Layout and circuit design were carefully optimized to minimize nodal capacitances along the carry chain. Since carry outputs are used in the sum generation logic, each carry output feeds a minimum-size inverter which isolates this logic from unnecessarily loading the carry chain.

The use of separate odd and even adder circuits, alternating between positive and negative logic, provides a single level of gate delay for each bit in the carry chain. Add times under 200 nsec. for 26 bits were typically measured. The performance of the adder is well-matched to other critical paths in the vocoder circuit, such as RAM and ROM access times, and does not by itself limit the computation rate.

The processor data memory uses the three transistor cell, chosen because of its relative insensitivity to process parameters and supply voltage variation. As is the case with word lengths, the number of words in the data memory of each processor is chosen to match that processor's function. The data memory is laid out on a more narrow pitch than the arithmetic unit bit-slice, with the memory bit-lines routed to the bit-slice array.

Additional flexibility is provided by allowing some of the memory locations to be programmed with read-only constants. This is needed in the implementation of the pitch tracker algorithm discussed in Section 2.5. In this algorithm, it is necessary to compare signals to constant thresholds. Note that the approach of directly including constants in the data path makes inclusion of a gateway between control and data paths unnecessary.

Each processor contains a small amount of control circuitry along one edge of the bit-slice array. This circuitry buffers the control signals that the

microsequencer sends to the processor, and allows external coefficients to be multiplexed into the control path as discussed in Section 2.2. Circuitry is also included that allows for long division operations in a manner similar to the parallel-serial multiply operations discussed above.

2.4 Adaptive Lattice Analyzer

2.4.1 Lattice LPC Algorithm

Preservation of spectral information is essential to any speech coding technique. One approach to spectral analysis models the speech samples as a random process whose second-order statistics are stationary over short time intervals. This leads to the LPC (linear predictive coding) method [13], so named because it involves predicting the value of an incoming sample as a linear combination of previous samples. The number of previous samples used in the linear combination is referred to as the order of the analysis. For speech coding, a tenth-order LPC analysis is often used [14].

The spectral information that results from an LPC analysis can be represented in one of several equivalent forms: predictor coefficients, normalized autocorrelation values, log area ratios, and reflection coefficients [15]. Each of these forms may be translated into any of the others, so each is a possible choice as part of the parameter set for a vocoder. The reflection coefficients (also known as partial correlation values) have the following practical advantages:

(1) The spectrum is relatively insensitive to quantization errors of the reflection coefficients.

(2) The reflection coefficients may be computed by an adaptive filter structure

known as the adaptive lattice analyzer, described below.

(3) The reflection coefficients may be used to program an all-pole filter (described in Section 2.6) which is used for resynthesizing vocoded speech.

A typical speech coding format is the 2400 bit/second LPC-10 format, which transmits a set of ten reflection coefficients every 22.5 milliseconds [16]. Thus it is convenient to use an LPC analysis method that yields these reflection coefficients directly, rather than first computing one of the other equivalent LPC parameter sets and converting to reflection coefficients.

The adaptive lattice analyzer contains two parts: an all-zero programmable lattice filter (Fig.2.7) and a correlator (Fig.2.8). The lattice filter contains ten identical stages, each of which has two inputs A_{i-1} and B_{i-1}. These inputs are fed to the correlator, which computes k_i as the normalized cross-correlation of A_{i-1} and B_{i-1}. For stationary input signals, A_{i-1} and B_{i-1} have the same energy

$$EA_{i-1}^2 = EB_{i-1}^2$$

(where E is the expected value operator). Burg [17] first proposed using the average of these two energies for computing the normalized cross correlation

$$k_i = \frac{2EA_{i-1}B_{i-1}}{EA_{i-1}^2 + EB_{i-1}^2}$$

The k_i are limited to values between -1 and +1. The vocoder IC computes the k_i to a precision of eight bits. This exceeds the precision requirements for commonly used 2400 bit/second transmission formats.

The outputs of the correlator are the k_i's, which are then used to program the lattice filter. Each k_i is recomputed every sample interval, allowing the lattice analyzer to adapt continuously to the changing statistics of the input signal.

The correlation (Fig.2.8) uses a single pole low-pass filter to approximate the expected value operator. Kang [18] claims a small performance improvement if a two-pole filter is used here.

The output A_{10} of the tenth-order lattice analyzer is known as the residual, and can be thought of as the input signal filtered so as to remove most of the correlation between samples. In the frequency domain, this means the residual will have a spectrum that is close to flat. The spectral information that has been removed is contained in the k_i.

This effect is illustrated in Fig.2.9. These photographs were obtained by applying a periodic input signal to the input of the vocoder IC's lattice analyzer, and monitoring the signal A_i at different stages in the filter. These signals appear on the data bus of Processor No. 1 and were strobed into a D/A converter to obtain the data. The input signal to the lattice analyzer has a periodicity of 200 Hz, and exhibits a strong resonance at about 1 kHz. This resonance is apparent from the ringing in the time response. As the signal travels through the lattice this resonance becomes less pronounced. The residual output of the final stage resembles an impulse train, which has a flat overall spectral shape. The fine structure of the spectrum is preserved in the residual. In this illustration, the spectral fine structure is characterized by a concentration of spectral energy at harmonics of the 200 Hz fundamental. Although this fine structure is not extracted by the lattice analyzer, in an LPC vocoder some of this information is extracted by performing a pitch analysis (described in Section 2.5).

In the vocoder IC, the lattice analyzer is preceded by a pre-emphasis filter with the transfer function $1-\frac{15}{16}z^{-1}$. The pre-emphasis serves to improve the

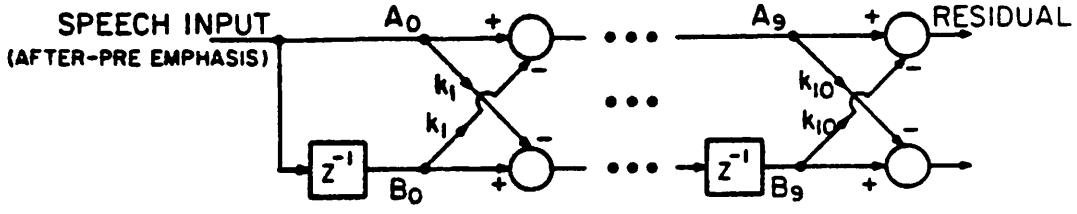


Fig.2.7 Analysis lattice filter

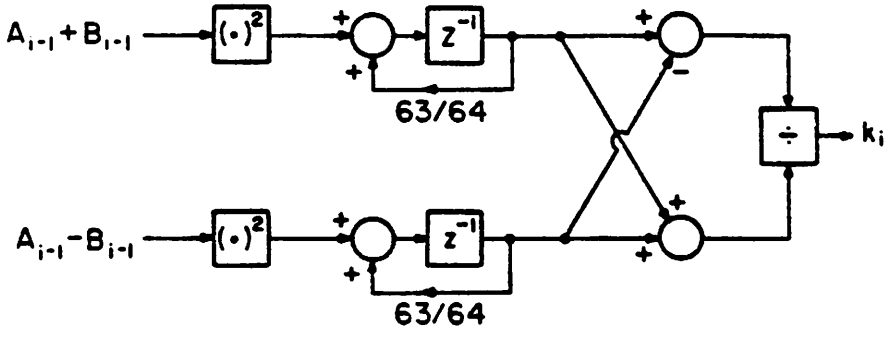
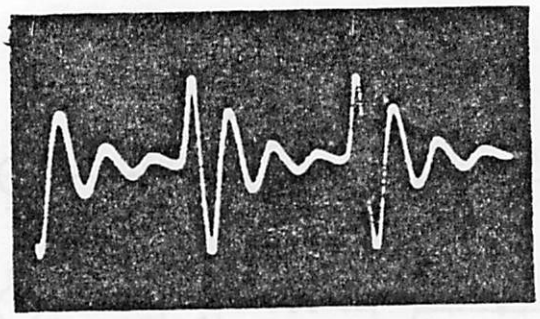
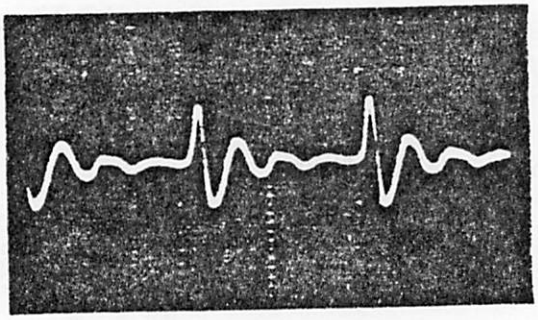


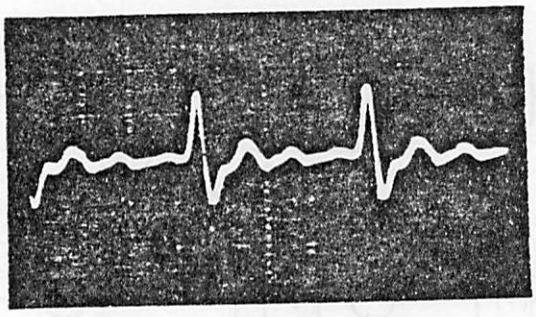
Fig.2.8 Correlator



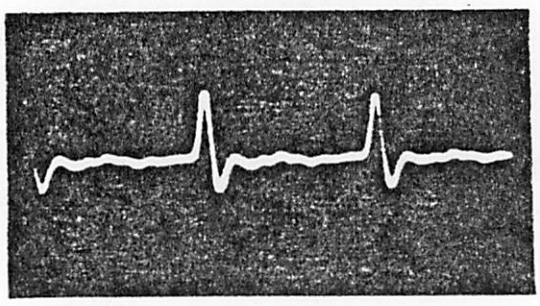
(a) Input



(b) Second stage output



(c) Fourth stage output



(d) Tenth stage output (residual)

Fig.2.9 Signals at different points in lattice analyzer

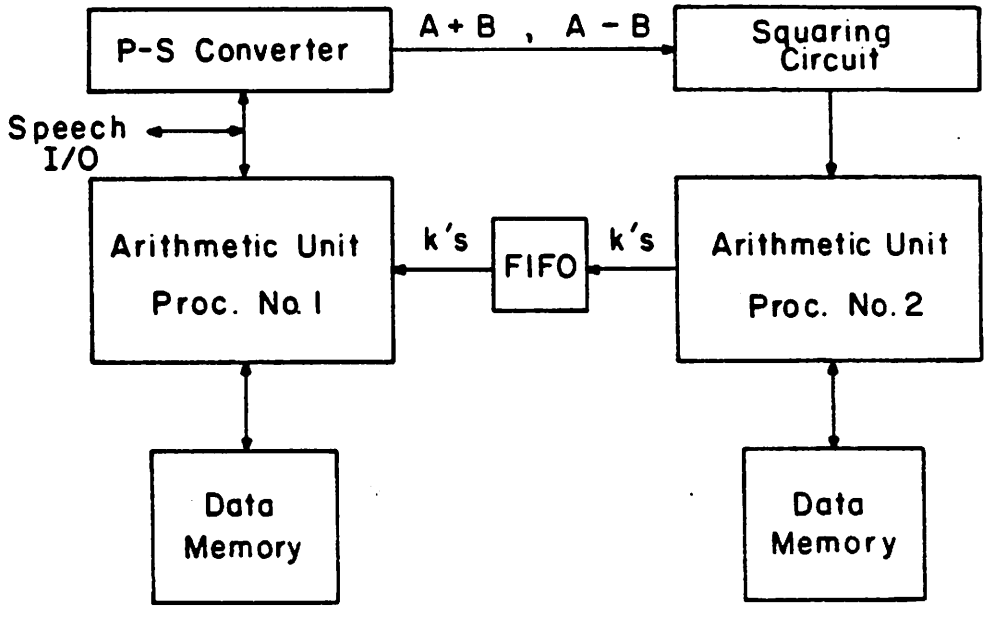


Fig.2.10 Configuration of Processors 1 and 2

distribution of the reflection coefficients.

2.4.2 Lattice Analyzer Implementation

Implementation of the lattice analyzer was the dominant issue when studying appropriate architectures for the vocoder IC. The lattice filter itself requires variable-coefficient multiplies, while the correlator requires a squaring operation, fixed-coefficient multiplies, and a two-quadrant division (two-quadrant because the denominator is always positive). Both lattice filter and correlator require single-sample delay operators. The processor structure described in Section 2.2 is characterized by a single-accumulator arithmetic unit and an addressible data memory. This processor may be programmed to perform the multiply-accumulate, divide, and delay operations required. Combined with a look-up table that approximates the squaring operator, this processor becomes a suitable architecture for the lattice analyzer.

The computation for the lattice filter is divided between two processors. Fig.2.10 shows the data path organization for these two processors. Processor No. 1 implements the pre-emphasis filter and the ten lattice filter stages. Processor No. 1 also implements the ten-stage all-pole lattice synthesis filter described in section 2.6. The microcode for a single stage of each ten-stage filter is contained in a subprogram, which is iterated ten times per sample interval to implement the two filters. The subroutine also computes the values $A_{i,-1}+B_{i-1}$ and $A_{i,-1}-B_{i-1}$, which are inputs to the squaring look-up table. These two values are transmitted over bit-serial lines to the squaring circuit, which is physically located adjacent to Processor No. 2. Processor No. 1 has a 16 bit wordlength.

The input to the squaring circuit is truncated to 12 bits unsigned, and the 20 most significant bits of the squared output are preserved. This output is used as

input to Processor No. 2, which has a 26 bit wordlength. A longer wordlength is required for Processor No. 2 since squaring a signal doubles its dynamic range.

Processor No. 2 is programmmed with the two single-pole estimation filters and the divide operation. These operations form a subroutine which is iterated ten times per sample interval, computing the ten reflection coefficients k_i. The k_i appear as a bit-serial output signal resulting from the long division operation. This bit-serial signal is routed back to Processor No. 1.

The adaptive lattice analyzer requires half the resources of Processor No. 1 and all of Processor No. 2. This amounts to half of the resources of the three-processor vocoder IC being devoted to the important task of spectral analysis.

2.5 Pitch Tracker

2.5.1 Pitch Tracker Algorithm

In addition to the spectral analysis described in the section above, the speech input of the vocoder must be analyzed for excitation parameters. Referring to the parameter set shown in Fig.2.1, the pitch period and voiced/unvoiced decision are both provided by implementing a pitch tracking algorithm developed by B. Gold [19]. With only minor changes, the algorithm used is referred to in [19] as the "second modification" of the original Gold Pitch Tracker.

In contrast with the algorithms for lattice analysis and synthesiser, the pitch tracker requires a variety of non-linear and decision-making operations. These include peak detection, Boolean arithmetic, comparing signals to constant thresholds, incrementing and resetting counters, and selecting the maximum among a set of signals.

This algorithm is shown diagramatically in Fig.2.11. The input speech is first

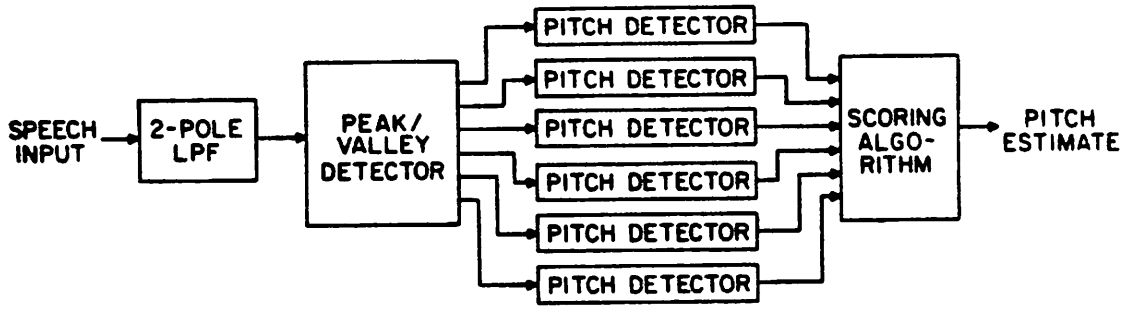


Fig.2.11 Gold Pitch Tracker algorithm

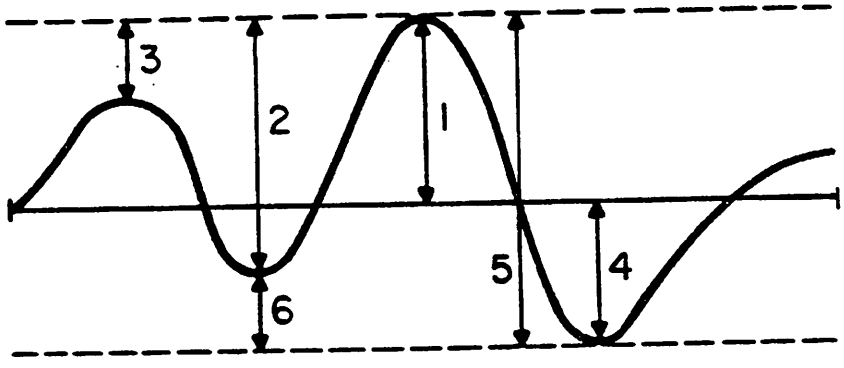
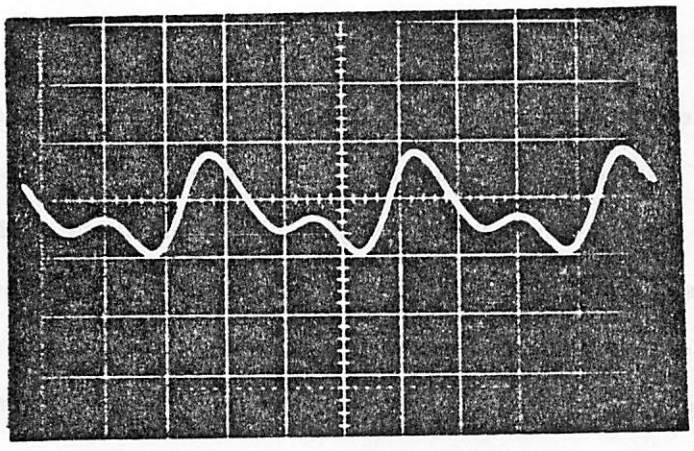
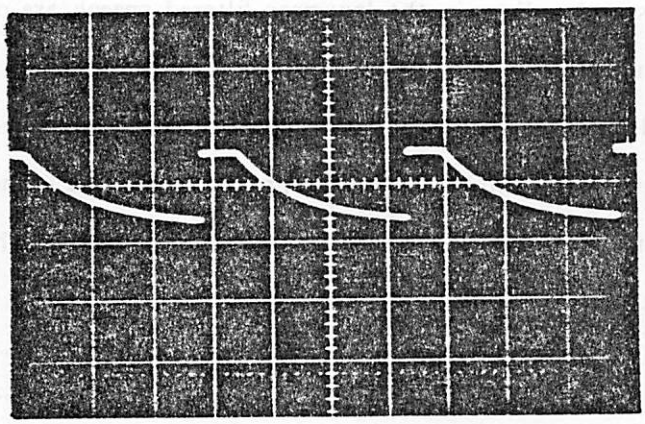


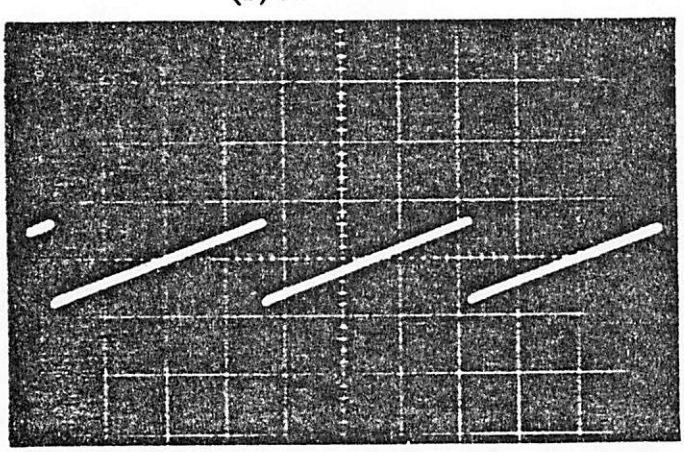
Fig.2.12 Six signals formed after peak-valley detection



(a) Input



(b) Threshold



(c) Pitch Period Counter

Fig.2.13 Signals from one of the six pitch detectors

low-pass filtered to a bandwidth of a few hundred Hertz. The fundamental pitch frequency is assumed to lie within this band. However, harmonics of the fundamental may also pass through the low-pass filter. The intent of the algorithm is to reject these harmonics and extract the fundamental period. The 2-pole linear low-pass filter is implemented by a short section of microcode in Processor No. 1. The resulting signal is then sent over a bit-serial line to Processor No. 3, which performs the remainder of the Gold algorithm.

First, peaks and valleys in the low-pass filtered speech are detected. Values for the amplitudes of the most recent peak and previous valley are maintained. Six new signals are formed, each being a different combinations of the current input sample (CS), the amplitudes of the previous peak (LP), and the amplitude of the previous valley (LP). With proper interpretation, these six signals correspond to the six quantities indicated in Fig.2.12:

signal	computed as	examined at	interpreted as
1	CS	input peaks	magnitude of peaks
2	(CS-LV)/2	input peaks	difference in magnitude between peaks and the preceding valleys
3	(CS-LP)/2	input peaks	difference in magnitude between adjacent peaks
4	-CS	input valleys	magnitude of valleys
5	(LP-CS)/2	input valleys	difference in magnitude between valleys and the preceding peaks
6	(LV-CS)/2	input valleys	difference in magnitude between adjacent valleys

(The values of these signals are ignored except at input peaks and valleys as shown above.)

The six signals are sent to six identical pitch detectors. These pitch detectors each form a estimate of the time interval (period) between major peaks in their inputs. A basic premise of the Gold algorithm is that of parallel processing: each individual pitch detector forms a pitch period estimate which is not in itself reliable. However, by combining the six estimate a reliable pitch estimate is obtained.

Each pitch detector functions by timing the interval between major peaks in its signal. To accomplish this, a means of rejecting minor, insignificant peaks is required. The decision as to whether the input is at a major input peak is based on three criteria:

(1) The peak detector input must be at a peak or valley as given in the above table.

(2) After a major peak is detected, a blanking interval of three milliseconds (24 samples) duration is entered during which all peaks are rejected.

(3) Following the blanking interval, an exponentially-decaying threshold signal is computed. This threshold is initialized with the amplitude of the previously detected peak, and decays with a time constant of five milliseconds. Minor peaks which fail to exceed this threshold are ignored.

Fig.2.13 illustrates the function of one of the six pitch detectors. These photographs were obtained by strobing various signals appearing on the data bus of Processor No.3, into a D/A converter. The top photo (a) is the input to the pitch detector. In this example there is a strong second harmonic which the pitch detector succeeds in rejecting. Photo (b) shows the threshold signal, with the blanking interval and exponential decay in evidence. The sawtooth waveform of the bottom photo (c) is the value of the pitch period counter. This counter is sampled prior to being reset to give a pitch period estimate.

In this way six estimates of the pitch are obtained. Each estimate is regarded as a candidate for possibly being the actual pitch period. A scoring algorithm is used to select one of the six as the best estimate. Each candidate is given a score

ranging from one to eighteen by performing a "window comparison" between the candidate and each of the following eighteen values:

The six current pitch estimates (i.e., the candidates themselves);

The previous pitch estimates from each of the six detectors;

The sum of the current and previous estimates for each detector.

(The rationale for including the sum of consecutive estimates from a pitch detector is that a pitch detector often locks on to the second harmonic rather than the fundamental. In this case the sum of two consecutive estimates is the period of the fundamental.)

If none of the estimates has a score greater than a fixed threshold the input speech is determined to be unvoiced and no pitch estimate is outputted.

2.5.2 Pitch Tracker Implementation

The pitch tracker microprogram is organized into a main program followed by six iterations of a subprogram. The entire microprogram repeats each sample interval. each input sample. The main program performs the peak/valley detection and part of the scoring algorithm. The subprogram implements the pitch detectors and most of the scoring computation.

The main program performs the following functions:

(1) Maintains the variables LP and LV (last peak and valley)

(2) Forms the six pitch detector input signals

(3) Decides if one of the six pitch estimates (the "current candidate") has the highest score so far. The current candidate may change from sample to sample.

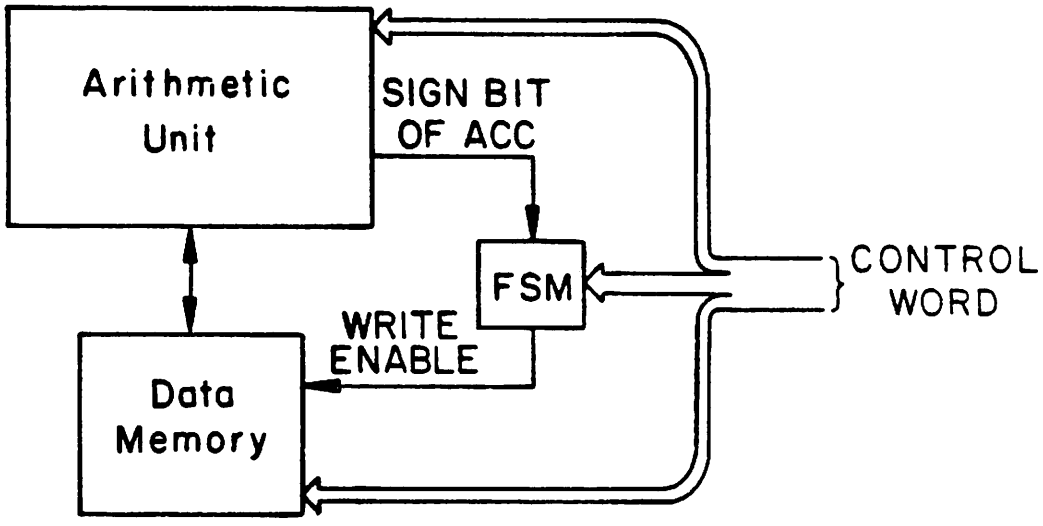


Fig.2.14 Processor No. 3 with FSM attached

(4) Every six samples, makes a pitch/voicing decision and transmits either the winning pitch estimate, or zero if unvoiced.

Each subprogram iteration performs the following functions for one of the six pitch detectors:

(1) Implements the pitch detector by maintaining the threshold level, the pitch period counter, and values for the current and previous pitch estimate.

(2) Adds the pitch detector's contribution to the score of the current candidate.

By computing the score for one candidate each sample, the pitch tracker makes a new pitch determination every six samples. Note that decimation is involved here. One could compute all six scores every sample interval, selecting a new value for the pitch estimate each time. However, a vocoder system does not require pitch estimates this frequently. Thus one can reduce the computational requirements by decimating to a lower sample rate for the scoring computation.

Implementing the algorithm above using the arithmetic processor that has been described requires additional hardware to support the decision-making and conditional operations required. This is in contrast to the spectral analysis and synthesis parts of the vocoder algorithm, which involve little or no conditional logic. To support the need for decision-making capability, two hardware features were included in Processor No. 3: a small finite-state machine (FSM), and a special conditional write operation. Fig.2.14 shows how this FSM is attached to the data and control paths for Processor No. 3.

The FSM is customized to perform the exact logical operations that are needed in the pitch detection algorithm. The FSM has three bits of state:

CC	Condition Code
SLP	The slope of the input signal
LSLP	The value of SLP for the previous sample

The CC (condition code) bit must be true for a conditional write (WC) instruction to effect a write cycle.

The inputs to the FSM are control signals from the microsequencer and the sign bit of the accumulator. (In the following, SIGN is true if the accumulator is non-negative.) These seven microinstructions were used to modify the state of the FSM:

instruction	action
SET	CC := SIGN
AND-	CC := CC and (not SIGN)
SSL	LSLP := SLP; SLP := SIGN
SIP	CC := LSLP and (not SLP)
SIV	CC := SLP and (not LSLP)
SPV	even subroutine iterations: functions as SIP
	odd subroutine iterations: functions as SIV
VPE	"valid pitch estimate" (true every six samples)

The FSM instructions are executed concurrently with the data path instructions (described in section 2.2.) for Processor No. 3.

The SET and AND- instructions allow comparisons to be made. The SSL instruction is executed once per sample, after loading CS-LS into the accumulator. This sets the boolean variables SLP and LSLP to indicate the slope of the input

signal the slope of the input at the current and previous sample instant respectively. SLP and LSLP can then be decoded to indicate whether the input is at a peak or valley. This allows implementation of the SIP, SIV and SPV instructions for the FSM.

The conditional write instruction assigns the MIR to a variable in data memory only if CC is set true. The following example shows how this is used to maintain the variable LP, which holds the amplitude of the most recent input peak. The following sequence of instructions maintains the value of LP.

```
R CS                    ; read current sample from data memory
acc:=mor SIP load_en    ; MIR = current sample
                        ; CC is true only if at a peak
WC LP                   ; store current sample in LP only if at peak
```

In summary, the use of a FSM and a conditional write instruction replaces the role of logical instructions and conditional branches in a conventional computer architecture. The FSM attached to Processor No. 3 is quite small (only eighteen product terms), but can perform the logical operations needed for the Gold pitch tracker algorithm.

2.6 Speech Synthesizer

The speech synthesizer function of the vocoder IC is performed using a system resembling the vocoder model for speech production, Fig.2.1. The excitation source has amplitude and pitch/voicing parameters as inputs, with its output driving a synthesis filter. The filter is programmed with a set of ten reflection coefficients.

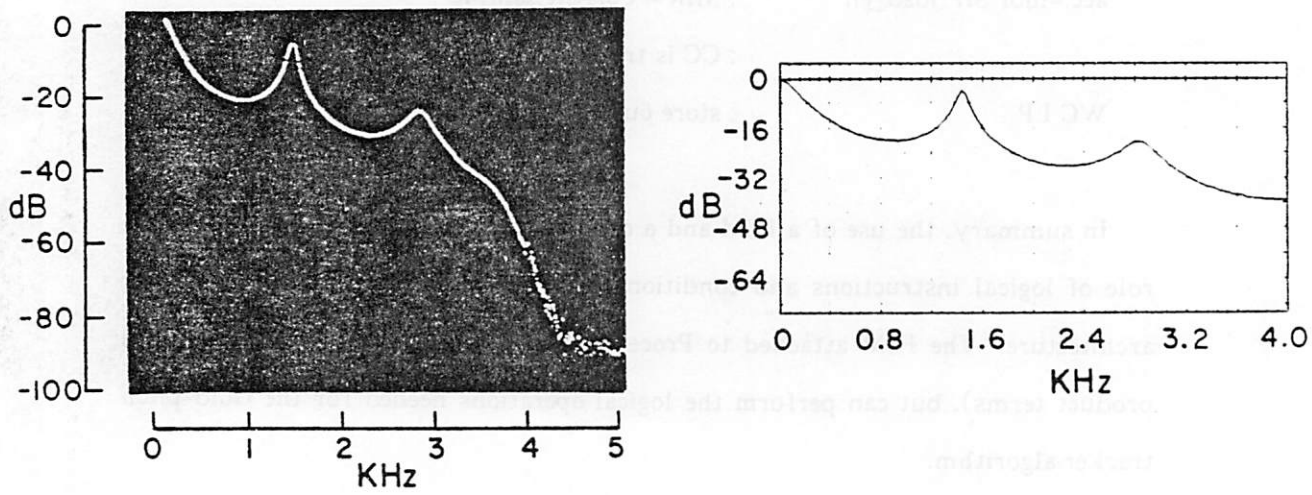


Fig.2.16 Predicted and measured response of synthesis filter

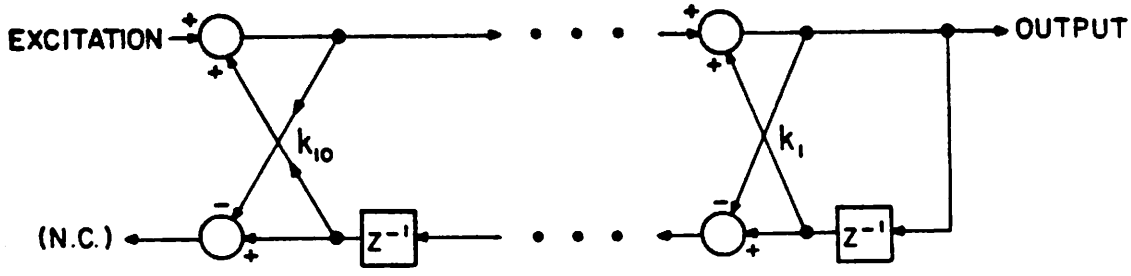


Fig.2.15 Lattice synthesis filter

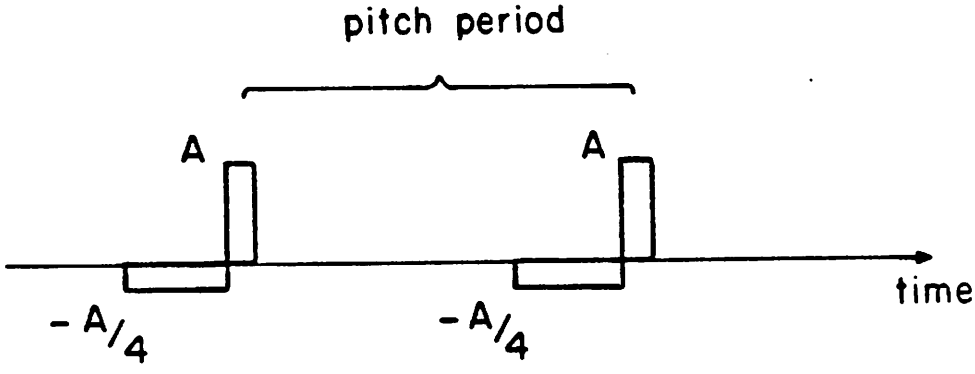


Fig.2.17 Voiced excitation waveform

Compared to the spectral and pitch/voicing analysis, the speech synthesis functions of the vocoder IC require relatively little hardware. About half of the resources of Processor No. 1 are devoted to implementing the lattice synthesis filter, Fig.2.15., and a de-emphasis network with the following response:

$$\frac{1}{1-\frac{3}{4}z^{-1}}$$

In addition, a small amount of specially-designed circuitry serves as the excitation source. Alternatively, an external excitation source may be used, allowing application of the vocoder IC in a base-band or voice excited vocoder [20].

The predicted and measured response of the lattice synthesis filter when programmed with a fixed set of reflection coefficients is shown in Fig.2.16. The reflection coefficients used were:

$k_1 = 0.391$

$k_2 = -0.586$

$k_3 = 0.703$

(all other k's are zero)

The measured response exhibits a rolloff due to anti-alias filtering in the test fixture. Taking this into account, the agreement between the predicted and measured responses is very close.

A number of different waveforms can be used for the unvoiced and voiced excitation waveforms. The voiced waveform used in the vocoder IC consists of a unit impulse train (whose period is the pitch period) convolved with the function shown in Fig.2.17. This function imparts a high-pass response to the otherwise

flat spectrum of the impulse train.

Each sample of the unvoiced excitation waveform has a magnitude of A/4, but with a sign which is the output of a pseudo/random generator. This results in a white (but not Gaussian) unvoiced excitation.

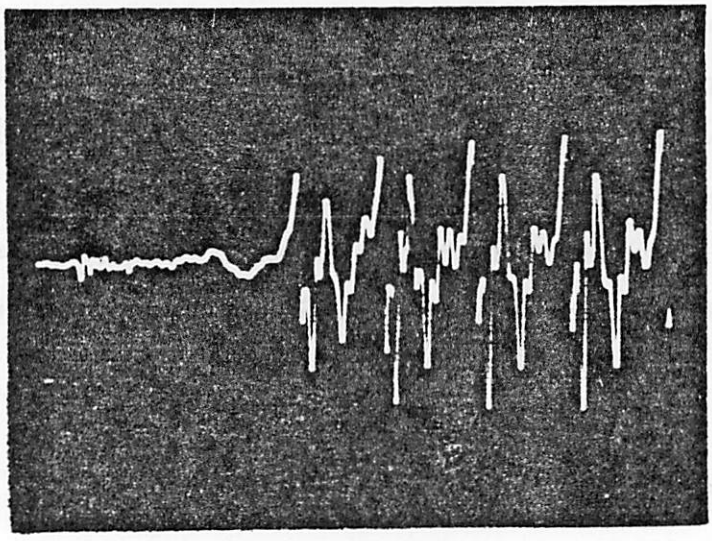
The unvoiced waveform has no D.C. component (i.e., average value of zero). The reason for using a high-pass waveform for the voiced excitation is to avoid an audible D.C. level shift on voiced/unvoiced transitions.

2.7 The Vocoder IC

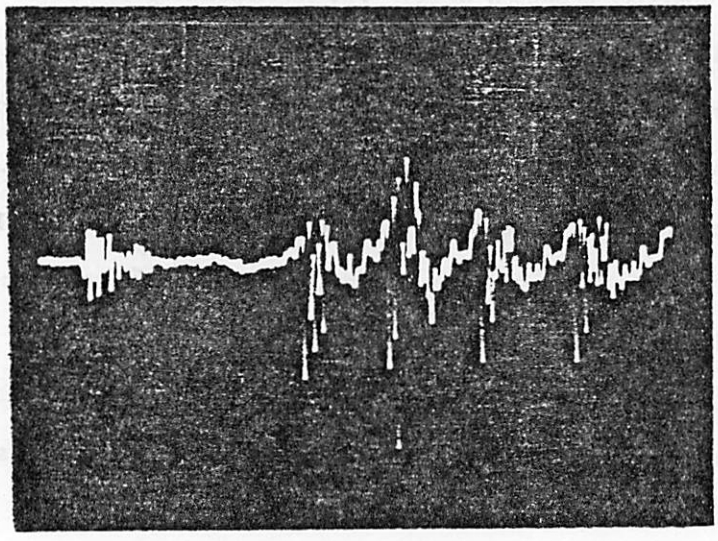
Fig.2.18 shows a complete analysis/synthesis sequence as performed by the vocoder IC. The input to the analyzer is shown in (a). This is a speech fragment of approximately 45 milliseconds in length. The residual output of the vocoder IC's lattice analyzer is shown in (b). The corresponding excitation signal (from a computer simulation) is shown in (c). The IC's lattice synthesizer, when excited by this signal and programmed with the reflection coefficients computed by the lattice analyzer, produced the synthetic speech signal shown in (d). Total length of the fragment is 44 milliseconds, divided into four frames of 11 milliseconds each. One set of pitch, voicing, energy, and reflection coefficient values was transmitted each frame.

Fig.2.19 shows a die photograph of the IC, with major functional blocks labeled. The three processors, the control sequencer, and parameter storage buffers consume most of the area.

The individual processor data paths consist of a bit-slice arithmetic unit attached to a data memory. The pitch of the data memory is narrower than that of the arithmetic unit; thus in Fig.2.19 the processor data paths are L-shaped. Groups of wires connecting the data memories to their arithmetic units are visible

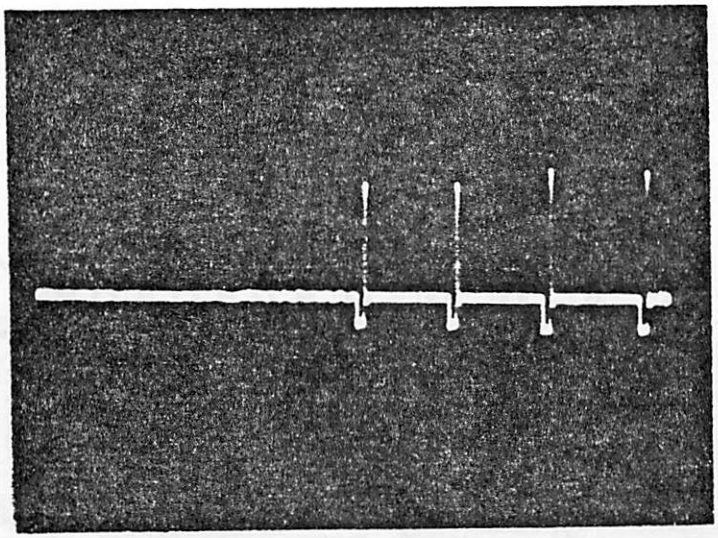


(a) speech input

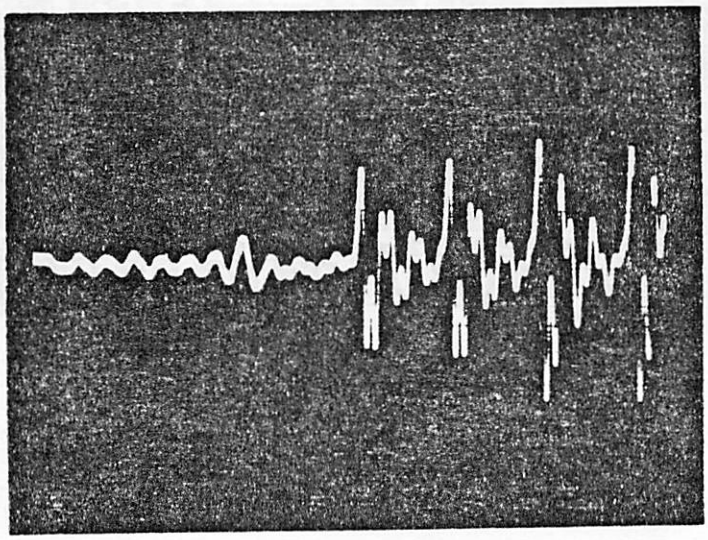


(b) residual

Fig.2.18 Analysis-synthesis sequence



(c) excitation



(d) synthetic speech output

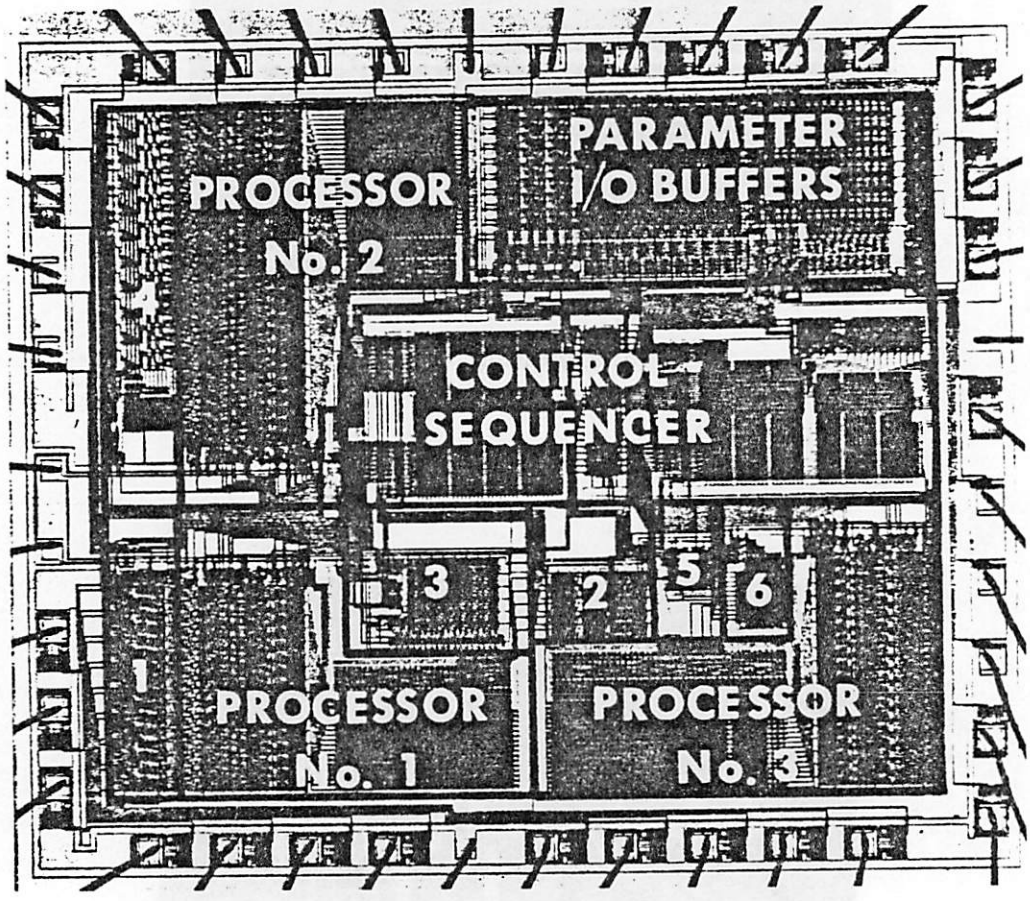


Fig.2.19 Die photograph of vocoder IC

in the photograph. Besides these, most wires in the vocoder circuit connect control sequencer outputs to the various processing elements.

The control sequencer contains two read-only memories (ROM's) and two program counters. Timing and control signals for all circuits are generated here. The two ROM's contain a total of 6 kbit of microcode.

In addition to the major blocks which are labeled in Fig.2.19. the following smaller circuits are identified by number:

(1) excitation source for synthesizer

(2) address indexing unit for Processor No. 1

(3) FIFO buffer for reflection coefficients

(4) look-up table for squaring operator

(5) address indexing unit for Processor No. 3

(6) FSM for Processor No. 3

The circuit is fabricated in a 4u NMOS process, containing 23,000 transistors on a 0.265" x 0.225" die. The circuit requires a two-phase, 2.88 MHz clock and dissipates 600 mW.

2.8 Conclusions

The design of any integrated circuit involves decisions on a number of issues. The LPC vocoder IC presented an interesting design problem due to the high

computation rate, the real-time constraint, and the fact that the algorithm as well as the architecture could be adapted to suit the technology. Held in balance by this design were performance, area efficiency and magnitude of design effort.

Quite a bit of efficiency was gained by recognizing that adaptive algorithms for spectral and pitch analysis were well suited to a fully integrated approach. These algorithms avoid the large amount of storage required by "block" algorithms, which typically require buffering a large number of input samples. Also, the fact that most of the algorithms used consisted of repetitions of identical stages (i.e., lattice filter stages or pitch detector channels) reduces the complexity of the control sequencer.

The most important aspect of a design such as this is the data path architecture. The single-accumulator architecture used in the three arithmetic units resulted in a compact and effective circuit. By operating three of these arithmetic units in parallel high computational throughput is obtained. It should be noted that a parallel array multiplier circuit would occupy more circuit area than all three arithmetic units combined. This suggests that array multipliers, which are often found in signal processing IC's, are not always the best architectural choice.

Chapter 3 – The Macrocell Approach to Digital Signal Processor Design

3.1 Introduction

Custom circuit design and production has never accounted for more than a small fraction of the IC industry. The vast majority of the digital integrated circuits designed and manufactured are high-volume parts such as memories, microprocessors, and standard logic families. Industrial production facilities are geared towards the manufacture of these high-volume components. Market conditions tend to favor the production of high volume components, with little incentive for the design and manufacture of more specialized circuits.

Still there are reasons to believe that dedicated signal processing I.C.'s will eventually play a more important role. One is the huge potential market represented by the telecommunications industry. Another is the slow but steady progress being made in the field of computer-aided design (CAD) of integrated circuits.

Several competing approaches exist for custom and semi-custom integrated circuit design. Gate arrays and standard-cell designs allow a short design cycle, but a premium is paid in terms of efficiency. The approach taken here is the use of large, parameterized blocks of circuitry called *macrocells*. These macrocells differ from standard-cells in that they are large (ranging from several hundred to several thousand transistors). Also, standard-cell designs consist of regular rows of cells separated by wiring channels. Macrocells are assembled from a cell library by forming a two-dimensional array of cells. This leads to macrocells which are themselves very dense. A major consideration in macrocell-based designs is the proper placement and interconnection of these large blocks.

Of primary importance is the ability to configure the macrocells for a

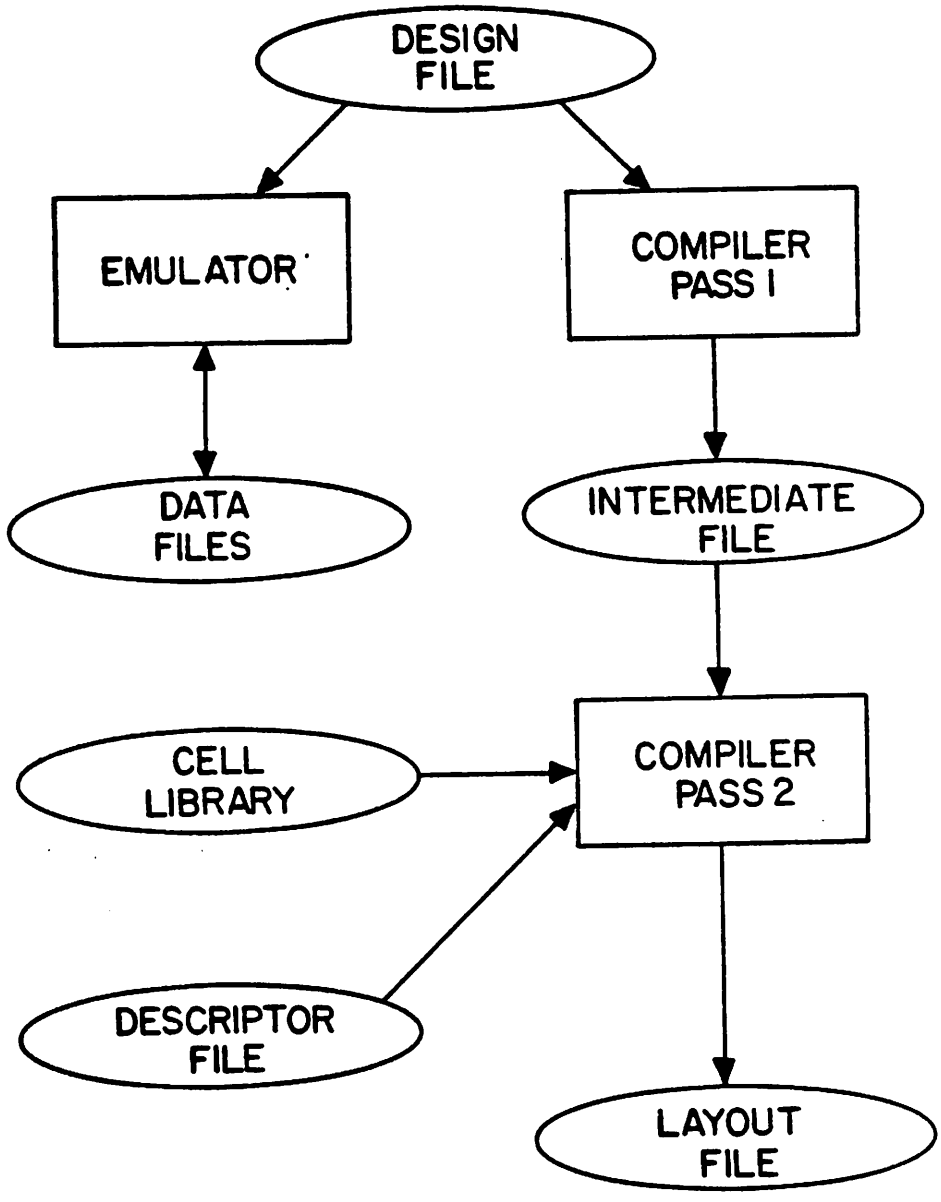


Fig.3.1 Macrocell design system software

particular application, resulting in a more efficient design. A number of parameters and options may be specified for the macrocells described here. These include data path widths; programming and sizing of control memory, data memory and programmed logic arrays (PLA's); and configuration of the interprocessor communication network. Thus a macrocell is not a fixed block of circuitry, but a class of circuit blocks configured according to explicit rules from the underlying cell library.

Design systems based on standard cells or gate arrays involve considerable supporting software for simulation and layout generation. A similar approach is taken here. The software system consists of an *emulator* and a two-pass *silicon compiler*. The emulator and compiler accept as input a *design file*, which is a text file prepared by the designer. This software system is diagrammed in Fig.3.1.

The design file has a very readable format. Each IC contains several processors operating concurrently. The designer specifies hardware parameters and symbolic microinstructions for each processor. Local constants and variables are declared. Interprocessor and off-chip communications are specified by declaring global variables which are external to the individual processors.

The emulator allows verification of circuit performance prior to fabrication. A full set of debugging commands is available, such as tracing, breakpoints, single-step, and setting or displaying variables. A non-interactive mode allows long simulation runs to be performed.

The first pass of the compiler extracts hardware parameters from the design file. Symbolic microcode is assembled into binary. The resulting hardware description is input to the second pass, which accesses the cell library and assembles the macrocells. Placement and interconnect routines follow. The system is intended to generate automatically a mask-level description of the I.C.

The term *silicon compiler* has been generously applied to specialized IC design

systems which generate circuit layouts automatically [21]. In fact, any layout-generating software which does not fit into some other category (graphics editor, router, etc.) is likely to be called a silicon compiler. The system described here satisfies this definition.

A major goal in the design of the software package was process independence. Integrated circuit fabrication processes evolve rapidly. Also, it is dificult for a designer who does not control a captive fabrication facility to ensure continued availability of a given process. For these reasons, a macrocell design system should function for any fabrication process that might become available. This goal was achieved by introducing a small number of parameters which characterize the process for the purposes of the software package. The software may then be used with any process by redesigning the cell library in the new process.

By limiting the focus of the design system to a family of signal processing functions, efficient and specialized architectures may be employed. The range of applications includes: speech processing (vocoders, sub-band and waveform coders, speech recognition), telecommunications (modems, line equalizers, echo cancellers), and digital audio (mixing, equalization, reverberation and other effects). It is estimated that several dozen commercially valuable dedicated IC's will be designed with this system.

The remainder of this chapter discusses the architecture and hardware organization used in the macrocell system. Chapter 4 describes the design file and the software system. In Chapter 5, example designs using the macrocell system are presented. Details of the cell library and design file format are presented in Appendices B and C respectively. Design file examples are given in Appendix D.

3.2 Processor Architecture

The LPC Vocoder circuit described in chapter 2 is essentially the prototype architecture for the macrocell based designs considered here. This architecture is characterized by the following:

(1) Multiple processors on a single IC.

(2) A bit-parallel, single accumulator processor architecture.

(3) Microprogramming of dedicated signal processing functions.

(4) Use of a finite state machine for decision-making.

(5) Bit-serial communications among the processors.

Study of algorithms within the target range indicated that the same general architecture, with suitable parameterization, can perform the algorithms efficiently.

For convenience, discussion of the architecture is divided into several sections. This section is an overview of the individual processor architecture. Sections 3.3 through 3.6 describe components of the processor in greater detail. Section 3.7 describes the multiprocessor and I/O aspects.

Each processor is organized into the following macrocells:

PC (program counter)

ROM (microcode read-only memory)

SPC (subprogram counter -- optional)

AAU (address arithmetic unit -- optional)

FSM (finite state machine -- optional)

AUIO (arithmetic unit with I/O circuits)

RAM (processor data memory)

A fully configured processor is diagrammed in Fig.3.2.

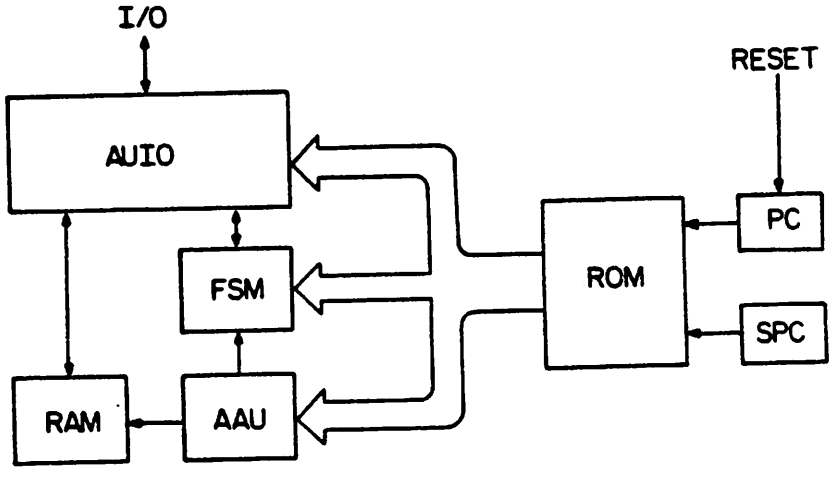


Fig.3.2 Macrocells forming a fully configured processor

The processor executes its microprogram (stored in ROM) once per sample interval. The microprogram consist of a "main program", optionally followed by a fixed number of "subprogram" iterations. The SPC macrocell is included only if the subprogram is used. Except for this iteration pattern, there are no branches (conditional or unconditional) in the program execution. The PC, ROM and optional SPC macrocells are collectively known as the control sequencer and are described in Section 3.3.

The output of the control sequencer is a sequence of control words. The control word can be subdivided into a number of fields. One of these is the address offset field. It is this field that is used as input to the AAU macrocell. The AAU modifies the address offset to produce the effective address for the processor data memory (RAM macrocell). In some simple cases, no modification is required and the AAU macrocell is not included. In these cases the address offset field of the control word serves directly as the effective address. The AAU macrocell is described in Section 3.4.

Viewed together, the ROM, PC, SPC and AAU present a stream of horizontal control and address words to the AUIO, RAM, and (optional) FSM macrocells.

The AUIO macrocell consists of an arithmetic unit and an I/O interface. The arithmetic unit together with the RAM (data memory) macrocell form the signal data path. This is where all operations on signals are performed. This data path may be microcoded to perform fixed coefficient multiplies, variable coefficient multiplies, add, subtract and accumulate opreations, division, and comparisons. In addition, the data memory allows delay operations, introduction of constants and table-lookup. The arithmetic unit and data memory are described in Section 3.5.

The final component of the processor assembly is the optional finite-state machine (FSM). This is where logical operations, conditional operations and decision-making are performed. The FSM accepts Boolean inputs from two

sources: a comparison in the arithmetic unit; and tests on index registers internal to the AAU. The FSM operates on these inputs and its own internal state. FSM opeations are controlled by a field of the control word. Outputs from the FSM are used for two purposes: to control write cycles in the RAM macrocell; and to initiate I/O to an off-chip host (see Section 3.7). The FSM is described in Section 3.6.

The processor assembly executes its microprogram once per sample interval. Since primitive operations such as multiplies must be microcoded, it follows that the sample rate must be substantially slower than the processor's clock rate. Otherwise, there would not be time for a significant amount of processing during the sample interval. On the other hand, if the ratio of clock rate to sample rate becomes very large, the architecture exhibits an imbalance wherein the control ROM consumes nearly all of the silicon area, and the data path only a small fraction. This situation is aggravated since the horizontal control words are not densely encoded. Based on the relatively sizes of the library cells, the processor architecture is most efficient if the ratio of clock rate to sample rate is between 50 and 1000. This corresponds to a sample rate range of 5 kHz to 100 kHz, since the cell library is designed to operate at a maximum 5 MHz clock rate.

3.3 Control Sequencer

An important aspect of the macrocell design system is the fact that the individual processors are microprogrammed to perform dedicated tasks. In fact, most of the design file input to the software system consists of microcode. The control sequencers are programmed with this microcode, and therefore control all timing and data-path operations for their processors. In addition, timing strobes generated by the control sequencers can exit the processor assembly to be used for control of interprocessor or off-chip communications.

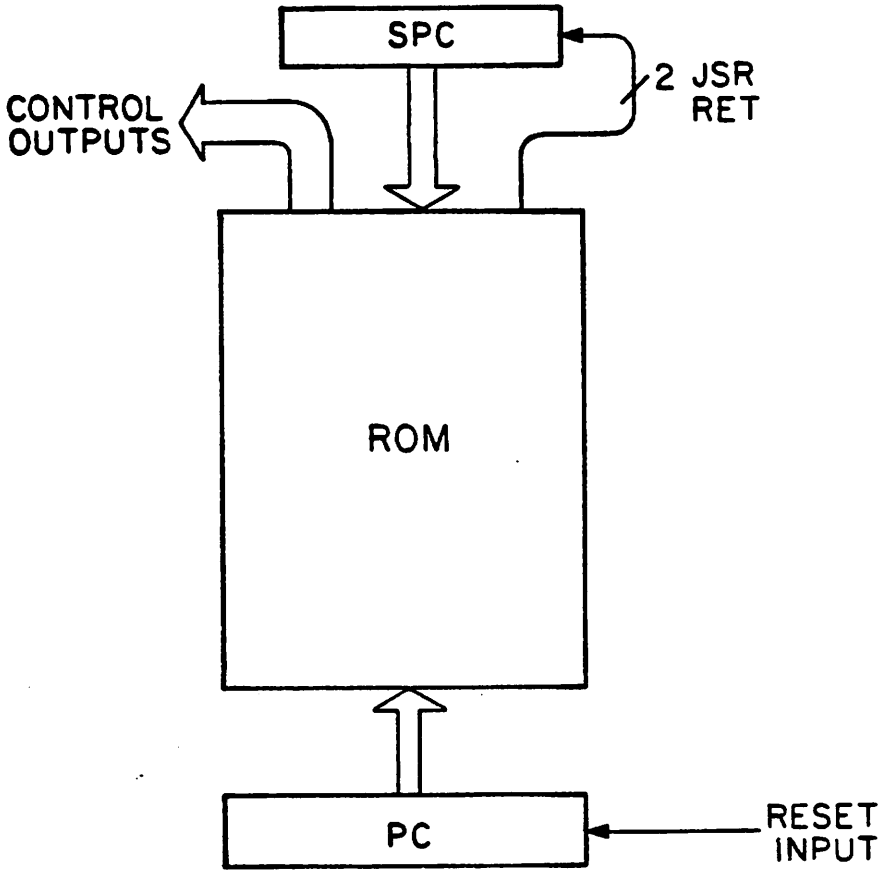


Fig.3.3 Control sequencer

The control sequencer consists of the ROM macrocell, the PC (program counter) macrocell, and the SPC (subprogram counter) macrocell. The subprogram is optional; if there is no subprogram, the SPC is not included. The outputs of the ROM are known as the control word. The control word is very horizontal: it contains many non-overlapping fields, each of which has an independent function. Rarely are two different functions assigned to the same bit of the control word.

The AUIO macrocell has many control inputs, accounting for perhaps half of the control word. Therefore, provisions are made for defaulting unused signals in this field to ground or Vdd, reducing the typical ROM width.

A few control signals are used to clear and increment the two program counters, PC and SPC. It is required that the various processors on a chip be synchronized, i.e., each processor begins its microprogram on the same clock cycle at the beginning of each sample interval. In order to allow this, one processor is designated as the master processor for timing purposes. Contained in the control word for the master processor is a signal named RESET. This signal is asserted at the end of each sample interval, and serves to reset the PC (program counter) macrocells for all processors on the chip. There are no inputs to the control sequencer aside from this reset input.

There are two possible configurations for the control sequencer: with or without a SPC. The following discussion applies to the case where there is a SPC. The hardware is diagrammed in Fig.3.3.

The SPC macrocell requires two timing signals JSR and RET as control inputs. The JSR signal is asserted at the end of the main program, and at the end of each subprogram iteration. The RET signal is asserted concurrently with the final JSR at the end of the final subprogram iteration. When JSR and RET occur simultaneously, the SPC is cleared and remains cleared until the next JSR. When JSR is

asserted but RET is inactive. SPC is loaded with a one and commences counting. Thus, SPC equals zero during the main program and counts from one to an upper limit during each subprogram iteration.

Note that the end of the final iteration of the subprogram need not be coincident with the end of the sample interval. The reason for this is that the total execution times for different processor's microprograms will in general be unequal. Therefore, there may be an idle interval following the end of the microprogram lasting until the end of the sample.

Also note that the subroutine iterations are not explicitly counted. The subroutine iterates the desired number of times only because the RET signal is asserted on the proper cycle.

RESET, JSR, and RET are asserted two cycles before their respective actions are effected. This delay is due to the pipeline register at the output of the control ROM. As an example, suppose that there are 200 clock cycles in a sample interval. The program counter counts from zero to 199. Suppose that there are 60 instructions in the main program, and 40 instructions in each of three subprogram iterations. The following table describes the sequence of events for a complete sample interval:

	PC value	SPC value	Timing signals
main program	0-59	0	
	58	0	JSR
1st subprogram	60-99	1-40	
	98	39	JSR
2nd subprogram	100-139	1-40	

	138	39	JSR
3rd subprogram	140-179	1-40	
	178	39	JSR, RET
idle interval	180-199	0	
	198	0	RESET

It is clear from the above table that the ROM must be programmed so that JSR is asserted whenever PC = 58 or SPC = 39, and that RET must be asserted whenever PC = 178. These signals are readily generated if the split control ROM structure of Fig.3.3 is employed. In this arrangement, the and-plane (decoder) of the ROM is split, with the upper half addressed by the SPC and the lower half addressed by the PC. The or-plane (core) of the ROM is a single section with bitlines from both halves connected together. This results in a given ROM output being asserted if an instruction in either half of the ROM is programmed to assert the output. Thus, in our example, location 39 in the upper-half ROM and location 178 in the lower-half ROM both assert the JSR signal.

The lower half-ROM in the example only need contain locations 0-59, 178 and 198 (62 words). The upper half-ROM only contains locations 1-40 (40 words). Thus the entire ROM contains 102 words. However, additional locations may be included in the lower-half ROM if they are needed to generate external timing strobes that happen to fall outside the main program, but still do not recur periodically with the subprogram. The dual program counter, split-ROM arrangement makes it possible to generate such strobes, which would not be the case with a more traditional single program counter approach.

Both PC and SPC provide complementary, buffered outputs which drive the and-planes directly. The PC contains an additional output EOS (end-of-sample) which is used for timing purposes in the AAU and in the host interface. The EOS

signal is always asserted during the last clock cycle of the sample interval (the cycle following RESET).

3.4 Address Arithmetic Unit

In many traditional computer architectures, address arithmetic and data operations are performed in the same arithmetic element. This can result in congestion and inefficient data flow. In signal processing applications, the types of address arithmetic encountered are less varied and may be categorized and implemented by a special functional unit. This functional unit can then operate in parallel with the arithmetic unit that processes the signal data, resulting in inreased throughput. The AAU (address arithmetic unit) macrocell serves this purpose.

The AAU is included only if indexed adressing is used. The alternative is direct addressing, where the address offset field of the control ROM is used as the effective address for the RAM.

The AAU contains one or both of the counters IX and IY. Indexing by either or both of these counters is possible. When indexed addressing is used, the index counter is added to the address offset to give the effective address.

The IX counter counts the subroutine iterations. IX equals -1 during the main program, 0 during the first subroutine iteration, 1 during the second iteration, and so forth.

IX indexing is used if identical operations must be performed on a fixed number of different data sets. The operation is coded into the subprogram of a processor. The subprogram references IX-indexed arrays in the RAM data memory. Indexing by the IX counter allows the subprogram to operate on elements of an array in data memory, accessing a different array element each time the subprogram iterates. This is the same sort of indexing used in the LPC vocoder IC (chapter 2)

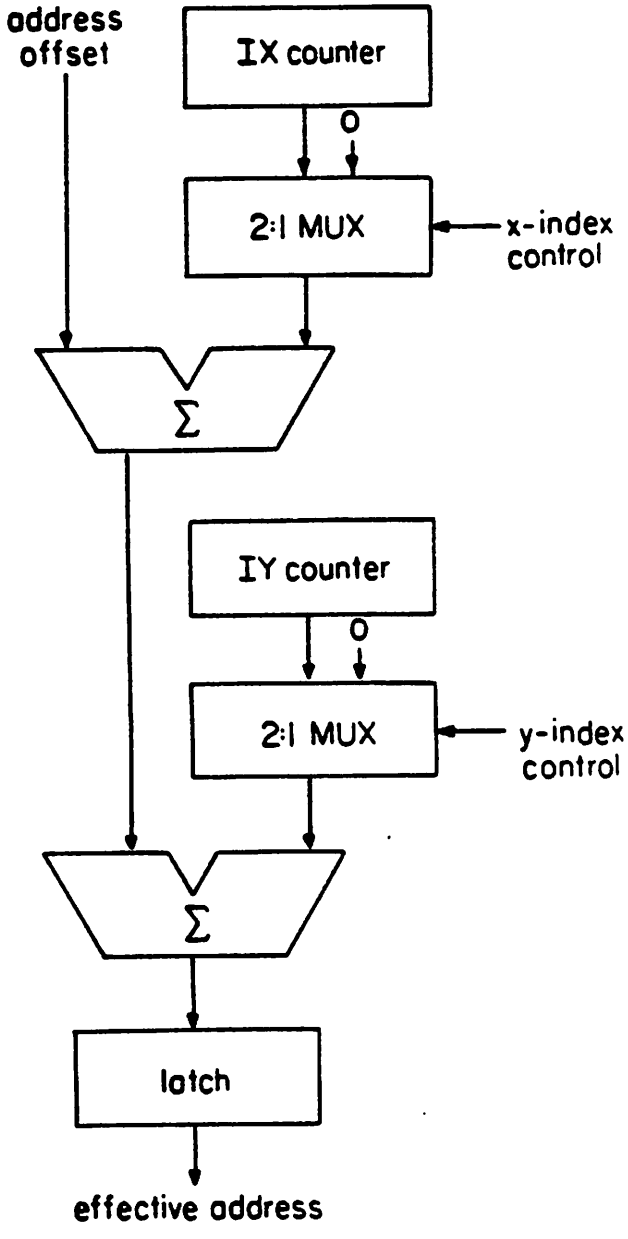


Fig.3.4 Counter mode AAU

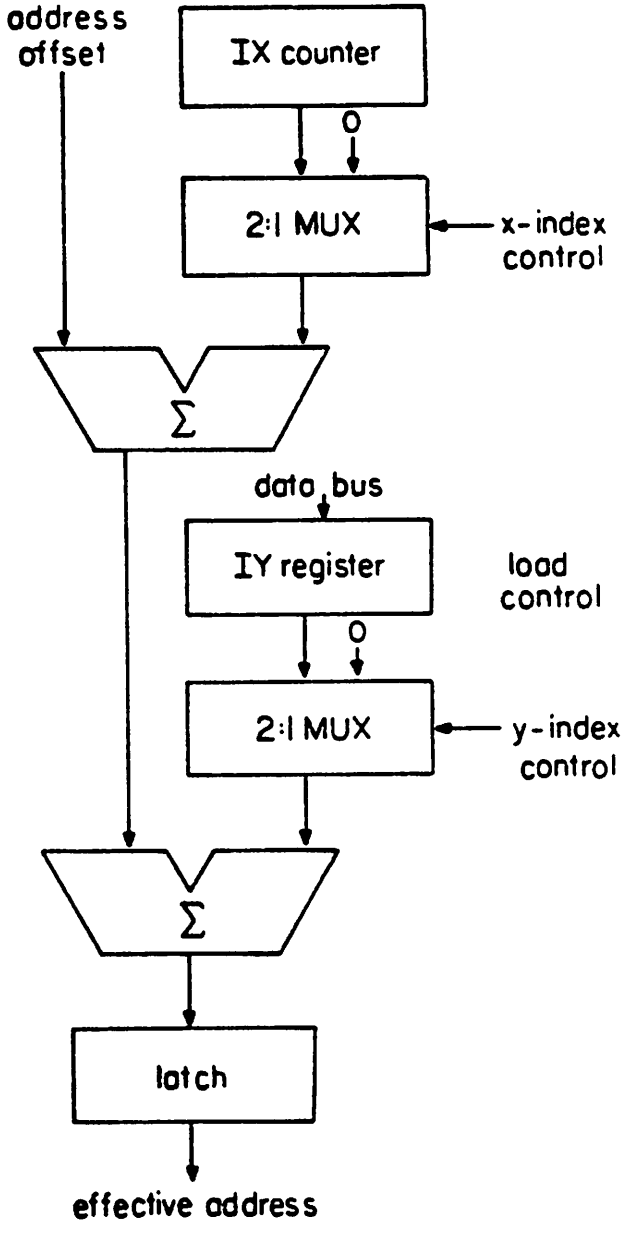


Fig.3.5 Pointer mode AAU

to multiplex the ten lattice filter sections into a single subprogram.

The IY counter has two modes: pointer mode and counter mode. In pointer mode, the counter serves as a register which may be loaded from the data bus of the arithmetic unit. This allows arbitrary address arithmetic to be performed in a moderately efficient fashion. In counter mode, IY is a sample counter with a fixed modulus.

Counter mode indexing by IY allows the main program to operate on elements of an array in data memory, accessing different elements from sample to sample. This permits multiplexed processing of a group of decimated signals. The implementation of the scoring algorithm for the Gold pitch tracker, described in Section 2.5, is an example of this sort of multiplexing.

Pointer mode addressing, on the other hand, is useful for table look-up operations. A good example is a controlled oscillator. A look-up table containing one quadrant of a sinusoid is stored in RAM. The address sequence for this lookup table is computed in the arithmetic unit, to give the desired phase and frequency. The computed address is then transferred to the AAU, where it is used to index into the look-up table.

The disadvantage of pointer-mode addressing is that it no longer preserves the separation of address and signal data paths. It therefore is less efficient than the IX and counter-mode IY indexing modes described above. Fortunately, pointer-mode addressing is needed only in a small number of applications.

Fig.3.4 shows a block diagram for a fully configured counter mode AAU, while Fig.3.5 shows a fully configured pointer mode AAU. Both types of IY indexing are not available in a single AAU.

One final feature of the AAU, not illustrated in Figs. 3.4 and 3.5, is a provision for providing outputs which test the IX and IY counters. Any number of test outputs may be provided. The test outputs may test either individual bits of a

counter, or test for equality between a counter and a constant. These test outputs are used as input to the FSM, and provide a useful hook into the address information. For example, it may be necessary to perform a given operation only during a particular subprogram iteration. By testing IX it is possible to determine whether the current iteration is the one in question, and condition an operation on this information.

3.5 Processor Data Path

3.5.1 Data Path Organization

The data path architecture is of paramount importance in any computational system. As a general rule, the most efficient architectures are highly specific to particular applications. However, the set of target applications for the macrocell system can reasonably be implemented using the same data path architecture. The word length (data path width) may be specified to suit the requirements of the particular application. A bit-slice organization for the arithmetic unit allows this parameterization. The size of the data memory, and the I/O circuits that interface the data path to other processors, may be configured as well.

The general philosophy of the arithmetic unit design was that of simplicity. The number of busses is kept to a minimum, and registers are included only when needed to allow pipelining. This approach results in a compact computational unit whose aritmetic elements can be heavily utilized.

The AUIO macrocell contains the arithmetic unit and the processor I/O section. The processor I/O section is described in Section 3.7. Together with the RAM (data memory) macrocell, the arithmetic unit forms the main processor data path where all signal processing is performed. Fig.3.6 shows the AUIO and RAM

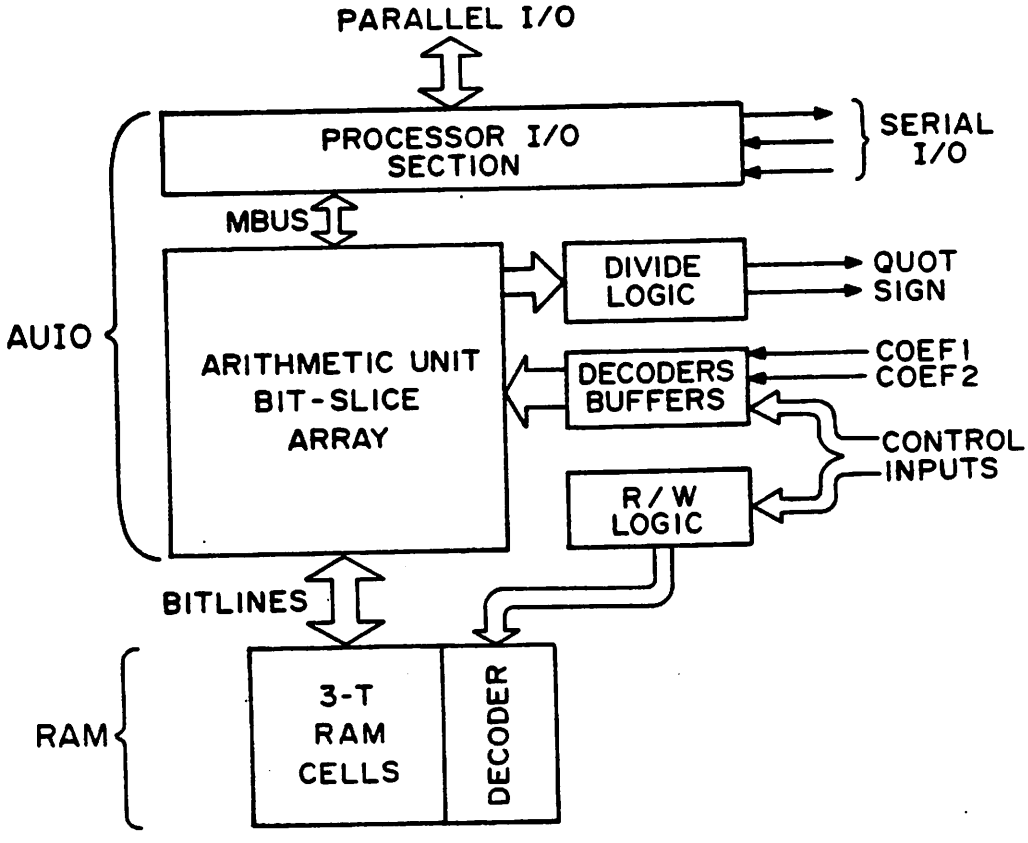


Fig.3.6 AUIO and RAM macrocells

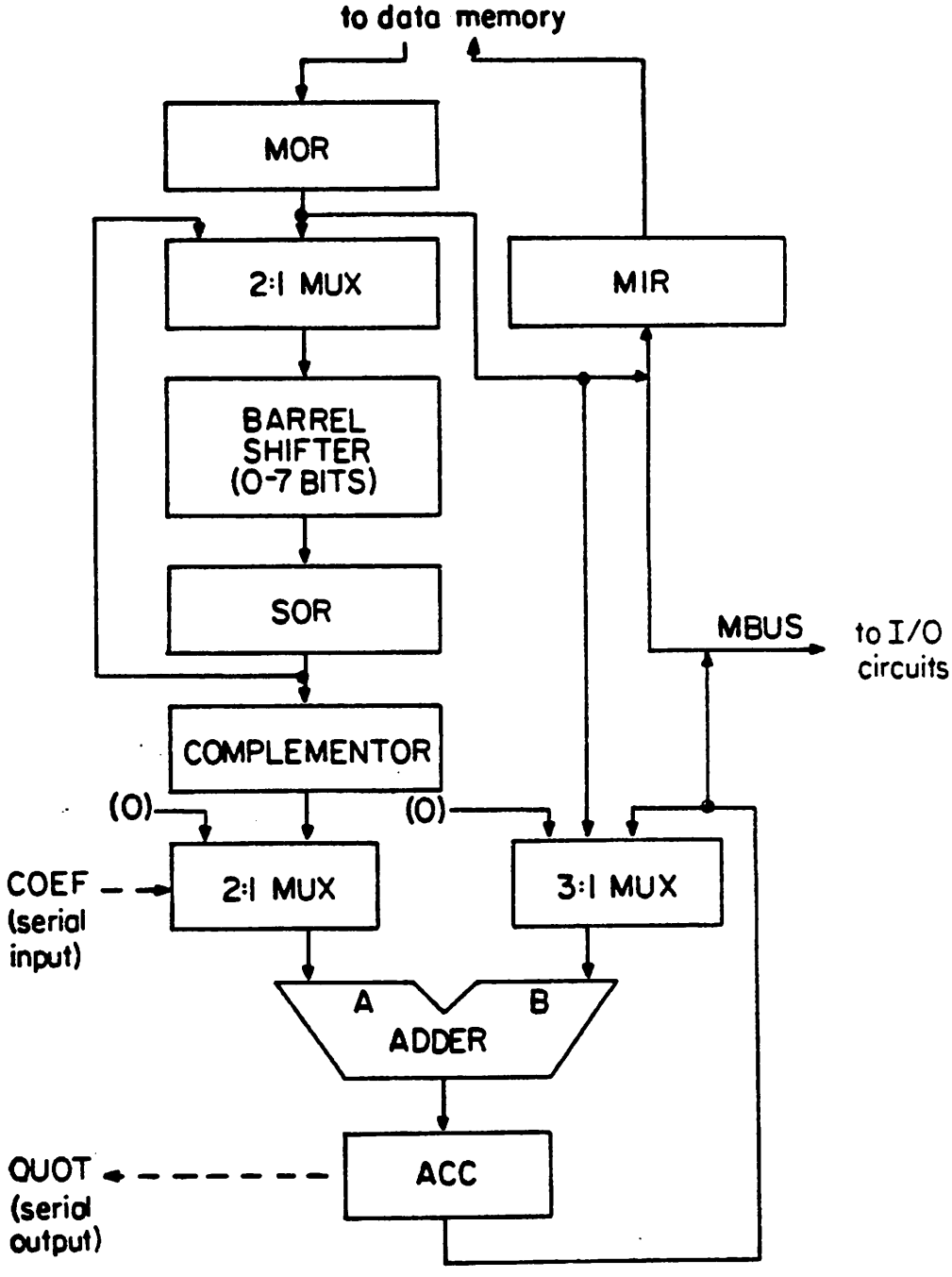


Fig.3.7 Processor arithmetic unit

macrocells in block diagram form.

A large number of control signals enters the control section of the arithmetic unit. These signals are decoded, buffered, and transmitted across the bit-slice array.

In addition to control signals, two serial data inputs (COEF1 and COEF2), one serial data output (QUOT), and the SIGN output connect to the arithmetic unit control section.

3.5.2 Arithmetic Unit

The processor arithmetic unit (Fig.3.7) consists of a barrel shifter of depth eight; a complementer; a saturating adder; three master-slave registers (MOR, SOR and ACC); and a transparent latch (MIR). (The value loaded into a master-slave register on one cycle appears at that register's outputs the following cycle, whereas the effect of loading the transparent latch is immediate.) This processor may be microprogrammed for all common signal processing functions. There is a single bus (the "MBUS") through which I/O operations are performed.

All data is in two's complement format.

Every clock cycle, a new set of control signals is transmitted from the control sequencer to the control section of the arithmetic unit. The following describes the effect of these control signals.

The MOR (memory output register) is loaded each cycle with the value on the data memory bitlines. This will be either the result of a read operation, or the inverted write data.

A multiplexer at the input to the barrel shifter selects either the output of the MOR or the output of the SOR.

The barrel shifter itself performs an arithmetic right shift of anywhere from

zero to seven bits. Three control signals are decoded to control the amount of shift.

The SOR register is always loaded with the output of the barrel shifter.

A number of controls are provided for the A and B inputs of the saturating adder. (The merits of saturating arithmetic were discussed in Section 2.2.) The A inputs may be set to zero, or to the true, complement, or absolute value of the SOR register. In addition, the A inputs may be gated (i.e., bitwise and'ed) with COEF1, COEF2 or the inverses of COEF1 or COEF2. This gating allows the microcoding of two's complement multiplies, described in the following section. The B inputs may be set to zero, MOR, or ACC.

The ACC (accumulator) has a single control option, "accumulate if positive", used mainly for microcoding of divisions. When this option is enabled, ACC is loaded with the adder output only if the adder output is non-negative. When the accumulate if positive option is not enabled, ACC is always loaded with the adder outputs.

A single bus (the MBUS) is provided. The MBUS is used for I/O. If an input operation is in progress, the processor I/O section will enable the input data onto the MBUS. At all other times, the ACC will be enabled onto the MBUS.

The transparent latch MIR always takes its inputs from the MBUS. A control is provided to either load or hold the value in the latch.

The COEF1 and COEF2 inputs are used for variable coefficient multiplies, described in Section 3.5.3. The QUOT output is used for division, also described in Section 3.5.3. The SIGN output is used as input to the FSM macrocell as described in Section 3.6. The two inputs COEF1 and COEF2 are identical in function: providing two suc inputs allows transmission of coefficients from two distinct sources (e.g., two procesors) to the same destination.

The pipeline organization of the arithmetic unit allows concurrent memory

access, barrel shift and add/accumulate operations. In addition, pipelining around the MBUS prevents bus settling time from impacting throughput. A fast ripple carry adder (identical to that described in Section 2.3) is used. This allows a flexible bit-slice organization, but gives add times comparable to more complex look-ahead designs.

Other organizations for single-accumulator arithmetic units are possible. One popular arrangement is to put the barrel shifter *after* the accumulator rather than before. In this case the least significant partial products are accumulated first during a mutiply sequence, and the binary weight of the value in the accumulator increases during the course of the multiply. This approach has the advantage that the truncation errors are not as severe, gaining a few bits of precision for the same wordlength. There are however two disadvantages: (1) an additional register is needed to store intermediate results in a multiply-accumulate sequence; (2) an external coefficient must be provided LSB-first. This is incompatible with the MSB-first format which is needed to implement long-division operations (described in the following section).

3.5.3 Multiply and Divide Operations

Essential to any signal processor architecture is the ability to perform multiply-accumulate operations. Two cases are handled separately: multiplying signal data by a variable coefficient external to the processor; and multiplying signal data by a constant.

In the case of variable coefficients, the coefficient is available externally in a bit-serial, MSB first, fractional two's complement format. Each processor arithmetic unit allows two such serial coefficient inputs. In fractional two's complement form, the coefficient k can be expressed as follows:

$$k = -k_{n-1} + \frac{k_{n-2}}{2} + \frac{k_{n-3}}{4} + \ldots$$

To multiply a signal A by the coefficient k, the following equation is used:

$$kA = -k_{n-1}A + k_{n-2}\frac{A}{2} + k_{n-3}\frac{A}{4} + \ldots$$

Each term on the right-hand side is a partial product, which is non-zero depending on one of the bits of the coefficient k. To perform these variable-coefficient multiplies, the external coefficient must be multiplexed into the control path for the arithmetic unit. This allows the partial products to be summed sequentially, one per clock cycle.

This method of multiplication is particularly convenient in conjunction with the bit-serial approach to interprocessor communication. Two coefficient inputs (COEF1 and COEF2) are provided for each processor arithmetic unit, along with the control circuitry to enable these coefficients under microprogram control. A coefficient may originate from the same processor that performs the multiply, or it may originate from a different processor and be transmitted over a bit-serial path.

As an example, suppose that one wishes to multiply a variable A in data memory by the 8-bit coefficient k. The following sequence is performed, each line representing one clock cycle:

Read A from RAM into the MOR

Load the SOR from the MOR

If k_7=1 (negative) load -SOR into ACC; shift SOR right one bit

If k_6=1 add SOR to ACC; shift SOR right one bit

If k_5=1 add SOR to ACC; shift SOR right one bit

If k_4=1 add SOR to ACC; shift SOR right one bit

If k_3=1 add SOR to ACC; shift SOR right one bit

If k_2=1 add SOR to ACC; shift SOR right one bit

If k_1=1 add SOR to ACC; shift SOR right one bit

If k_0=1 add SOR to ACC

If one is performing a sequence of multiplies or multiply/accumulate operations, the 8-bit multiplies can be performed at the rate of one every eight clock cycles.

In the case of a fixed coefficient, it is not necessary to input the coefficient into the processor through the COEF inputs. Instead, a signed-digit representation of the coefficient is embedded in the control stream. This is done by specifying a sequence of shift depths and sign values. Thus, any coefficient g has a signed-digit representation:

$$g = (-1)^{s_0}2^{c_0} + (-1)^{s_1}2^{c_1} + \ldots$$

Any binary number has a unique such representation with the minimum number of digits. This is known as the canonical signed-digit (CSD) representation [22]. The importance of the CSD representation is that is minimizes the number of clock cycles required to perform a multiply by a fixed coefficient. The multiply is performed by sequencing the barrel shifter and complementer so as to present the desired terms to the accumulator.

Suppose, for example, that a variable A in data memory needs to be multiplied by the constant g = .10111100, which may be recoded as

$$2^{-1} + 2^{-2} - 2^{-6}$$

The following sequence performs the multiply:

read A from memory into MOR
load SOR with MOR shifted right one bit
load ACC with SOR; shift SOR right one bit
add SOR to ACC; shift SOR right four bits
add (-SOR) to ACC

The product gA is now in the accumulator.

In both the above cases, a sequence of multiplies may be performed, with the result of each being accumulated. This is an efficient feature of the single-accumulator architecture. The same accumulator is used for adding partial products, and for accumulating the results of several multiplies.

Although not as prevalent as mutiplies in signal processing algorithms, divide operations are also needed in some cases. Usually, a divide is used for normalization. The lattice LPC algorithm described in Section 2.4 requires a division to compute a normalized cross-correlation value, for example.

A common situation requires performing a two-quadrant divide N/D, where $D > |N|$. This will give a quotient between minus one and one. It is desirable to create a two's complement signed quotient. In order to achieve this, first the absolute value of the numerator N is loaded into the accumulator, and the sign bit is saved. The sign bit is immediately outputted as the sign bit of the bit-serial quotient. Then, an unsigned (one quadrant) long division of $|N|/D$ is performed. The resulting quotient bits are exclusive-or'ed with the saved sign bit, giving a two's complement result.

In order to do the long division, the accumulate-if-positive control option is used. This loads the accumulator only if the result of the accumulator operation is not-negative. The inverted sign bit of the result of the accumulator operation is used as the quotient bit. With $|N|$ in ACC, $D/2$ in SOR, and the accumulate-if-positive option enabled, the following operation is repeated:

load ACC - SOR into ACC only if positive ; shift SOR right one bit

Each time this operation is performed, another bit of the quotient is computed, that bit being the sign bit (inverted) of ACC-SOR.

Hardware for performing the absolute value, accumulate-if-positive, sign-bit storage and exclusive-oring of the quotient bit is contained in the arithmetic unit. This allows two-quadrant divides to be performed, with the resulting quotient being in the same bit-serial, two's complement, MSB-first format that is required for coefficients in variable-coefficient multiplies.

3.5.4 Data Memory

The data memory (RAM) macrocell is used for storage of temporary results and state variables of the signal processing algorithm. As such, a given memory location is typically written at least once each sample interval. This generally eliminates the need to explicitly refresh memory locations; thus no hardware refresh is provided.

Read-only and read-write memory locations are intermixed in the RAM. The read-only locations allow introduction of constants into the computation. Note that in traditional architectures, constant introduction is accomplished by the use of immediate operands, requiring a gateway between control and data paths. The

fact that the RAM is a macrocell configured for a dedicated application allows a more direct and efficient method of introducing constants.

Read-only locations also allow the implementation of look-up tables, although the current memory size limit of 64 words makes larger look-up tables impractical.

3.6 Finite State Machine

In a traditional computer architecture, the only conditional operations available are conditional branch operations. Usually, condition code flags are set by testing one or more CPU registers. Then, a conditional branch operation may be executed. The branch takes place only if the specified condition is satisfied by the flags.

Another characteristic of the traditional approach is that all decision-making logic is performed in the same ALU that processes numeric data. This is convenient for general-purpose CPU's since the type of decision making is not known in advance, and the hardware cannot be tailored to the application.

A radically different approach to decision-making was taken in the macrocell-based design system. First, a conditional write operation is made available to the programmer, as opposed to the conditional branch. Second, Boolean data is processed in a finite state machine (FSM), separate from and concurrent with the processing of numeric data in the arithmetic unit.

The configuration of the FSM with the processor data path is shown in Fig.3.8.

The use of conditional assignments as opposed to conditional branches is not new. A classic result from the theory of programming languages [23] states the the conditional assignment statement is equally as general as the conditional branch, although not necessarily as efficient. Intuitively, a conditional assignment

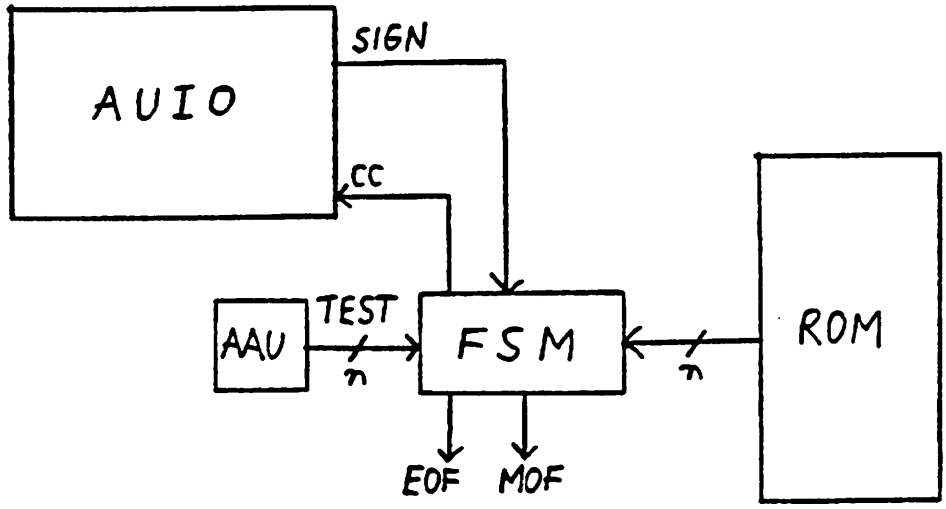


Fig.3.8 Use of finite state machine

acts as follows:

if (condition is satisfied) then (assign a value to a variable)

while a conditional branch does the following:

if (condition is satisfied) then
(branch to different stretch of code)

The conditional branch has two major disadvantages if used in signal processors such as those considered here. First, if conditional branches are used the execution time of the program is data-dependent. This is undesirable in a real-time sampled system, since the program must execute in its entirety each fixed-length sample interval. Also, the interprocessor communication (Section 3.7) is dependent upon the programs of the processors being synchronized with one another. Second, the control flow is also data-dependent. This makes it less practical and less efficient to pipeline control sequence generation with data operations (i.e., overlapped fetch/execute).

The conditional write approach has one major drawback. It is much less efficient than the conditional branch if the test selects between two long, substantially different pieces of code. Fortunately, this situation seldom arises in signal processor design.

An important feature of FSM usage in the macrocell approach is that the size and contents of the FSM is tailored to the application. Thus, the state variables of the FSM can be selected to correspond to conditions present in the signal processing algorithm. A field of the horizontal control word is used to modify the state of the FSM; this modification is also made algorithm-specific.

In Section 2.5.2, the design of a customized FSM for the LPC Vocoder IC was described. This FSM was sucessfully used in implementing the Gold pitch tracker algorithm.

The design file input (described in Chapter 4) allows the design of the FSM to be easily specified. This is a very powerful capability, since "FSM instructions" may be specified and then later used in the microprogram for the processor. In effect, the user is able to design part of his instruction set prior to programming the processor.

The FSM is implemented as a PLA (programmed logic array) with an output register for the state, and feedback from the output register to the PLA inputs. Inputs to the FSM come from three sources:

(1) The sign bit of the ACC register in the AUIO macrocell

(2) "test" outputs of the AAU macrocell

(3) A field of the control word, used to modify the FSM.

Outputs of the FSM go to two possible places:

(1) the CC (condition code) output goes to the AUIO macrocell, and must be asserted to enable conditional write instructions

(2) the MOF (middle of frame) and EOF (end of frame) outputs, used to initiate host input and host output operations. These signals go to the Host Interface, discussed in Section 3.7.

3.7 Input-Output and Communications

3.7.1 Off-chip I/O

In designing any system, the issue of external interface is always important. In designing digital IC's, how the circuit will integrate into larger systems is a matter of concern. Circuits designed with the system described here interface in two distinct ways (Fig.3.9).

Signal I/O refers to transfers of data at the sample rate over the signal I/O bus. Here, the timing of data transfers is synchronous with the signal processing IC's clock. The transfers are controlled by strobes generated by the signal processing IC. Since the clock rate is considerably higher than the sample rate, transfer of a number of signals is possible each sample interval. These signals are the sampled-data inputs and outputs of the IC.

Host I/O refers to the transfer of data between the signal processing IC and a host system. The host is typically a microprocessor, with the signal processing IC behaving as a peripheral chip. In general, the host is incapable of sustained data transfers at the signal sample rate, since it has other functions to perform as well. Instead, a slower frame rate is defined. The frame rate is determined by the signal processing IC, which interrupts the host with once-per-frame signals. The host responds to these interrupts by reading and writing data in a buffer (the host interface) contained in the signal processing IC. These data transfers occur asynchronously with the signal processing IC's clock. This simple handshaking arrangement guarantees the validity of the data transfer.

Frame interrupts occur twice per frame (Fig.3.10). The write interrupt (WINT*) occurs close to the end of frame. The host responds to WINT* by writing into the host input interface. This must be performed before the end of the frame. The new data is available to the signal processor after the beginning of the new frame.

The read interrupt (RINT*) occurs just after the beginning of a frame. The host responds by reading from the host output interface. The data read was

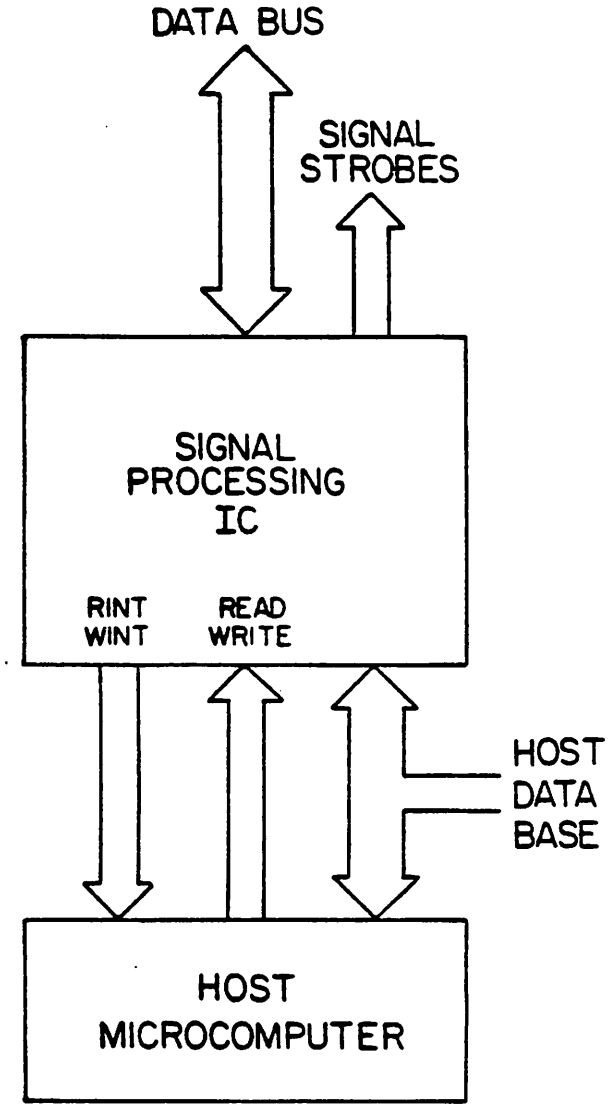


Fig.3.9 Signal and host interfaces

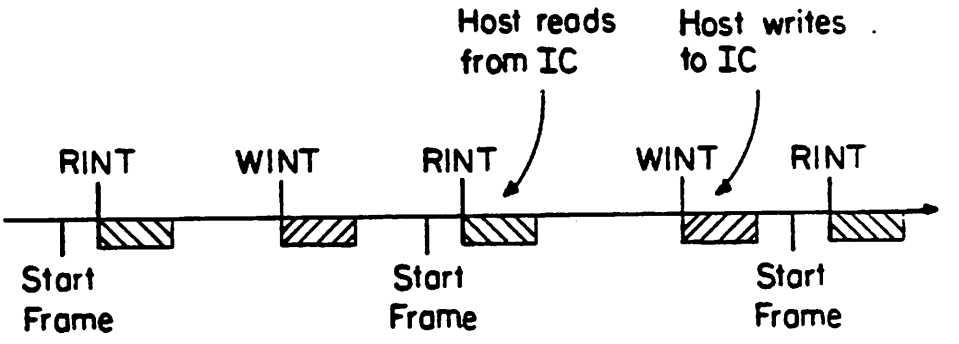


Fig.3.10 Frame timing sequence

collected during the last sample of the previous frame.

The frame discipline described above was designed to support signal analysis and synthesis operations by allowing a frame rate to be defined for the purposes of host I/O. Transfer of sample-rate signals is done on a separate bus to prevent congestion on the host processor's bus. The resulting I/O structure is not completely general, but goes a long way towards integrating the signal processor IC into a system constructed from standard components.

3.7.2 Interprocessor Communication

Internally, the signal processing IC contains one or more processors as shown in Fig.3.11. If more than one processor is used the chip designer needs a way of sending signals between processors, as well as on- and off-chip. This is done by providing for serial communication paths among the processors, and between the processors and the host interface. The number and function of these paths can be selected to suit the application. Each path has a source and a destination.

The possible sources are the MBUS of a processor; the QUOT output of a processor; and the host interface. The possible destinations are the MBUS of a processor; the COEF1 or COEF2 inputs of a processor; and the host interface.

It is not required that the source and destination be different processors. Although it would be pointless to have the MBUS of a single processor as both source and destination, it is often useful to transmit from the MBUS to a COEF1 or COEF2 input on the same processor. This allows two variables local to the processor to be multiplied together. Similarly, one might wish to transmit from the QUOT output to the MBUS to obtain locally the result of a division.

If the source of a serial communication path is the MBUS, a parallel-serial converter must be provided at the source end since the MBUS data is in bit-parallel

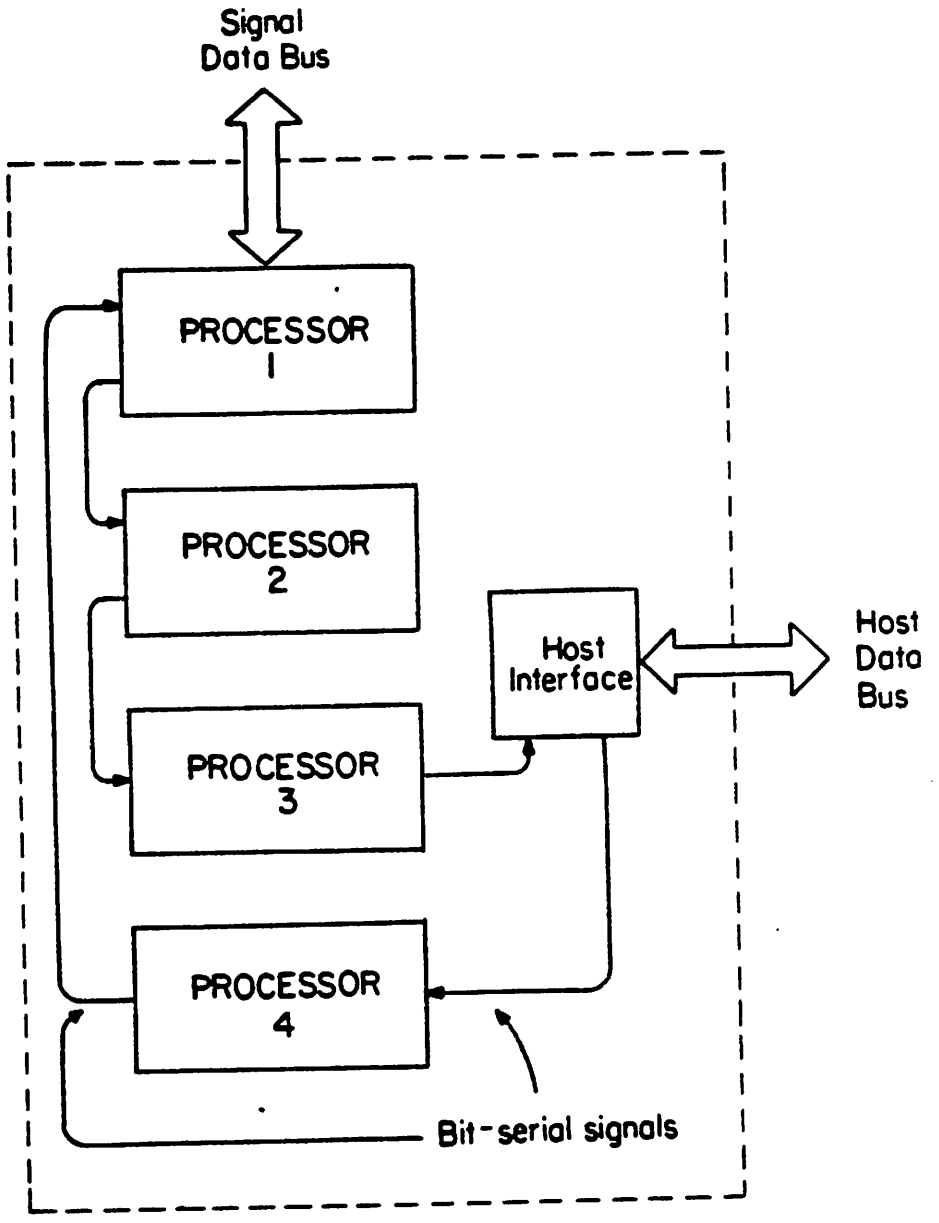


Fig.3.11 Internal communications paths

form. A separate parallel-serial converter is provided for each communication path. Similarly, a serial-parallel converter is provided at the destination end of a serial communication path if the destination is the MBUS.

The serial-parallel and parallel-serial converters are included in the processor I/O section as needed. A separate option provides for a parallel port. This port is used for signal I/O (discussed above). Only one processor per IC may perform signal IO. The parallel port is also used to provide a pathway from the MBUS to the AAU macrocell if pointer-mode addressing is being used. If neither signal I/O or pointer-mode addressing is being performed, the parallel port hardware is not included in the processor IO section.

Provision is made for including a temporary storage latch in the parallel-serial or serial-parallel converters. This is necessary to decouple the timing of the source and destination processors. Thus, the destination processor may access the data being transmitted over the communication path at any time. The data retrieved will be that which was most recently generated by the source processor (taking into account the delay of clocking the bit-serial data). This allows the microprogram for a processor to perform an input or output operation at any point in time, independently of the other processors.

3.7.3 Host Interface

The Host Interface consists of the I/O sequencer and the HI (host interface) macrocell. This arrangement is shown in Fig.3.12. The HI macrocell consists of FIFO buffers for data being transmitted to or from the host. Thus, there are two major pieces to HI, the host output interface and the host input interface. If only host input (output) is used, the host output (input) interface is not included.

The I/O sequencer is a small finite-state machine which is clocked by the EOS

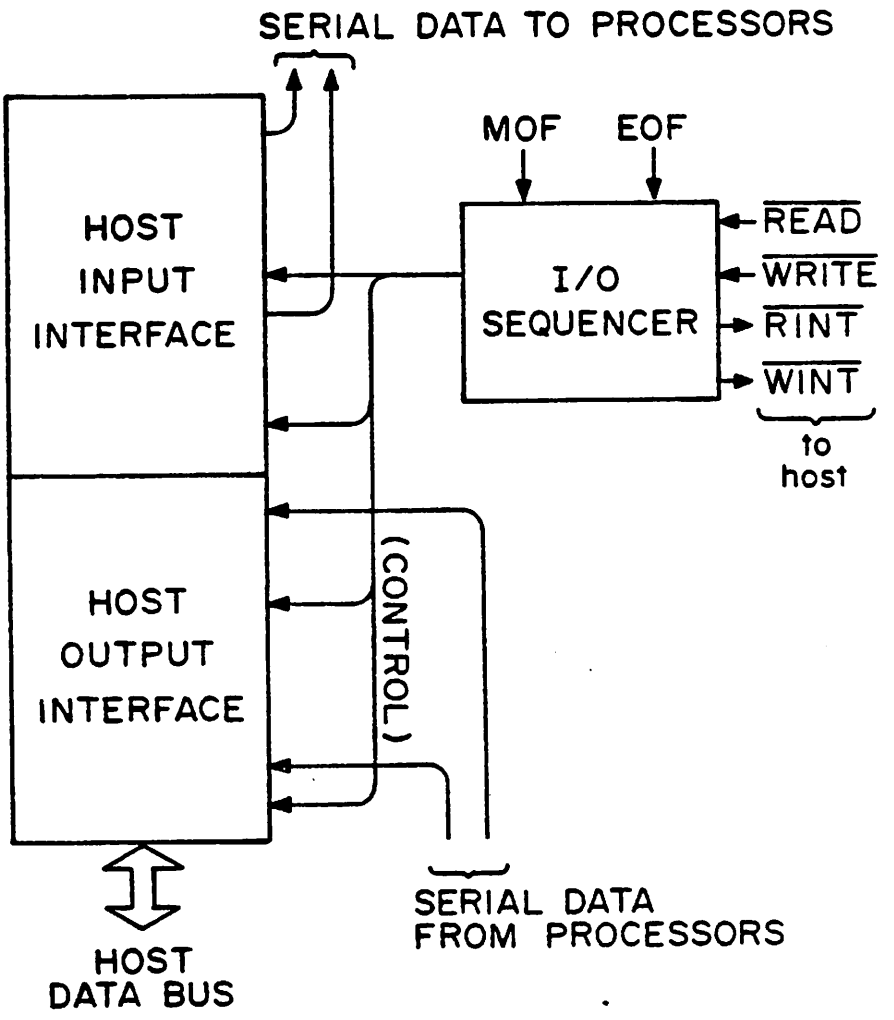


Fig.3.12 Host interface hardware

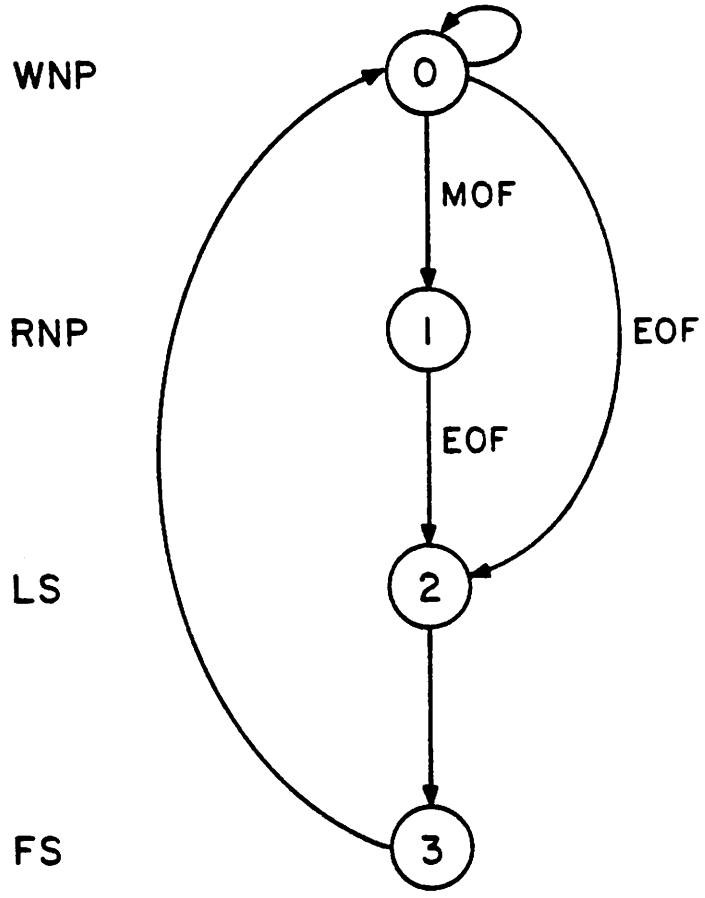


Fig.3.13 State diagram for the host interface

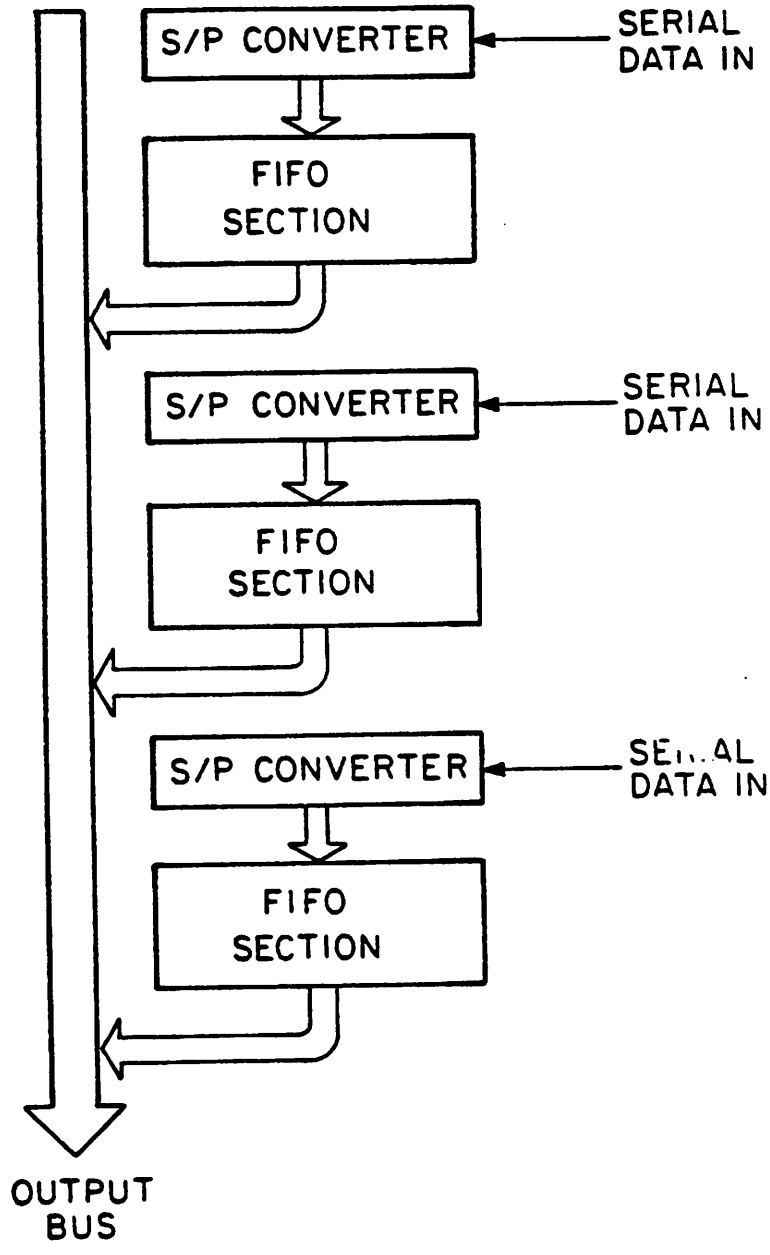


Fig.3.14 Split-FIFO arrangement for the Host Output Interface

(end of sample) signal (discussed in Section 3.3). Thus the sequencer never changes states in the middle of a sample. The I/O sequencer controls the reading and writing of data in the FIFO's and determines the sequence of events during the frame. There are four states as shown in Fig.3.13.

There are four states:

0	WNP	write not in progress
1	RNP	read not in progress
2	LS	last sample of frame
3	FS	first sample in frame

The signals MOF and EOF (Section 3.6) drive the state sequencer through its sequence. Ordinarily, the frame begins in state 3, progresses through states 0 and 1, and ends in state 2.

The role of the host output interface is to collect the various bit-serial signals coming in from the processors; convert them into parallel form; and store them in a FIFO. Once this is done, the FIFO may be read by the host. The timing is arranged so that the FIFO is loaded with data produced by the processors during the last sample of the frame. (Some of the signals may not arrive until the beginning of the first sample of the following frame due to clocking delays of the serial paths.) At the end of the first sample the RINT* interrupt is asserted and the host responds by reading the data out of the FIFO.

It is assumed that the host has read the complete contents of the FIFO before a nominal two millisecond refresh interval has elapsed. Therefore no refreshing is necessary for the host output interface.

The host input interface is somewhat more complex than the host output interface for two reasons. First, the frame consists of many samples; the host input

interface must transmit data to the processors on each sample. The host output interface need only receive data from the processors on the final sample of the frame. Second, the frame may exceed a refresh interval in length. Consequently, the data stored in the host input interface must be refreshed.

To accomodate these added complexities, a double buffered FIFO arangement is used in the host input interface. The MOF flag causes an immediate WINT* interrupt, and the sequencer enters state 1. The host proceeds to write the data into the "master" section of the double-buffered FIFO. At correct moments near the end of the frame, the data are transfered from the "master" to the "slave" FIFO. The timing is arranged so that the processors receive the new data starting with the first sample of the new frame.

The "master" section of the FIFO is not refreshed; therefore the MOF signal must be asserted less than a refresh interval before the end of the frame, but long enough before that the host is always able to fill the "master" FIFO. The "slave" FIFO is refreshed throughout the frame.

In both the input and output interfaces a split-FIFO arrangement was used. The arrangement for the host output interface is illustrated in Fig.3.14.

The idea here is that each serial communication path feeds its own FIFO section. A FIFO section may contain one or more words of storage. Inclusion of more than one word of storage in a single FIFO section is allowed for two reasons: (1) The host interface wordlength may be less that the wordlength of the data coming from the processor, so that more than one word is needed; and (2) an array of data, indexed by the IX or IY counter of the source processor, can be placed in the FIFO section.

Operation of the split FIFO is as follows. For the purposes of writing into the FIFO, the sections operate independently. Each processor sees a separate FIFO which it can clear and write into until it is full. For the purposes of reading out

of the FIFO the sections are joined and act as a single, large FIFO. For reading, the FIFO is initialized at the same time the RINT* interrupt is asserted. The host may then read its entire contents one word at a time.

The host input interface is split in an analogous fashion. The split FIFO approach provides an interesting solution to the problem of merging together several unrelated and unsynchronized streams of data and formatting them into a single block.

3.8 Cell Library

3.8.1 Characteristics of Library Cells

A major part of the macrocell based design system is the cell library. The library consists of 160 cells, each of which was designed by hand and digitized using the graphics editor *Kic* [24]. This may seem at first a rather large number, but many of the cells are either very simple (containing just a few geometries) or are slightly altered versions of other cells in the library.

The cells are designed specifically for the role of being formed into the macrocells described in the preceding sections. There are several ways in which this requirement affects the design of the library cells.

Many of the macrocells (specifically: AUIO, PC, SPC, AAU, and HI) are organized in a bit-slice fashion. These macrocells have several common characteristics:

(1) A GND (ground) line busses along the left side of the macrocell. This is formed from library cells whose names typically end in the suffix ".gnd".

(2) Each bit of the bit slice has a fixed pitch (for the 3u NMOS cell library, the

pitch is 66u).

(3) Generally, polysilicon data busses run vertically through the bit slices.

(4) Aluminum control wires run horizontally across the bit slices. Aluminum wires for power, ground and clocks have the same orientation.

(5) These control wires are driven by cells along the right side of the macrocell. These cells have names typically ending in the suffix ".ctl".

(6) Vdd (power), phi1 and phi2 (two-phase system clocks) bus along the right side of the macrocell, through the ".ctl" cells.

A convenient feature of the bit-slice organization is that distribution of power, ground, clock and control lines is facilitated by the way in which the cells are tiled together. Thus, within limits, it is desirable to collapse as much circuitry as possible into large bit-slice macrocells.

Control inputs to a bit-slice macrocell have terminals along the extreme right-hand side of the macrocell. Data inputs and outputs are along either the top or bottom, connecting to the individual bit slices. There are no terminals along the left side.

Another convenient aspect of the bit-slice approach is that a bit slice array is easily configurable. Word lengths may be varied simply by including the appropriate number of slices. In the case of the HI (host interface) and I/O section of the AUIO macrocells, sections of the bit-slice are stacked vertically to implement circuitry needed to store and transmit global variables. The number and function of the global variables is completely dependent on the application and its

design file description. The fact that the AUIO and HI macrocells may be built up from the design file declarations of global variables is one of the more powerful features of the macrocell system.

The two-phase clock (phi1 and phi2) is distributed globally to all macrocells. At this time, off-chip clock drivers are assumed. This clock is symmetrical and non-overlapping. A timing convention for most signals running between macrocells is the following. At the source macrocell, the signal is latched by phi1. At the receiving end the signal is assumed to be valid during phi2. This convention assures that a fraction of the clock cycle is available for signal propagation between macrocells. The exceptions to this convention are: PC and SPC outputs, which feed ROM address inputs; RAM bitlines and control signals which connect to AUIO; and signals going to and from bonding pad circuits.

The cell library is designed to function at the target clock frequency (5 MHz in this case) over all allowable parameterizations of the macrocells. This requires design for "worst case" in many instances. Improved speed/power performance would be achieved if the individual cells could be parameterized (with regards to transistor sizes, for example).

Each cell is a UNIX file [25]. All cells are contained in a single UNIX directory. Also contained in this directory, and used by the software system, are the files .KIC [26], descriptors (Section 4.2), and .techno (Sections 3.8.2 and 4.3).

3.8.2 Technology Parameters for Macrocells

One goal of the software package is that it be applicable to different semiconductor processes without substantial modification. Clearly, the cell library itself must be designed using the rules and characteristics of the particular process. However, by suitable parameterization of dimensional information, the software

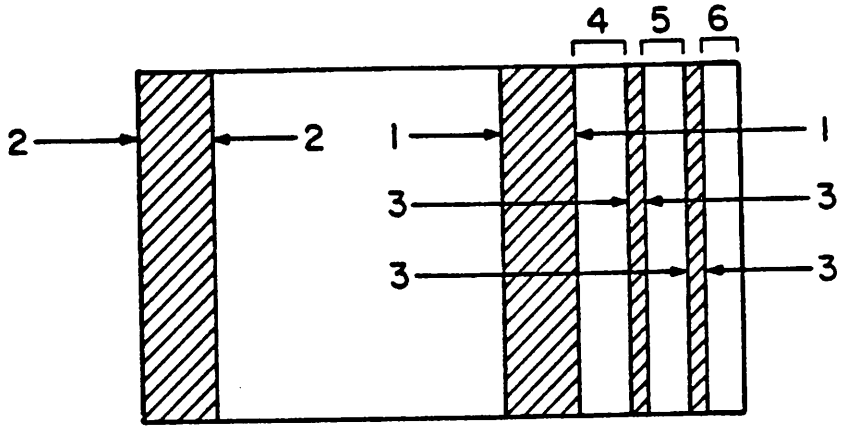


Fig.3.15 Technology parameters for macrocells

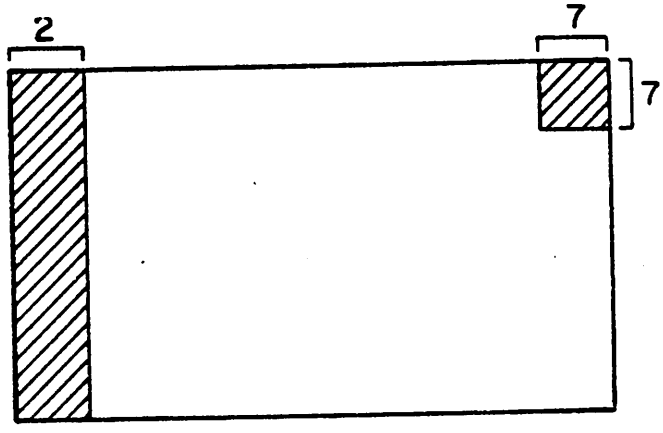


Fig.3.16 Technology parameters for the RAM macrocell

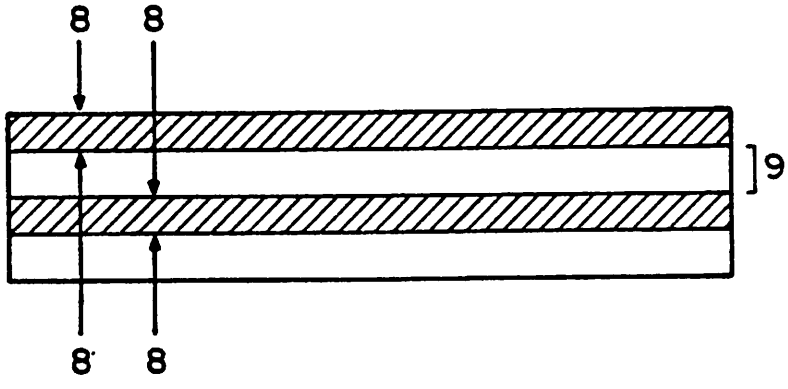


Fig.3.17 Technology parameters for the bonding pad groups

package is technology-independent.

A set of technology parameters is provided that describes the width and position of power, ground and clock conductors with respect to the bounding boxes of the macrocells. (Additional technology parameters relating to signal wiring are described in Section 4.3.3.) Dimensions are given below in units of lambda (1.5 micron) for the 3 micron NMOS library.

For all macrocells except RAM and the bonding pad groups, the following parameters apply (Fig.3.15):

(1) width of Vdd conductor -- 20

(2) width of ground conductor -- 20

(3) width of clock conductors -- 4

(4) spacing from Vdd conductor to phi2 -- 3

(5) spacing from phi2 to phi1 -- 8

(6) spacing from phi1 to edge of bounding box -- 7

For the RAM macrocell, the following are used (Fig.3.16):

(2) width of ground conductor -- 20

(7) width of Vdd conductor -- 9

For the bonding pad groups, the following apply (Fig.3.17):

(8) Width of Vdd and ground conductors -- 50

(9) Spacing between Vdd and ground conductors -- 117

Chapter 4 – Software Package for the Macrocell Design System

4.1 Design File and Front-end Software

4.1.1 The Design File

A primary goal in creating the format for the design file is that it be easy to prepare and modify. Otherwise, the advantages of automatic layout generation would be partially cancelled. The design file is the main user interface to the system since it is the input to both the emulator and the silicon compiler. In constructing this interface it is helpful to consider the user of the system, and the purpose for which it is used.

Although practical signal processors operate in real-time, much algorithmic research involves non-real-time simulations. There is, however, a gap between that which may be simulated and that which is efficient to implement on a real time architecture. Bridging this gap is a major goal of the system. A designer who knows both the requirements of the algorithm and the capabilities of the hardware is in a position to understand the trade-offs involved and therefore optimize his design.

The design system addresses this situation by providing an interface that allows both simulation and hardware specification. The level of expertise required to use the system is consistent with that expected for a typical signal processing researcher. No prior knowledge of integrated circuit design is necessary.

The design file format may be viewed as a programming language for multiprocessor IC's with a specific architecture. Programming languages generally become easy to use if they involve abstraction of complex concepts. Several examples of abstraction in the design file are: symbolic names for processors, variables,

constants, and instructions; a register-transfer description of the data-path; and a simple way of specifying interprocessor and off-chip communications. In each of these cases, much of the hardware complexity is hidden from the user. Yet very little efficiency is sacrificed considering the amount of simplicity gained.

The design file begins by declaring global variables, and specifying which of these are used for off-chip communications over the host or signal interfaces. Globals which are not used for off-chip communications are presumed to be used for interprocessor communications; however, the source and destination processors are not specified explicitly. Instead, this information is implicit in the microcode for the processors which reference these globals.

The following is an example of the global declaration and I/O specification part of a design file.

```
.global
begin

  /* host input globals */
  ks[10]<8>;     /* reflection coefficients */
  pitch<8>;      /* pitch period; zero means unvoiced */
  amplitude<16>; /* excitation amplitude */

  /* signal output global */
  output<12>;    /* synthetic speech output */

end
```

```
.io <8> /* I/O description */
begin
   infile : ks, pitch, amplitude : host_out;
   outfile : output : signal_out;
end
```

(Comments begin and end with the sequences "/*" and "*/".)

The keyword ".global" introduces global declarations. The declaration "ks[10]<8>;" declares an array of ten eight-bit global variables. Arrays of global variables may be used for host input and output, and are indexed by the IY or IX counters of the destination or source processor. Array globals are not permitted for signal I/O or interprocessor communications. The declaration "amplitude<16>" declares a 16-bit scalar (non-array) global variable.

The sequence ".io <8>" introduces the I/O declarations, and indicates that an eight-bit host interface is to be used. (The other option is 16 bits, covering the two most common microprocessor bus widths.) "infile" and "outfile" are UNIX file names to be used for simulations. These are ASCII text files, so that test cases may be easily prepared.

Following these global and I/O specifications is a definition of each of the processors. Each processor section begins by specifying the processor wordlength, and declaring local variables and constants. An optional section specifying a FSM is next, followed by the symbolic microcode for the main program and subprogram, if used.

The processor wordlength must be an even number no larger than 32. Often, this is a critical parameter affecting the dynamic range and noise performance of the IC. Through simulation, the designer may optimize this value.

Local variables and constants all have the same wordlength as the processor itself. The variables may be either scalars or arrays. An array variable is usually used in conjunction with indexing by the IY or IX counters.

The following is an example of the initial section of a processor declaration, showing the local and constant declarations.

```
.processor : filter <18>
begin

    .local          /* local variables for processor */
    begin
    _pitch, _amplitude; /* local copies of pitch, amplitude */
    c[10];          /* lattice filter state variables */
    end

    .constant       /* local constants for processor */
    begin
    ONE = 1;
    FRAMELENGTH = 180;  /* gives frame of 180 samples (22.5 ms) */
    end

    /* . . . (fsm declaration and microcode go here) . . . */

end  /* end of processor "filter" */
```

The first line above declares an 18-bit processor with symbolic name "filter". The

symbolic processor name is useful when running the emulator in debug mode.

A section may be included to define an FSM. This is a powerful device, since symbolic instructions are defined which modify the FSM state. These instructions may then be used in the microcode for the main program and subprogram.

The FSM is described by giving a symbolic name to each state variable, and providing equations which specify the state transitions. These Boolean equations are translated by the software into a truth table format suitable for programming the PLA-based FSM.

One bit of state for the FSM is usually called "cc", for "condition code". It is this bit which must be set (true) if a conditional write operation is to be enabled. Also, one of the FSM inputs is called called "sign", and refers to the sign bit of the accumulator. Thus, recalling that the contents of the accumulator is in two's-complement form, if sign is zero, the accumulator is non-negative.

Simple examples of FSM instructions which may be defined are "SET" and "AND" instructions. The "SET" instruction sets "cc" for non-negative accumulator. The "AND" instruction performs a logical and of "cc" with the condition that the accumulator is non-negative. An FSM declaration defining just these two instructions would be the following:

```
.fsm
begin
SET : cc = !sign;
AND : cc = !sign & cc;
end
```

More complex FSM instruction sets may be defined for specific applications.

If host I/O is used, one of the processors must be configured with an FSM

which defines two outputs called mof (middle-of-frame) and eof (end of frame). The eof signal is asserted during the sample before the last sample in the frame, and results in a RINT* (read interrupt) being sent to the host a short time later. The mof signal is also needed if host input, or IY-indexed host output is used. mof is asserted no longer than a refresh interval (two milliseconds) prior to eof. However, the interval between mof and eof must be sufficient for the host to write all necessary input data into the host input interface.

Following the variable, constant, and FSM declarations is the microcode for the processor. Because of the pipeline nature of the data path, the instruction set is divided into groups, with one instruction per group allowed in a single line of microcode. Each line of microcode corresponds to a single clock cycle.

These are the instruction groups:

(1) Memory group — reads and writes, which may be indexed or conditional.

(2) The SOR group, which controls the contents of the SOR register.

(3) The accumulator group, which controls the contents of the ACC register.

(4) The "le" (latch enable) instruction, controlling the MIR latch.

(5) The "mbus" group, which enable signals onto the processor's "mbus" (by default, mbus=acc);

(6) The user-defined FSM group.

(7) The "aip" (accumulate-if-positive) instruction

(8) Instructions setting up a correspondence between global variables and the coefficient inputs, quotient outputs, or "mbus"; or assigning the mbus to IY (pointer mode only). (several non-conflicting instructions from this group may be used in the same line.)

Thus, a single line of microcode may contain many of the above instruction groups, as in the following:

```
rx(C[0]), sor:=mor>4, acc:=acc+sor, mbus=pitch, le, SET;
```

There are several thing happening here:

An IX-indexed read from the local array C[] -- group (1);
Assigning MOR, shifted right 4 bits, to SOR -- group (2);
Adding SOR to the accumulator -- group (3)
Enabling the global variable "pitch" onto the mbus -- group (5);
Enabling the MIR latch -- group (4);
Executing the user-defined FSM instruction "SET" -- group (6);

As with any assembler-level programming, familiarity with the processor architecture is needed in order to write efficient code.

In the design file, the main program may be begin with one of the following three types of sequences:

```
.main_pr
.main_pr <8>
.main_pr <*>
```

In the first form, no IY indexing is used by the processor. In the second form, the IY indexing is used in counter mode with modulus (decimation ratio) eight. In the third form, the IY indexing is used in pointer mode.

Similarly, the subprogram might begin as follows:

```
.sub_pr <10>
```

In this case, there would be ten iterations of the subprogram.

Within the main program or subprogram, commas are used to separate the different instructions within a single line of microcode, and a semicolon terminates the line. The entire main program or subprogram is bracketed by the "begin" and "end" keywords.

In summary, the design file contains declarations for global variables and I/O specifications, followed by a block for each processor. The processor block defines local variables and constants; the FSM; and the microcode for the main program and subprogram.

4.1.2 The Emulator

The emulator allows simulation of the signal processing IC from the design file. This is done in non-real-time. For a typical design file, on a VAX 11-750 or 68000-based Sun Workstation, the ratio of emulator simulation time to real time might be 500 or 1000.

Upon invoking the emulator, the design file is read in and checked for errors. If there are no fatal errors, an interactive mode is entered. A number of debug commands are available, including:

Tracing a variable;

Printing a variable;

Setting a breakpoint at a point in the source program;

Setting the value of a variable;

Running a simulation for a given number of samples;

Aliasing a debug command to a shorter form.

These debug commands considerably reduce the amount of time needed to prepare a design file for a given application. Also, single-stepping through a program is a very good way to familiarize oneself with the architecture of the data path.

Much care was taken to ensure that the emulation exactly matches the hardware itself. Truncation, round-off, and other arithmetic details are emulated precisely.

The ability to simulate the signal processing IC from the design file, rather than from a description of the circuit layout, is critical to the success of the entire macrocell approach. The reason is that signal processors require exceptionally long test sequences to fully verify performance — perhaps several million clock cycles, or more. For this reason, verification at the circuit level is essentially impossible. Thus the only practical choice for signal processor design is to verify performance from a higher-level description such as the design file. The layout, generated automatically from the design file, is "correct by construction".

4.1.3 The silicon compiler first pass

The compiler first pass resembles the emulator in its initial phases; in fact, many subroutines which parse the design file are shared between the two programs. After parsing, the first pass generates an intermediate file, a process which is analogous to intermediate code generation in a compiler for a programming language. Besides assembling the microcode into binary, the macrocells must be specified at a hardware level. This involves extracting hardware parameters such as the following:

Processor word lengths

Processor data memory sizes

Constant programming of data memory words

Configuration of IX and IY counters in the AAU

Translation of FSM declaration into and-plane and or-plane code

Program counter modulus

Subprogram counter modulus

Programming of microcode in ROM

Inclusion of parallel-serial and serial-parallel registers in AUIO

Constructing the host interface from the host-io declarations

In Section 4.2, the format for the intermediate file is described. Section 4.3 discusses the placement and routing of the processor assemblies.

4.2 Intermediate File and Macrocell Assembly

The first pass of the compiler outputs an intermediate file. This intermediate file is a hardware description defining the macrocells and how they are interconnected. The macrocells are defined by describing how the library cells are to be tiled (arrayed) to form the macrocells. The interconnect is defined by providing unique names for the macrocells' terminals, and giving a net list.

Several examples of tiling programs exist [27]. Tiling is a useful method for generating a complex block of circuitry from smaller cells. The two most common organizations for a block of circuitry are the bit-slice organization (useful for arithmetic units and register files) and array-type structures such as RAM's, ROM's and PLA's. The second pass of the compiler employs a tiling routine that can generate either of these structures.

Two features (generally lacking in existing programs) were needed for the tiling routine used here. One is that terminal information must be preserved by the tiling process, so that the assembled macrocells could then be used as input to placement and routing routines. The second is that dimensional information must be encapsulated in an easily modifiable form. This makes it possible to use the software with cell libraries designed in different technologies.

To this end, dimensional information defining the sizes of library cells and their terminal locations is contained in a descriptor file. The output of the first pass contains no dimensional information, and is therefore technology independent. The descriptor file contains an entry for each cell in the cell library. A typical entry might look like this:

```
cell ix.ctl 66 88
right eos 66 25
right inc 66 72
```

The keyword "cell" begins the entry for the cell "ix.ctl". The numbers "66 88" are the tesselation dimensions of the cell. The x-dimension is 66 and the y-dimension 88. Tesselation dimensions specify the distance between origins of cells in adjacent rows and columns of an array. Cells are assumed to contain geometries mostly in the first quadrant, with the origin being at or near the lower left corner of the cell.

Note however that the tesselation dimensions are not the same as the dimensions of the cell's bounding box. Thus when cells are tiled, their bounding boxes may overlap, abutt, or be separated by a gap.

The keyword "right" denotes a terminal on the right edge of the cell. Thus the

cell "ix.ctl" has two terminals, one of which is named "eos" and has x and y coordinates 66 and 25.

Given the information in the descriptor file, the macrocells are defined in the hardware description by simple listing the cells in each row of the array. This is done row by row, starting at the bottom of the array. The cells in a given row are listed from left to right. The following is an example of a hardware description for a program counter:

```
begin
counter.gnd, counter.0, counter.0, counter.0, pc.ctl;
pc.0, pc.1, pc.1, pc.1, pc.2;
end
```

To make more complex examples readable, a shorthand notation is allowed for cells within a row. The following description is equivalent:

```
begin
counter(.gnd, .0, .0, .0), pc.ctl;
pc(.0, .1, .1, .1, .2);
end
```

The tiling routine assembles macrocells from these descriptions. The location of terminals relative to the origin of the macrocell is determined. These terminals are given unique names by appending the terminal name in the descriptor file to the macrocell name and the row and column in which the terminal occurs. For example,

pr0_pc[1.5]_reset

refers to the terminal "reset" in the first row, fifth column of the macrocell pr0_pc (meaning the program counter for processor number zero).

The net-list portion of the hardware description uses these unique terminal names to specify interconnection. A unique node number is assigned to each set of terminals which are interconnected. This allows net-list verification subsequent to routing using a circuit extractor program such as Mextra [28].

Power, ground and clock connections are not specified in the same way as the signal terminals. Instead, entry points for power, ground and clock connections bear a standard relationship to the bounding box of the macrocell, and are handled separately during placement and routing.

4.3 Placement and routing

4.3.1 Outline of placement and routing strategy

The problem of placement and routing of integrated circuitry has received much attention in recent years. The reasons for this are clear. When performed by hand, with only a graphics editor for assistance, placement and routing can be a tedious and time-consuming affair. Thus a software system to perform placement and routing has been the goal of a number of research projects. Unfortunately, the software problem for the general case has proved to be difficult, and satisfactory general purpose placement and routing systems are not available to the typical IC designer.

The macrocell-based design system does not require a general-purpose place and route program. Instead, it is only necessary to assemble those IC's the system

is capable of producing. This more constrained problem is much more easily solved.

The approach used here is to subdivide the problem into two phases. In the first phase, a combination of ad hoc and algorithmic methods is used to assemble the individual processors. In the second phase, the processors, the host interface (if included) and the bonding pads are assembled into the final IC. The first phase is performed automatically by software. The second phase has been performed by both by hand and by software [32]. In either case, the software package is capable of verifying whether the second phase wiring was performed correctly.

Because inter-processor signals are bit serial, the amount of wiring to be performed in the second phase is relatively minor. It is feasible for a designer to perform this final step by hand in less than a day's work, and verify completely that the hand wiring is correct.

4.3.2 Processor Floorplan

The individual processors are assembled from the AUIO, RAM, PC, ROM, and optional SPC, AAU and FSM macrocells. In order to perform this assembly efficiently, a fixed floorplan approach is used. The floorplan for the fully configured case is illustrated in Fig.4.1. The floorplan consists of a left side (containing AUIO, RAM, AAU and FSM); a central wiring channel; and a right side (containing PC, ROM and SPC). Wiring internal to either the left or right side is accomplished by repeatedly calling a simple river router. The central channel is wired by a general purpose channel router, based on Kuh's channel routing algorithm [29].

In order to employ a river router in the assembly of the left and right sides, it is necessary to have prior knowledge of the interconnection pattern. This

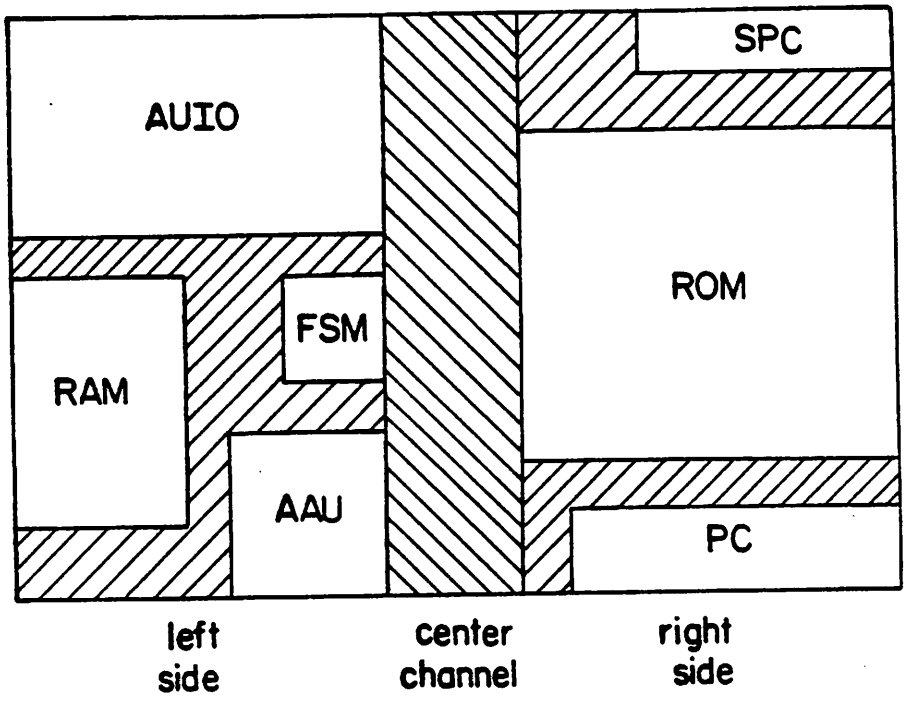


Fig.4.1 Processor floorplan

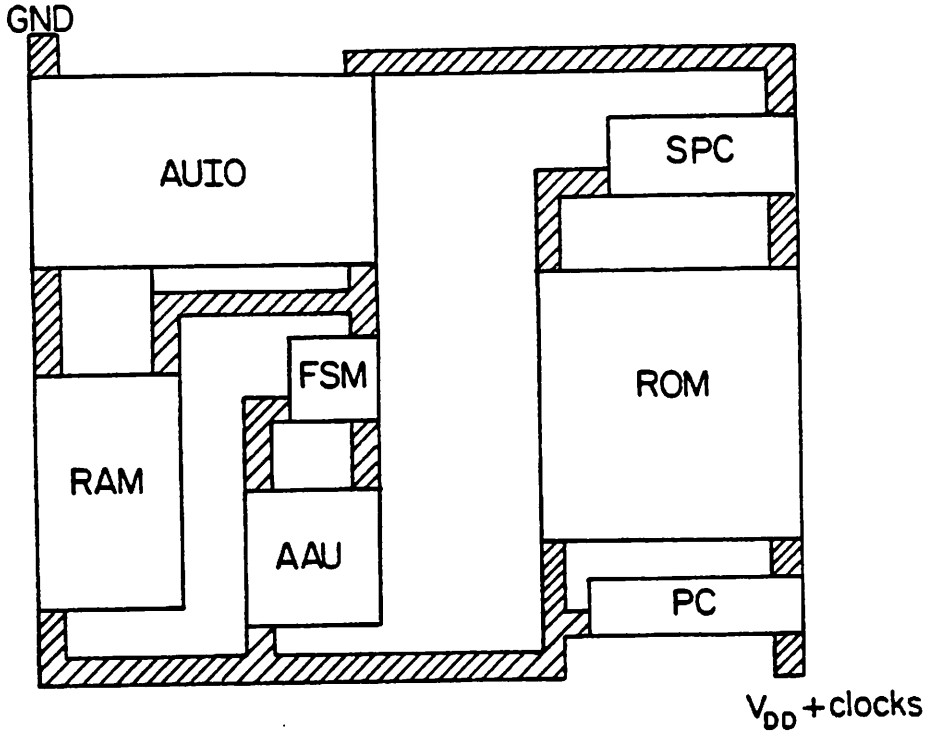


Fig.4.2 Power, ground and clock li. es

knowledge is available since there is a limited number of ways in which these macrocells may be organized.

A net is defined to be a collection of wires which share the same source and destination. A source and destination are considered to be a given side (left, right, top, or bottom) of a given macrocell. Thus, all the wires going from the top of PC to the bottom of ROM are in the same net, whereas wires going from the top of ROM to the right side of the PC are in a different net. Many wires, including all wires which connect the left and right sides, are brought to the central channel.

For the fully configured case, the following nets may be identified:

	source	destination	function
left side:			
	AUIO top	channel	IY inputs (pointer mode)
	AUIO top	(external)	signal I/O bus
	FSM top	FSM top	FSM feedback
	FSM top	channel	FSM inputs and outputs
	AUIO bottom	RAM top	RAM control and data
	AAU top	channel	address offset
	AAU bottom	RAM right	effective address
right side:			
	PC top	ROM bottom	lower ROM address
	SPC top	ROM top	upper ROM address
	PC right	channel	reset line
	PC right	(external)	end-of-sample line
	SPC right	ROM top	SPC controls
	ROM top	(external)	I/O strobes
	ROM top	channel	all other controls

(The functions of these signals are described in detail in Appendix B.)

In addition to the above nets, the right sides of the AAU and AUIO abutt directly to the central channel. Many control signals connect to terminals on the right sides of these macrocells and are routed through the channel.

There are four places where signals enter and leave the processor assembly.

These locations, listed as "external" in the above chart, are as follows:

(1) At the top of the central channel are connections for bit-serial inputs and outputs and the reset signal (used to synchronize the PC's of the various processors).

(2) At the top of the processor assembly, directly above the AUIO macrocell, are terminals for the parallel signal I/O bus. This bus is only used for exactly one processor per IC. If used, a bus enable signal also appears at the top of the central channel.

(3) Along the right edge of processor assembly is generally a group of ROM outputs used as strobes for I/O operations. These strobes go to other processors, the host interface, or to strobe output pads.

(4) Also along the right edge of the processor, closer to the bottom edge, will be an EOS (end of sample) output, used for timing host input/output operations.

Because of special requirements regarding conductor width and series resistance, power, ground and clock signals are handled as special cases during processor assembly. This is fairly straightforward in the fixed floorplan approach. Fig.4.2 shows how these connections are arranged.

4.3.3 Technology Specification

As has been stated earlier, it is desirable that the software for a macrocell system be technology independent. Then the same software, in theory, could be used for any technology simply by redesigning the underlying cell library. The descriptor file, described in Section 3.5, provides technology independence for the macrocell tiling software. It is also necessary to define the the technology for the

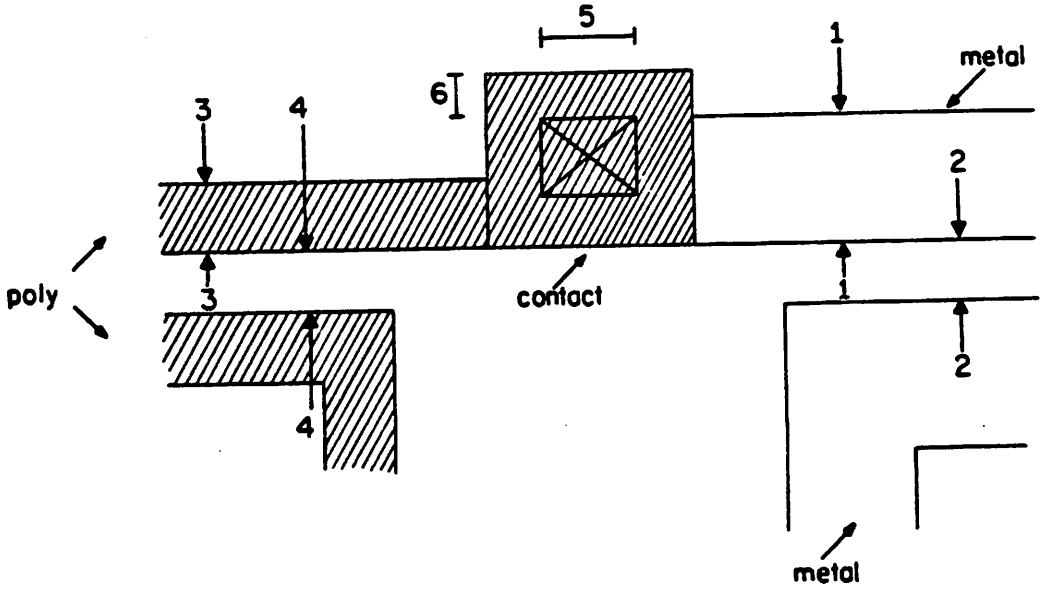


Fig.4.3 Parameterization of critical dimensions

purpose of placement and routing.

All target MOS processes considered here contain at least two levels of interconnect: polysilicon and metal (usually aluminum). Although many processes offer additional levels of interconnect, for generality it is best to assume that only these two will be available.

The first step in specifying a set of geometric design rules for an MOS process is to decide to what resolution features will be digitized. This figure (called lambda by Mead and Conway [30]) is usually taken to be some small integral submultiple of the drawn gate length. Once a value for lambda is established geometric design rules may be specified in terms of integral multiples of lambda.

For the purposes of routing, the technology can be specified by the following six critical dimensions (illustrated in Fig.4.3):

	3u NMOS (lambda=1.5u)	3u CMOS (lambda=1.0u)
1. Metal width	3	3
2. Metal spacing	3	3
3. Poly width	2	3
4. Poly spacing	2	3
5. Contact size	2	3
6. Contact surround	1	2

(The above critical dimensions are given in lambdas for the processes used for this project.)

4.3.4 Channel Routing Algorithm

Of all the possible routing situations, one of the most common is that of channel routing. Suppose we have a rectangular channel with terminals along the top and bottom rails. The position of the terminals, and the alignment of the top and

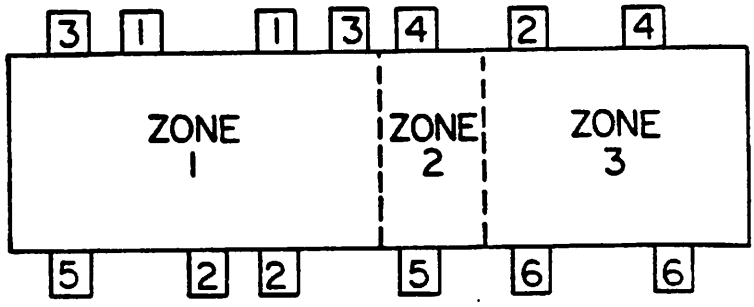
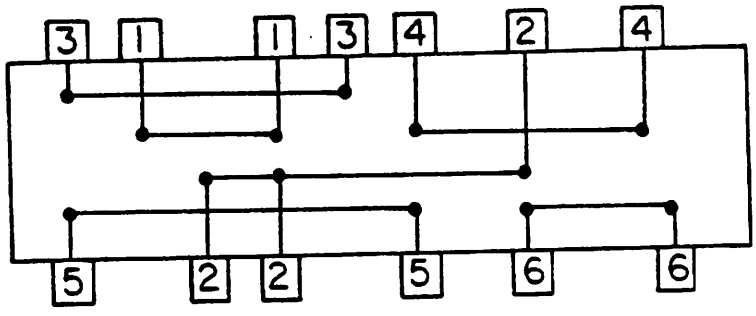
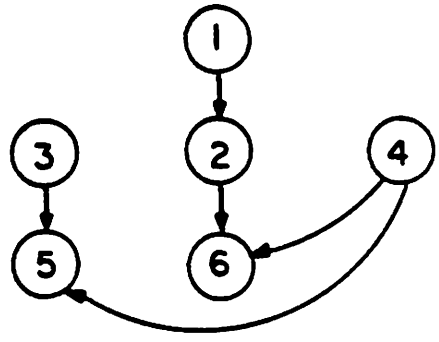


Fig.4.4 Routing example showing zone decomposition

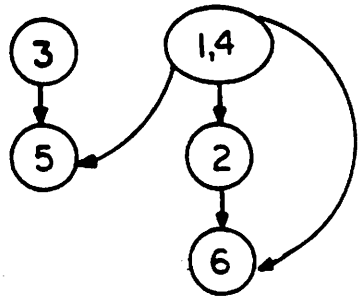


tracks	nodes
1	3
2	1,4
3	2
4	5,6

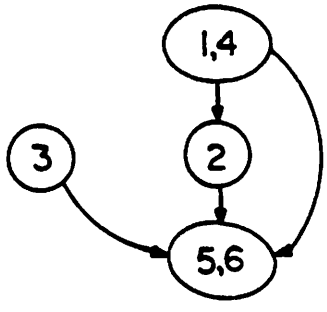
Fig.4.6 Final route



INITIAL VERTICAL CONSTRAINT GRAPH



MERGED GRAPH AFTER ZONE 2



MERGED GRAPH AFTER ZONE 3

Fig.4.5 Merging the vertical graph

bottom rails is fixed. The height of the channel may be determined by the router. The channel router is given a net list which specifies the desired interconnection among the terminals. In other words, the set of terminals is partitioned into disjoint subsets, each of which is a circuit node (sometimes called a net). In addition, it is specified that some nodes must be brought to the extreme right edge, or the extreme left edge, of the channel for external connection. However, the positions of these external connections may be assigned by the router.

In general, the horizontal dimension of the channel is much longer than the vertical dimension. Thus, if two levels of interconnect are available, the horizontal wires, or tracks, are routed using the interconnect level with the lower sheet resistance (usually aluminum) and the vertical wires in the level with higher sheet resistance (usually polysilicon).

(An optimization, not used here, would be to attempt to use aluminum for some of the vertical wires as well.)

The above scenario is a common one in IC layouts, and describes the center channel of the processor assembly as well (note that the orientation is rotated 90 degrees). Unfortunately, most channel routing algorithms add two further constraints to the problem:

(1) The terminals are aligned with a coarse grid, the dimensions of the grid being the minimum allowable contact-to-contact spacing.

(2) The density of terminals along the top and bottom rails is high, with a terminal in nearly every possible location.

The center channel of the processor assembly satisfies neither of these constraints. Fortunately, a very efficient channel routing algorithm due to Kuh [29] can be modified to eliminate these constraints.

The channel routing problem can be thought of as an attempt to assign nodes to tracks. More than one node may be assigned to the same track if the terminals for the different nodes occupy disjoint regions in the channel. The goal is to reuse as many tracks as possible, to obtain the minimum channel height.

The channel router used for the processors never assigns more than one track per node. It can be seen that in the case of the processor assembly, the need for multiple tracks per node will never arise if the signals emerging from the right side of the processor assembly (the ROM outputs) are ordered correctly with respect to their destinations. Since the ordering of the ROM outputs may be easily specified, it is not necessary to use a channel router capable of assigning more than one track per node.

The first step in Kuh's algorithm is to form a vertical constraint graph. The nodes of the circuit are nodes in this directed graph, with an edge from one node to another if the first node has an upper-rail terminal overlapping in horizontal position a lower-rail terminal of the second node. The significance of the vertical constraint graph is the following: if there is a directed path from one node to another in this graph, then the track assigned to the first node must be above the track assigned to the second node.

Note that far fewer vertical constraints exist if the density of terminals along the rails is low.

Situations where a single track per node is insufficient to achieve a route will have a cycle in their vertical constraint graph.

Once the vertical constraint graph is formed, it is now possible to proceed with a track assignment for the nodes. We first divide the channel into zones. Within a zone, only one node will be assigned to a track. Zone decomposition is accomplished by starting at the left edge of the channel, and assigning terminals to the first zone by scanning left to right. Terminals are included until (1) the current

zone contains the rightmost terminal of a node and (2) the next terminal would be the leftmost terminal of a node. This next terminal will be the first terminal in the next zone. The process continues until the entire channel is subdivided into zones. Fig.4.4 illustrates the zone decomposition for a simple channel routing situation.

Note that if it is specified that a node must be brought to the extreme right edge (or left edge) of the channel, then that node has no rightmost (or leftmost) terminal for the purposes of zone decomposition.

An important concept, introduced by Kuh, is that of merging the nodes in the vertical graph. This is a transformation wherein more than one node in the netlist occupies a single node in the vertical graph, the implication being that these nodes will share a track in the final track assignment. Starting with the second zone, as each zone is determined the vertical graph is merged. The criterion for merging the vertical graph is to minimize the length of the longest vertical path. This will tend to minimize the total number of tracks used in the final result. Heuristics are used to satisfy this criterion.

Fig.4.5 shows the initial vertical graph, and a sequence of vertical graph mergers, for our example. Each merger occurs after determination of a new zone. The total number of tracks used will be the total number of nodes in the merged vertical graph after the final zone is processed. The tracks must be assigned so that arcs in the final version of the vertical graph always go from higher to lower tracks.

4.4 Software Package Summary

Described in this chapter was the software system used to support the cell library described in Chapter 3. There are two primary motivations for providing this software package. First, the detailed design of an IC layout is time consuming and error-prone. Thus, to the extent that software may be used to produce the layout without compromise in functionality or area-efficiency, the cost of designing the software is easily justified from an economic point of view. Second, integrated circuit design is a highly technical skill for which there is currently a shortage of qualified individuals. Thus, it makes sense to involve those without a specific IC background more directly in the IC design process.

An important goal of this effort is to provide an interface to the software system which is general, simple and abstract. The design file format qualifies in these respects. It is general in the sense that it allows expression of any reasonable combination of the library cells. Its simplicity stems from the fact that it is essentially a programming language and that a relatively small file (usually a few pages of text) is all that is required to represent most applications. As with any programming language, abstraction is used judiciously to hide unimportant implementation details, without an undue compromise in efficiency.

The emulator provides an interactive environment for the development of design files. This is important, since without adequate feedback to the designer the design files would be difficult to prepare. The availability of non-real-time simulations is very useful in performance evaluation.

The compiler combines many techniques that are often applied separately in the course of an integrated circuit design: microcode assembly, PLA optimization, tiling, placement and routing. By integrating these techniques into a single software program a large manpower savings results.

From a software design standpoint, a major decision for the compiler was to divide compilation into two passes, with an intermediate file being the output of the first pass and input to the second. It should be noted that the second pass, with its module generation and routers, embodies most of what is traditionally thought of as a silicon compiler.

Presented in the following chapter are case histories of the use of the macrocell system for several classical signal processing applications.

Chapter 5 – Case Histories and Conclusions

5.1 Digital Audio Equalizer

Other than test circuits programmed with diagnostic microcode, the first IC designed with the macrocell system was a single-band digital audio equalizer. The algorithm involves a two-pole, two-zero canonical form filter. A signal flow chart of this filter is given in Fig. 5.1.

The five fourteen-bit filter coefficients are inputted from the host processor, and can be selected to give boost, cut, or shelf frequency response characteristics. A two-millisecond frame rate is used, this interval being short enough to allow the coefficients to be changed gradually without audible discontinuity. A 50 KHz sample rate is used.

Two versions of this filter were designed. The design files (Appendix D) were prepared by Mats Torkelson [31]. Both versions used a single processor with a twenty-four bit wordlength and an eight-bit host interface. A finite state machine is included, whose sole purpose is to generate the EOF and MOF flags for the host interface, since there are no conditional operations in the algorithm itself.

The first version implemented the entire application in the main program of the processor, without using a subprogram. The five coefficients were implemented as five scalar global variables.

A layout plot of this circuit is shown in Fig.5.2. The wiring of the bonding pad groups and host interface to the processor assembly required six man-hours of manual effort. Other than this, layout generation was performed entirely by the compiler. Net-list extraction using the program Mextra [28] was used to verify that the manual wiring was correct, eliminating a possible source of error.

Observing that the host interface and the I/O section of the processor's AUIO

macrocell consumed large amounts of area in this design, a second design file was prepared. A single five-element global array was used for the coeficients, rather than the five scalar globals. These globals were indexed by IX, with one multiply being performed during each subprogram iteration.

A layout plot of the resulting circuit is shown in Fig.5.3. The smaller area of this second implementation is indicative of the fact that global variables are a significant expense in terms of area, and their use should be kept to a minimum.

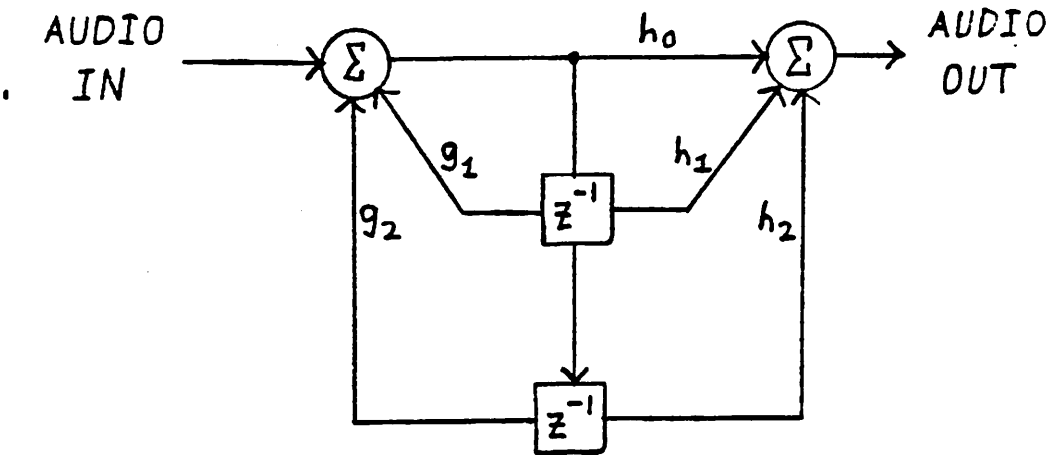


Fig. 5.1 Signal Flow Chart for Digital Audio Equalizer

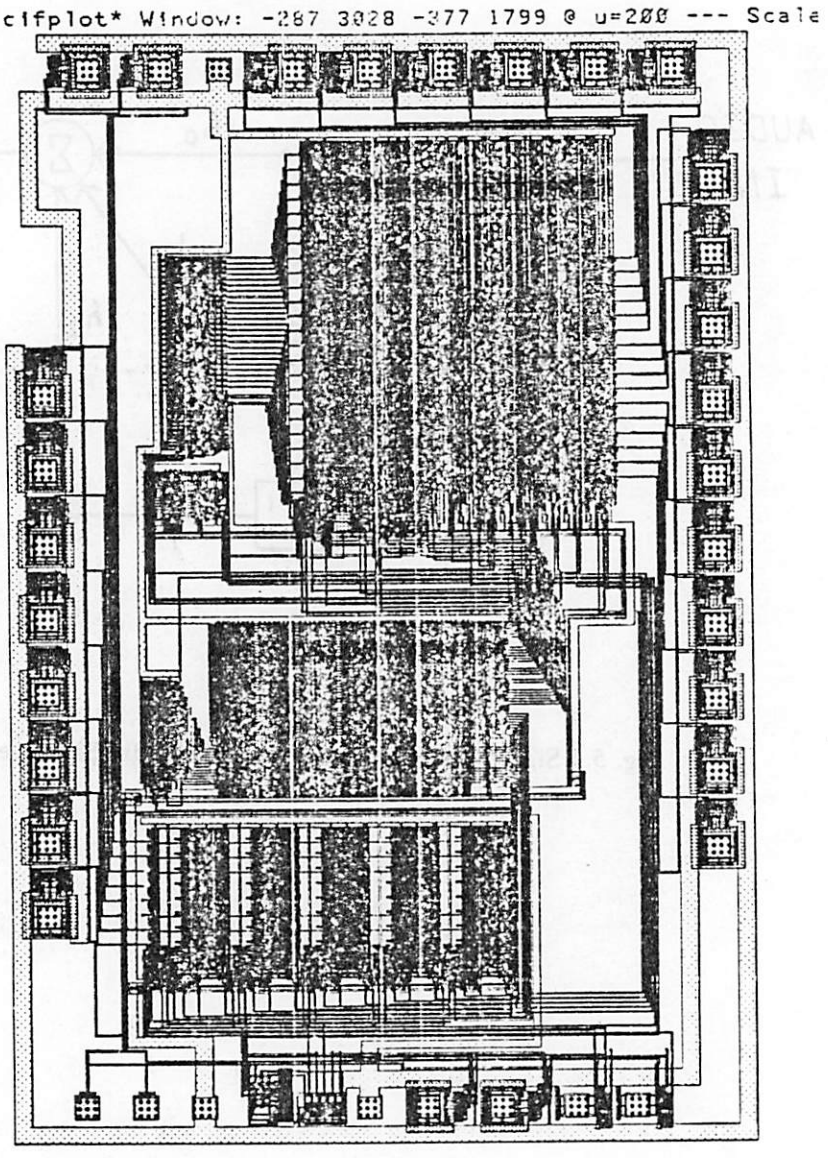


Fig. 5.2 First Version of the Digital Audio Equalizer

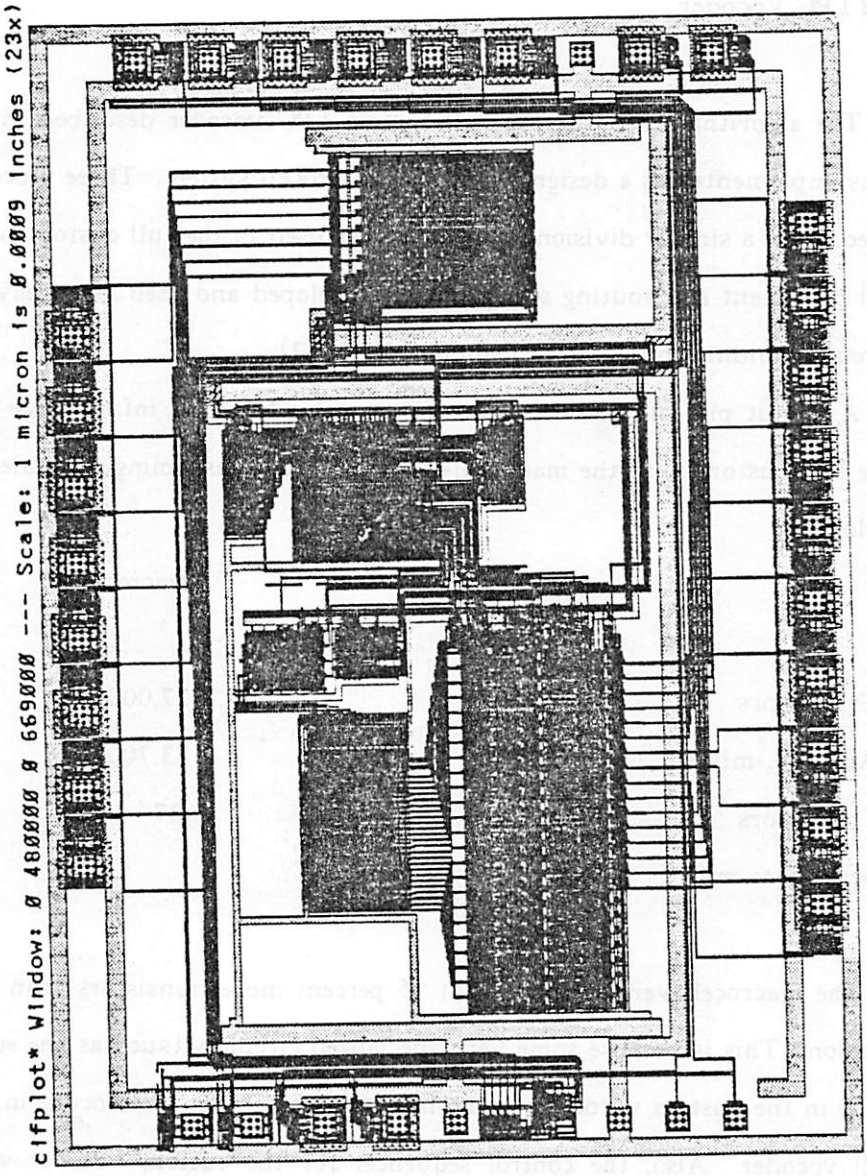


Fig. 5.3 Second Version of the Digital Audio Equalizer

5.2 LPC Vocoder

The algorithm used for the full-custom LPC vocoder described in Chapter 2 was implemented as a design file for the macrocell system. Three processors were used, with a similar division of labor as was used in the full custom circuit. Global placement and routing software was developed and used in the layout of this circuit, avoiding any need for manual wiring [32].

A layout plot of this circuit is shown in Fig.5.4. It is informative to compare the full-custom with the macrocell-designed circuit, assuming equivalent 3u technologies:

	full custom	macrocell
Transistors	23,000	27,000
Area (sq. mils)	33,200	73,700
Transistors per square mil	.69	.37

The macrocell version uses about 15 percent more transistors than the custom version. This is because some very specialized circuitry (such as the squaring circuit) in the custom vocoder was replaced by less-efficient microcode in the macrocell vocoder. Also, the control sequencer for the custom vocoder was smaller since the sequencers for the three processors shared substantial amounts of circuitry.

Three factors contribute to the lower density of transistors per unit area in the macrocell designed circuit. These are:

(1) The macrocells are less dense than the blocks of the full custom circuit.

(2) More area is consumed by wiring in the macrocell version.

(3) Suboptimal placement results in more empty areas in the macrocell version.

Factor (1) is a direct result of attempts to make the macrocells, particularly those performing I/O functions, very general and facilitative to automatic generation. Large numbers of timing strobes are needed to control these I/O functions, resulting in read-only memories with fairly low transistor densities in their OR-planes. One the other hand, arithmetic, addresing and data memory circuits are of essentially the same density as the full-custom approach.

Factors (2) and (3) -- the fact that placement and routing of the macrocells is less than optimal -- could certainly be mitigated by additional development effort. Only a few man months of software engineering went into this aspect of the project, in contrast to the many man-years of effort that an industrial placement and routing system would typically involve. But ultimately, it can be predicted that hand-wiring will always yield a denser circuit.

The design file for the LPC vocoder can be found in Appendix D.

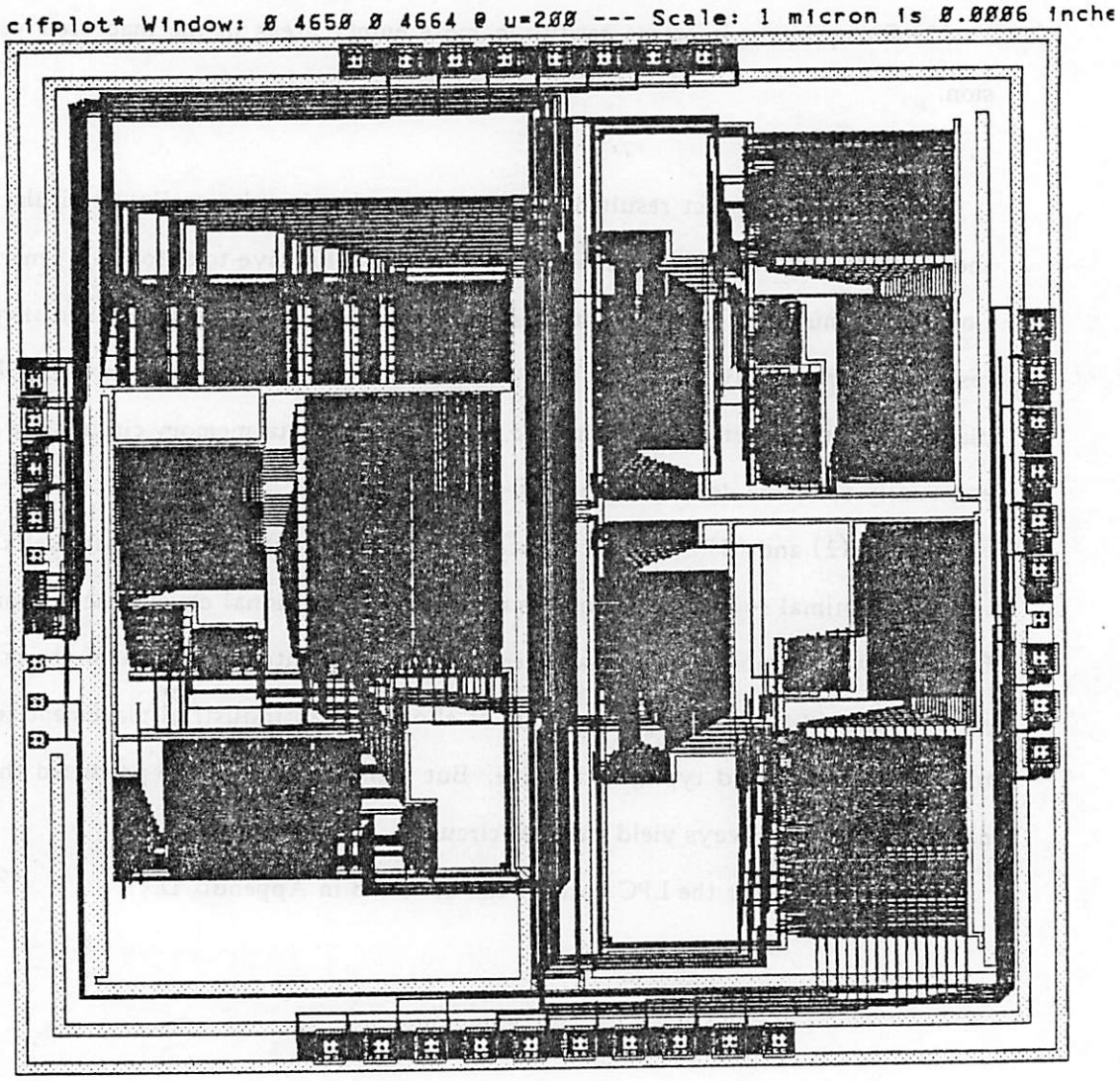


Fig. 5.4 LPC Vocoder

5.3 Decision Feedback Equalizer

The compiler was used to design a two-processor circuit intended for use as a equalizer for local telephone lines [33]. The sample rate is higher than the applications already discussed, on the order of 120 kiloHertz. Also, the system makes use of quite a bit of non-linear processing and decision-making. The user-specifiable FSM construct was found to be highly useful in this application.

One processor implements a transversal-filter adaptive filter whose inputs are a binary decision, represented by the values plus or minus one. The filter uses a stochastic gradient algorithm. The filter's output is an estimate of the inter-symbol interference observed for modulated data on the line. This estimated inter-symbol interference can then be subtracted from the received signal to created an equalized signal.

The second processor is used to recover timing information from the received data. The output of the second processor is used to control an off-chip voltage-controlled oscillator (VCO) which provides system timing, including the clock input to the IC.

A signal flow chart for the circuit is given in Fig. 5.5. A circuit plot is shown in Fig.5.6.

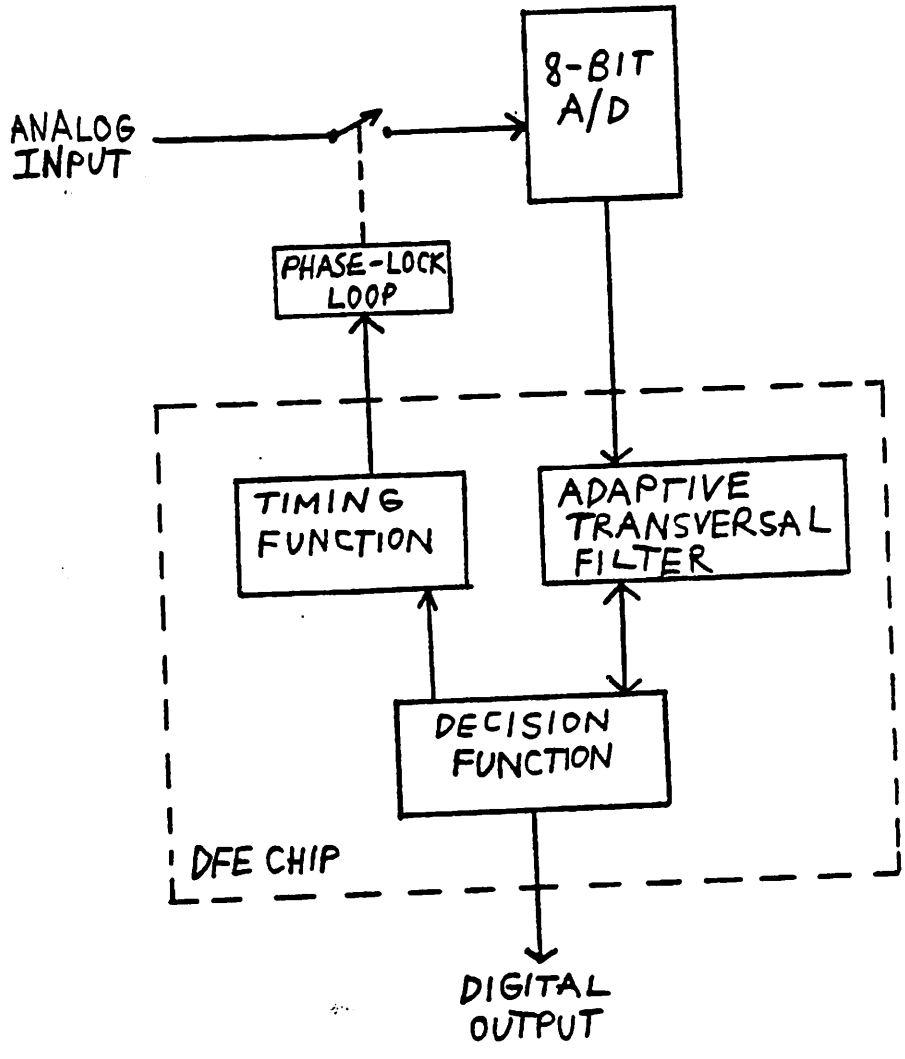


Fig. 5.5 Signal Flow Chart for Decision Feedback Equalizer

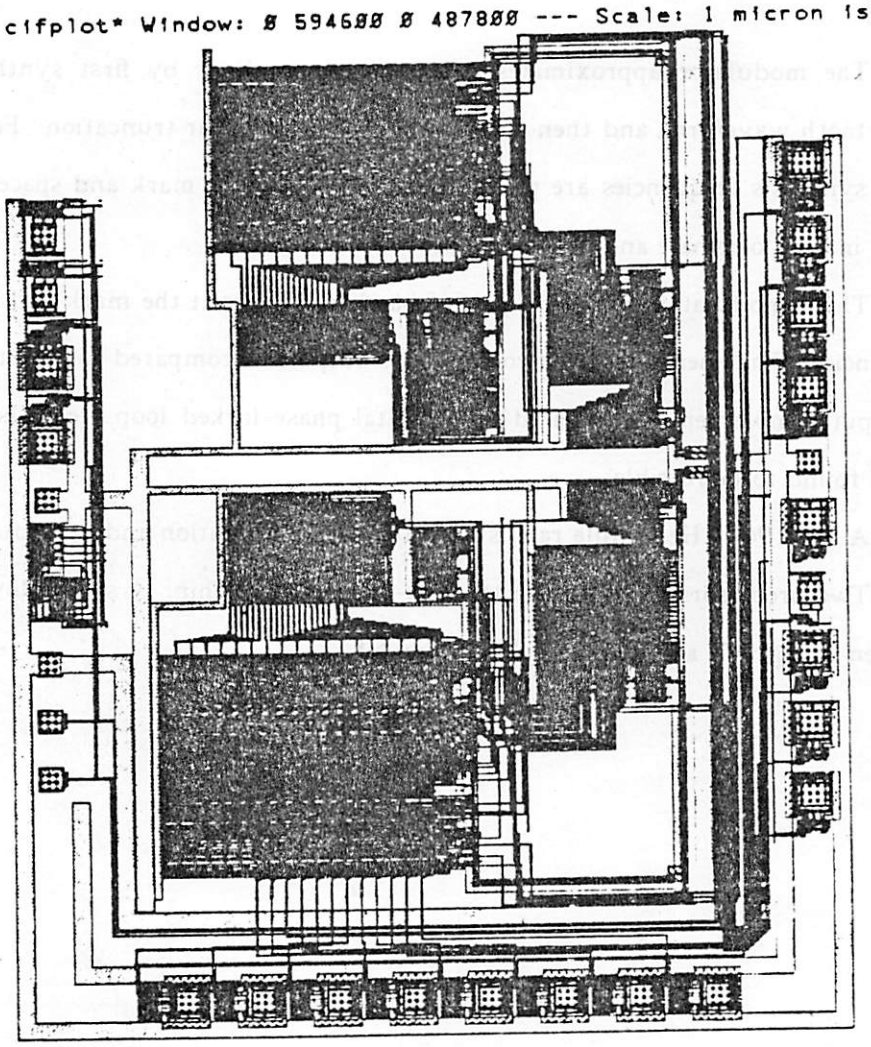


Fig. 5.6 Decision Feedback Equalizer

5.4 300-Baud Modem

A 300-Baud full duplex modem has been designed using the silicon compiler [36].

The modulator approximates a sine-wave synthesis by first synthesizing a sawtooth waveform, and then performing piecewise-linear truncation. Four possible synthesis frequencies are provided, corresponding to mark and space frequencies in both originate and answer modes.

The demodulator employs a pair of bandpass filters at the mark and space frequencies. The energy of the two bandpass outputs is compared to give the digital output. Implementations based on a digital phase-locked loop were also studied and found to be feasible.

A fixed 9600 Hz sample rate is used for both modulation and demodulation.

Two processors were used in the 300-baud modem chip. A signal flow chart is given in Fig. 5.7, and a circuit plot in Fig. 5.8.

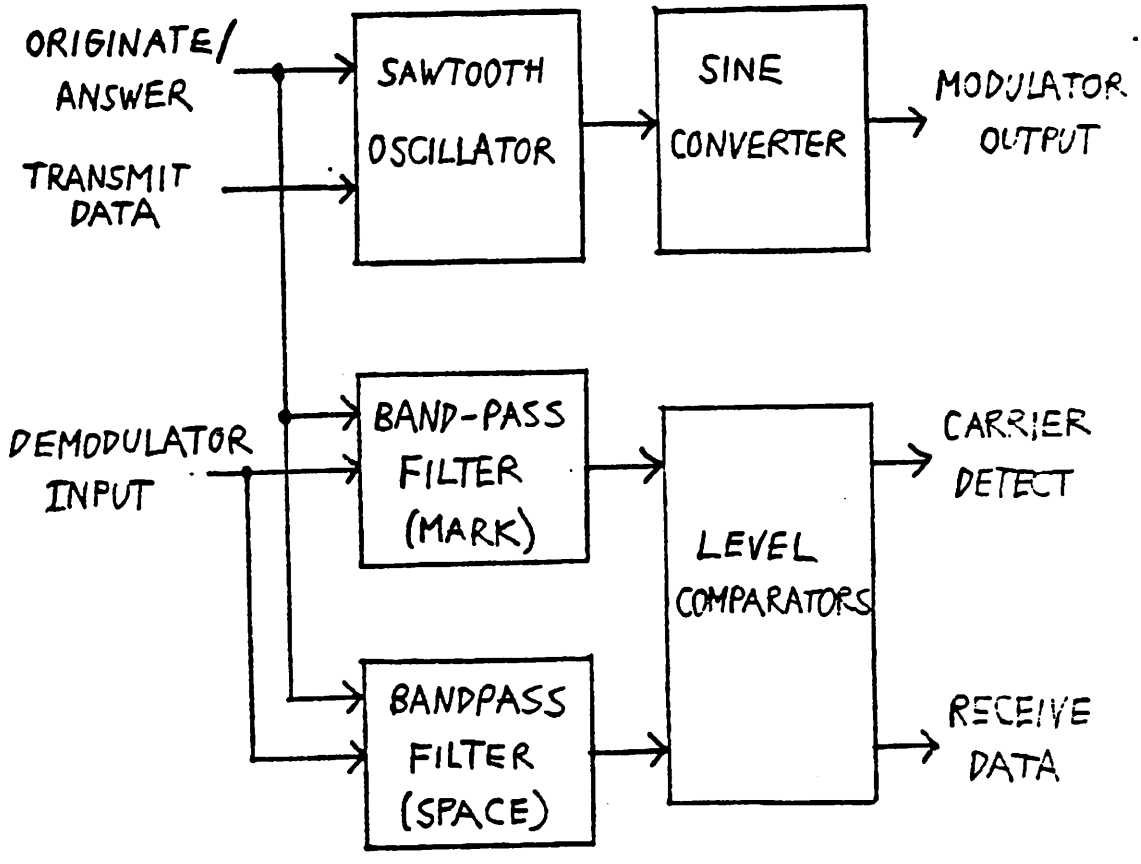


Fig. 5.7 Signal Flow Chart for 300 Baud Modem

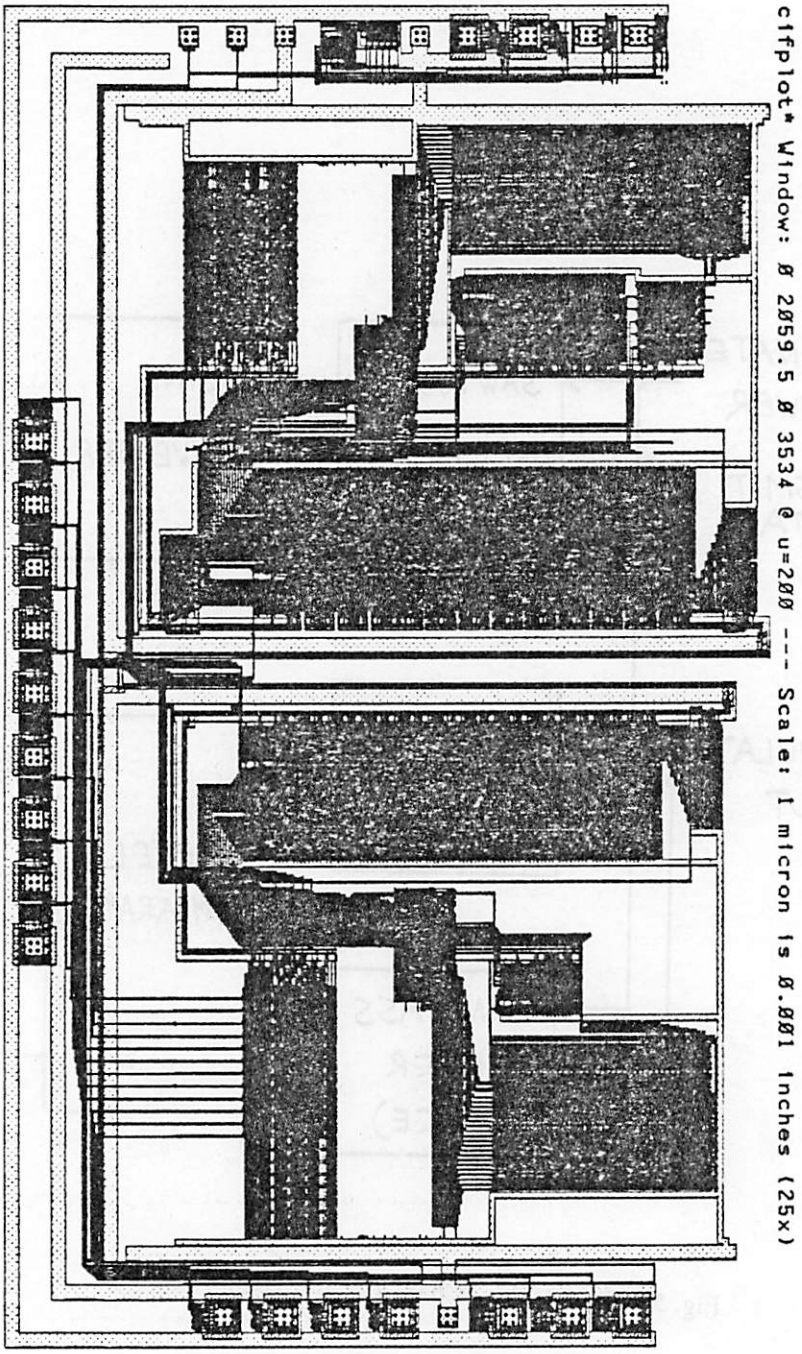


Fig. 5.8 300-Baud Modem

5.5 Conclusions

In Section 1.4, Chapter 1, the differences between procedural and graphical methods of designing an integrated circuit were discussed. In the procedural approach, a language is designed in which programs may be written. These programs are then translated into a circuit layout. In the graphical approach, the designer creates the circuit directly at the level of layout geometries.

The LPC vocoder IC described in Chapter 2 was designed with purely graphical methods. This approach allowed a very large degree of freedom and resulted in a very efficient design. However, the amount of design effort was also very large. It became clear that one could not routinely design circuits of this complexity without adopting higher-level design methods.

Much early work on procedural design was oriented towards the generation of low-level circuit elements such as data paths [34]. Although there are applications for custom data-path generation, this capability is of only limited use when the application range is sufficiently narrow. In the case of digital signal processors, a single data path architecture may serve many applications. By hand-crafting this data path the available silicon area is efficiently used.

Procedural design was used to advantage in the macrocell design system by parameterizing the data path (rather than customizing its architecture) and by configuring the I/O, multiprocessor and communications aspects of the IC in a procedural fashion. Graphical design was used to advantage in the creation of the cell library itself. This combined approach resulted in a powerful and useful design system.

There are a number of drawbacks to the system as currently implemented. The two most severe drawbacks are probably the restricted control structure, and the fixed-floorplan approach to processor assembly.

The control structure requires definition of a single sample rate for all processors on an IC. Each processor repeats its microprogram every sample interval. Thus applications with many different sample rates, or those which are best represented by programs whose execution spans many sample intervals, do not map well on to this architecture.

The fixed-floorplan method of assembling processors does not always result in an efficient layout. Experience to date suggests that in about two thirds of the cases, the processor assembly is reasonably area-efficient. In the other cases, however, size mismatch among the macrocells yields a processor assembly with a significant amount of wasted space. In these instances a different floorplan would have provided a more efficient layout. One proposal is to have a small number of alternative floorplans for the processor assembly, and to allow the compiler to select the most compact. A second proposal is to allow some degree of human intervention in the compilation process, allowing the designer to manipulate floorplans in a symbolic fashion.

Despite these two disadvantages, and several others of more minor consequence, the macrocell design system remains useful and valuable. Also, neither disadvantage is inherent in the macrocell approach. The macrocell system described in this dissertation remains both useful in its own right and a model for future efforts.

Appendix A — LPC Vocoder circuit microcode

A.1 Instruction Set

This appendix describes the microcode used in the three processors of the LPC vocoder circuit described in Chapter 2. (See Appendix D for equivalent information on the LPC vocoder of Section 5.2 of Chapter 5.)

The microprograms consist of a line of microcode per clock cycle. There are several symbolic microinstructions per line. All microinstructions in a given line operate concurrently. The ":" character begins a new line of microcode. Separate programs are given for the main program and subprogram of each processor.

Memory Instructions:

All memory instructions are followed by a field which is the symbolic address. Except for these address fields, all fields are instruction fields. Indexed instructions involve adding a index counter to the address offset to obtain the effective address. For Processor 1, a mod-10 counter which counts subprogram iterations is used for indexing. For Processor 2, all memory accesses are indexed by the same mod-10 counter. For Processor No. 3, two mod-6 index counters, X and Y, are available. The X counter counts the subprogram iterations, whereas the Y counter counts samples. In addition, Processor No. 3 allows conditional write instructions, whereby the write cycle is controlled by the FSM state variable "CC".

R : read from data memory into mor

RX : indexed read (Processors 1 and 3 only)

W : write from mir into data memory

WX : indexed write (Processors 1 and 3 only)

WC : conditional write (Processor 3 only)

WCX : indexed conditional write (Processor 3 only)

RY : indexed read from Y counter (Processor 3)

WY : indexed write from Y counter (Processor 3)

(transparent) : enable write data without cycling RAM (Processor 2)

Memory input register instruction:

writelatch : store mbus into mir (Processors 1 and 2 only)

This instruction causes the mir latch to be strobed. In Processor No. 3, the mir latch is always held transparent (strobe always enabled) so this instruction is not used.

Shift Register instructions:

These instructions indicate the value to be loaded into the shift register (sr) in terms of the other registers in the arithmetic unit.

sr:=sr/2 : shift right

sr:=mem/2 : load for mor

sr:=-mem/2 : load from mor, inverted

sr:=|mem/2| : load form mor, absolute value

Special shift register instructions. Processor No. 1:

These instructions complement the data with the sign bit of one of the coefficients ka or ks. The coefficient ka is computed by Processor No. 2 for each of the ten lattice analyzer stages. The coefficients ks are inputted from off-chip to program the synthesis filter.

sr:=-ka*mem/2

sr:=ks*mem/2

sr:=-ks*mem/2

Accumulator instructions:

These instructions specify the value to be loaded into the accumulator in terms of the various data path registers.

acc:=0 : clear accumulator

acc:=acc : hold accumulator

acc:=mem : load from mor

acc:=sr : load from sr

acc:=sr+acc : add sr to accumulator

acc:=sr+mem : add mor to accumulator

aip : accumulate if positive (Processor No. 2)

Special accumulator instructions. Processor No. 1:

These instructions gate the data with a magnitude bit of one of the coefficients ka or ks.

acc:=ka*sr+acc

acc:=ka*sr+mem

acc:=ks*sr+acc

acc:=ks*sr+mem

No operation:

NOP

Subprogram call and return, Processor No. 3:

jsr : jump to subprogram

ret : return from subprogram

These instructions are used to pass control from the main program to the subprogram and back. Due to pipelining these instructions act on a one-cycle-delayed basis.

Conditional logic instructions, Processor No. 3:

These instructions modify the state of the FSM which is attached to Processor No. 3.

SET : set condition code if positive accumulator

AND- : and condition code with negative accumulator

APV : and condition code if at peak or valley

VPE : valid pitch estimate (Y = 5)

SIP : set if peak

SIV : set if valley

SSL : set slope and last-slope flags

I/O instructions in general:

Input instructions serve to disable the accumulator from the mbus and enable an input signal. Output instructions serve to latch the value of the accumulator into an on-chip or off-chip register.

I/O instructions for Processor No. 1:

in : read from parallel bus

out1 : serial output to squaring circuit

out2 : serial output to Processor No. 3

xmit_exc : input from excitation source

SDBdisable : disable data bus for input operation

Bogus instructions used for emulation:

These instructions were included in the source code but do not affect the assembled object code.

(res)

(input)

(output)

(exc)

(ks_msb)

(plus)

(ka_msb)

(minus)

(quotient)

(plus_input)

(minus_input)

(energy)

I/O and timing signals for Processor No. 2:

mem:=in : input from squaring circuit to mir

ldsqr : timing signal for squaring circuit

strobedac : output strobe for Processor No. 1

stroberes: output strobe for processor No. 1

fifo1 : timing signal for FIFO buffer

fifo2 : timing signal for FIFO buffer

fifo3 : timing signal for FIFO buffer

egc1 : timing signal for excitation source

egc2 : timing signal for excitation source

LSK : load ks into P-S converter

I/O instructions for Processor No. 3:

mem:=in : input from Processor No. 1

out:=acc : output pitch estimate

A.2 Processor No. 1 Microcode

```
;                Filter main program
;
;
; Finish up last multiply of last subprogram from previous sample.
; Put residual in accumulator.


: sr:=sr/2 acc:=ks*sr+acc
: sr:=sr/2 acc:=ks*sr+acc
: R last_A acc:=ks*sr+acc writelatch
: W D(1) acc:=mem

;
; Copy C to D(0).  Demphasis filter.  Write speech input to E.
; Output residual, speech to SDB.  Leave speech output in writelatch.

: R last_C     (res)                    ; Residual is on SDB.
: R II sr:=mem/2 acc:=mem writelatch
: W D(0) sr:=mem/2 acc:=sr
: acc:=sr+acc sr:=sr/2 writelatch in SDBdisable
: W E acc:=sr+acc sr:=sr/2 in   (input)
: acc:=sr+acc writelatch
: (output)                          ; Speech output is on SDB.

;
; Preemphasis filter, with output going to A and next_B(0).  Store
; writelatch at next_H following final read from lasr_E (same address).
;

: R E
: R last_E acc:=mem
: R last_E acc:=acc sr:=-mem/2
: R last_E acc:=sr+acc sr:=-mem/2
: W next_H sr:=-mem/2 acc:=sr+acc
: R E sr:=sr/2 acc:=sr+acc
: R E sr:=mem/2 acc:=sr+acc
: sr:=mem/2 acc:=sr+acc
: acc:=sr+acc writelatch
: W A
: W next_B(0)

; Lowpass filter first stage.  Leave filter output in accumulator.
; Store excitation at C.

: R F
: sr:=-mem/2 acc:=mem
: R E sr:=sr/2 acc:=acc
```

```
: sr:=mem/2 acc:=sr+acc in SDBdisable writelatch xmit_exc
: W C sr:=sr/2 acc:=acc in  (exc)

: Lowpass filter second stage.  Lowpass output goes to out2.

: R G acc:=sr+acc writelatch
: W next_F sr:=mem/2
: acc:=sr+mem sr:=sr/2
: acc:=sr+acc writelatch
: W next_G out2

.end

;                    Filter Subprogram
;
;


;
: Finish final multiply/accumulate of previous subprogram iteration.
; putting result in writelatch.  Start to form sum (A + B)/2.

: sr:=sr/2 acc:=ks*sr+acc
: R A sr:=sr/2 acc:=ks*sr+acc
: RX B(i-1) sr:=mem/2 acc:=ks*sr+acc writelatch

;
: Load out1 with (A+B). Store writelatch at next_D(12-i).  Do first
: multiply accumulate  operation (C + ks*D(10-i)), putting result in
: writelatch.  Start second multiply/accumulate (A - ka*B(i-1)).

: RX D(10-i) sr:=mem/2 acc:=sr
: R C sr:=ks*mem/2 acc:=sr+acc   (ks_msb)
: WX next_D(12-i) sr:=sr/2 acc:=ks*sr+mem out1   (plus)
: sr:=sr/2 acc:=ks*sr+acc
: sr:=sr/2 acc:=ks*sr+acc
: sr:=sr/2 acc:=ks*sr+acc
: sr:=sr/2 acc:=ks*sr+acc
: RX B(i-1) sr:=sr/2 acc:=ks*sr+acc
: R A sr:=-ka*mem/2 acc:=ks*sr+acc writelatch  (ka_msb)

: Store result of first multiply/accumulate at C.  Finish second
: multiply/accumulate. putting result in writelatch.  Start
: to form sum (A - B(i-1))/2.

: W C sr:=sr/2 acc:=ka*sr+mem
: sr:=sr/2 acc:=ka*sr+acc
: sr:=sr/2 acc:=ka*sr+acc
: sr:=sr/2 acc:=ka*sr+acc
: sr:=sr/2 acc:=ka*sr+acc
```

```
: R A sr:=sr/2 acc:=ka*sr+acc
: RX B(i-1) sr:=mem/2 acc:=ka*sr+acc writelatch

; Store result of second multiply/accumulate at A.  Load out1 with
; (A - B(i-1)/2). Do third multiply/accumulate (A - ka*B(i-1)),
; putting result in writelatch.

: R A sr:=-mem/2 acc:=sr
: RX B(i-1) sr:=-ka*mem/2 acc:=sr+acc  (ka_msb)
: sr:=sr/2 acc:=ka*sr+mem out1  (minus)
: W A sr:=sr/2 acc:=ka*sr+acc
: sr:=sr/2 acc:=ka*sr+acc
: sr:=sr/2 acc:=ka*sr+acc
: sr:=sr/2 acc:=ka*sr+acc
: R C sr:=sr/2 acc:=ka*sr+acc
: RX D(10-i) sr:=-ks*mem/2 acc:=ka*sr+acc writelatch   (ks_msb)

;
; Store result of third multiply/accumulate at next_B(i). Start
; final multiply accumulate (C - ks*D(10-i)).

: sr:=sr/2 acc:=ks*sr+mem
: WX next_B(i) sr:=sr/2 acc:=ks*sr+acc
: sr:=sr/2 acc:=ks*sr+acc
: sr:=sr/2 acc:=ks*sr+acc

.end
```

A.3 Processor No. 2 Microcode

The main program and subprogram of Peocessor No. 2 differ only in the assertion of various timing and I/O strobes.

```
;            Correlator Main Program
;
;
; Finish division operation from previous iteration.

: sr:=sr/2 acc:=sr+acc aip          ; quotient bit 4 available
: sr:=sr/2 acc:=sr+acc aip          ; quotient bit 3 available
: sr:=sr/2 acc:=sr+acc aip LSK ldsqr  ; quotient bit 2 available
: acc:=sr+acc aip   (quotient)      ; quotient bit 1 available
: fifo2 stroberes                   ; quotient bit 0 available
```

```
; Single pole lowpass filter first signal from input port, leave
; result in accumulator.

: R C
: sr:=-mem/2 acc:=mem
: sr:=sr/2 acc:=acc
: sr:=sr/2 acc:=acc
: sr:=sr/2 acc:=acc     egc1
: sr:=sr/2 acc:=acc fifo3 strobedac mem:=in writelatch
: (transparent) sr:=sr/2 acc:=acc mem:=in  (plus_input)
: sr:=mem/2 acc:=sr+acc
: R D acc:=sr+acc writelatch

; Store result of first filter at C.  Single pole lowpass filter
; second signal from input port, leave result in writelatch.

: W C sr:=-mem/2 acc:=mem
: sr:=sr/2 acc:=acc
: sr:=sr/2 acc:=acc
: sr:=sr/2 acc:=acc
: sr:=sr/2 acc:=acc mem:=in writelatch ldsqr
: (transparent) sr:=sr/2 mem:=in acc:=acc  (minus_input)
: R C sr:=mem/2 acc:=sr+acc
: R C sr:=-mem/2 acc:=sr+acc writelatch

; Store result of second filter at D.  Put (D-C)/2 in accumulator
; and (C+D)/4 in shift register.

: W D sr:=-mem/2 acc:=sr+acc
: R D acc:=sr+acc  writelatch      ; now acc = (D-C)
: R C sr:=mem/2
: (transparent) sr:=mem/2 acc:=sr LSK
: (transparent) sr:=-mem/2 acc:=sr+acc writelatch ; now acc = (C+D)/2
: (transparent) sr:=mem/2 acc:=sr  (energy)
: sr:=mem/2 acc:=sr aip

; Do divide operation with sequence of "accumulate-if-positive" cycles.

: sr:=sr/2 acc:=sr+acc aip        ; quotient bit 7 (sign) available
: sr:=sr/2 acc:=sr+acc aip        ; quotient bit 6 available
: sr:=sr/2 acc:=sr+acc aip        ; quotient bit 5 available  egc2

.end

;            Correlator Subprogram
;
;
; Finish division operation from previous iteration.
```

```
: sr:=sr/2 acc:=sr+acc aip          ; quotient bit 4 available
: sr:=sr/2 acc:=sr+acc aip          ; quotient bit 3 available
: sr:=sr/2 acc:=sr+acc aip LSK ldsqr  ; quotient bit 2 available
: acc:=sr+acc aip                   ; quotient bit 1 available
: fifo2

; Single pole lowpass filter first signal from input port. leave
; result in accumulator.

: R C
: sr:=-mem/2 acc:=mem
: sr:=sr/2 acc:=acc
: sr:=sr/2 acc:=acc
: sr:=sr/2 acc:=acc
: sr:=sr/2 acc:=acc fifo3 mem:=in writelatch
: (transparent) sr:=sr/2 acc:=acc mem:=in
: sr:=mem/2 acc:=sr+acc
: R D acc:=sr+acc writelatch

; Store result of first filter at C.  Single pole lowpass filter
; second signal from input port, leave result in writelatch.

: W C sr:=-mem/2 acc:=mem
: sr:=sr/2 acc:=acc
: sr:=sr/2 acc:=acc
: sr:=sr/2 acc:=acc
: sr:=sr/2 acc:=acc mem:=in writelatch ldsqr
: (transparent) sr:=sr/2 mem:=in acc:=acc
: R C sr:=mem/2 acc:=sr+acc
: R C sr:=-mem/2 acc:=sr+acc writelatch

; Store result of second filter at D.  Put (D-C)/2 in accumulator
; and (C+D)/4 in shift register.

: W D sr:=-mem/2 acc:=sr+acc
: R D acc:=sr+acc  writelatch       ; now acc = (D-C)
: R C sr:=mem/2
: (transparent) sr:=mem/2 acc:=sr LSK
: (transparent) sr:=-mem/2 acc:=sr+acc writelatch  ; now acc = (C+D)/2
: (transparent) sr:=mem/2 acc:=sr
: sr:=mem/2 acc:=sr aip

; Do divide operation with sequence of "accumulate-if-positive" cycles.

: sr:=sr/2 acc:=sr+acc aip          ; quotient bit 7 (sign) available
: sr:=sr/2 acc:=sr+acc aip          ; quotient bit 6 available
: sr:=sr/2 acc:=sr+acc aip          ; quotient bit 5 available

.end
```

A.4 Processor No. 3 Microcode

```
;          Pitchtracker Main Program
;

; Finish up scoring calculation from previous sample by comparing
; "score" to "topscore" and if greater, updating "topscore" and
; "winner".

: R score
: R topscore sr:=mem/2
: sr:=-mem/2 acc:=sr
: R score acc:=sr+acc
: RY pp acc:=mem   SET
: WC topscore acc:=mem
: WC winner

;
; If VPE (every six samples) compare "topscore" to constant "VOICED".
; set "pitch" to either "winner" or zero, and reset "topscore".

: R winner
: R VOICED acc:=mem sr:=mem/2 VPE
: WC pitch sr:=-mem/2
: R topscore sr:=sr/2 acc:=0
: WC topscore acc:=sr+mem
: R signal acc:=0 AND-        ; Take "signal" from last sample and
: WC pitch acc:=mem SIP       ; update "lp", "lv", and "ls" (last
: WC lp acc:=acc SIV          ; peak, last valley, last signal).
: WC lv acc:=acc
: W ls

;
; Beginning of new sample. Get new sample from input and store in
; "signal". Compare signal to last signal and set slope (SSL).
; Reset "score".

: R ls mem:=in
: W signal sr:=-mem/2 mem:=in
: sr:=-mem/2 acc:=sr
: acc:=sr+acc
: SSL acc:=0
: W score

;
; Send "pitch" to output port. Form "signal(1)" through "signal(5)"
```

```
; from signal, lp, lv.  Jump to subsprogram ("jsr").

: R signal
: R pitch acc:=mem
: W signal acc:=mem
: R lv acc:=mem  out:=acc
: W signal(1) sr:=-mem/2
: R signal sr:=mem/2 acc:=sr
: R lp acc:=sr+acc sr:=mem/2
: W signal(2) sr:=-mem/2 acc:=sr
: R signal(2) acc:=sr+acc
: W signal(4) acc:=mem
: W signal(2) acc:=mem
: W signal(5) acc:=mem jsr
: W signal(3)

; End of main program sequence.  Following word is addressed during
; last iteration of subprogram.  Next word is a left over, and following
; are filler.

: NOP ret
: NOP
: NOP
: NOP
: NOP
: NOP
: NOP
: NOP
: NOP
: NOP

.end

;              Pitchtracker Subprogram
;
;
; Increment pitch period counter by four and set condition code if
; greater than BLANK (blanking interval).  Conditionally decay
; threshold.  And condition code with peak/valley indicator.

: R FOUR
: RX ppc sr:=mem/2
: R BLANK acc:=sr+mem      ; Use spare memory cycles to refresh
: WX ppc sr:=-mem/2        ; thresh, lp, and lv.
: sr:=-mem/2 acc:=sr
: RX thresh acc:=sr+acc
: sr:=-mem/2 SET          R lp acc:=mem
: sr:=sr/2                WX thresh acc:=mem
: sr:=sr/2                W lp
```

```
: sr:=sr/2              R lv
: RX thresh sr:=sr/2      acc:=mem
: sr:=sr/2 acc:=mem       W lv
: RX signal sr:=sr/2 acc:=sr+acc
: RX signal sr:=-mem/2 acc:=sr+acc
: WCX thresh acc:=sr+acc sr:=mem/2 APV

;
; And condition code with result of (signal > threshold)
; comparison.  Conditionally update thresh. lpp. ppc. pp.

: RX pp acc:=sr AND-
: WCX thresh acc:=mem
: WCX lpp
: RX ppc acc:=0
: WCX ppc acc:=mem
: WCX pp

; Add contribution for this channel to score of current candidate:
; do three window comparisons with the current candidate and pp.
; lpp. and pp+lpp.  In each case increment score by four if true.

: RY pp
: RX pp sr:=-mem/2
: R WINDOW1 sr:=mem/2 acc:=sr
: R WINDOW2 sr:=mem/2 acc:=sr+acc
: R FOUR sr:=mem/2 acc:=sr+acc
: R score sr:=mem/2 acc:=sr+acc SET
: RY pp acc:=sr+mem  AND-
: WC score sr:=-mem/2
: RX lpp acc:=sr
: R WINDOW1 sr:=mem/2 acc:=acc
: R WINDOW2 sr:=mem/2 acc:=sr+acc
: R FOUR sr:=mem/2 acc:=sr+acc
: R score sr:=mem/2 acc:=sr+acc SET
: RY pp acc:=sr+mem  AND-
: WC score sr:=-mem/2
: RX lpp acc:=sr
: RX pp sr:=mem/2 acc:=acc
: R WINDOW1 sr:=mem/2 acc:=sr+acc
: R WINDOW2 sr:=mem/2 acc:=sr+acc
: R FOUR sr:=mem/2 acc:=sr+acc
: R score sr:=mem/2 acc:=sr+acc SET
: RY pp acc:=sr+mem AND- jsr
: WC score sr:=mem/2

.end
```

Appendix B – NMOS Cell Library Documentation

B.1 Introduction

The purpose of this appendix is twofold. First, it describes the NMOS cell library at the level of mixed-mode (gate and transistor) schematics. Second, it describes how the cells are to be assembled into macrocells.

It is often the case that two cells which are adjacent to one another in part of an assembled macrocell will connect electrically at one or more points. This is indicated by labeling the schematics of the two cells with signal names, and using the same signal name for points which are to be connected.

Cells which are located around the periphery of a macrocell may have terminals on their outside edges. The existence and location of such terminals can be determined by examining the descriptor file, Section B.12.

Most schematics are drawn such that the orientation of the cells and location of any signals corresponds to the cell layout itself. Thus, for example, a control signal which busses horizontally through a cell are drawn that way in the schematic.

Schematics are given only for cells which contain active circuitry. Cells without active circuitry generally perform interconnect functions. Mention is made in the text of any instance in which a cell without active circuitry performs an interconnect other than GND, Vdd, phi1 or phi2.

The macrocell assemblies are described in a fashion corresponding to the way in which they are tiled together by the compiler. Thus, a macrocell is a stack of rows of cells.

Only a small number of cells are found in more than one macrocell. The same counter bit-slice is used in the PC, AAU and SPC macrocells. The adder cell used

in the AAU macrocell is identical to the adder used in the AUIO macrocell, although the two adders are in different cells. The FSM and ROM macrocells use nearly the same set of cells.

This appendix presumes familiarity with the contents of Section 3.8 of Chapter 3.

B.2 Program Counter and Subprogram Counter

The cell *counter* is the basic counter bit-slice. This is a synchrounous counter with "count" and "load" control signals, as well as clocks, bussing through the cell horizontally. The variants *counter.0* and *counter.1* have their data input hard-wired to ground or Vdd respectively.

The PC macrocell has a control cell, *pc.ctl*, that sits to the right of the bit slice array. *counter.0* cells are used in the array, and the control cell clears the PC when the "reset" signal is asserted. Other than this the PC just increments. The "reset" signal is delayed one clock signal to create "EOS", or end-of-sample.

The ouputs of the bit-slice counter are buffered by complementary drivers in the cells *pc.1*. The drivers provided inverted and non-inverted addresses for the lower half of the ROM and-plane.

Schematics of the cells *counter*, *pc.ctl*, and *pc.1* are given in Figs. B.1 and B.2.

The organization of cells within the PC macrocell is given in Fig. B.3.

The SPC also has a control cell, *spc.ctl*, and a complementary driver *spc.1*.

Sequencing the SPC is a little more complex. The cell *counter* is used in the least-significant bit slice, and the cell counter.0 is used in the remaining slices. The logic in the cells *spc.4*, *spc.5*, *spc.6* and *spc.ctl* (Fig. B.4) allows loading a zero into the LSB during the main program, and a one at the beginning of a subprogram. During the subprogram, the SPC just increments.

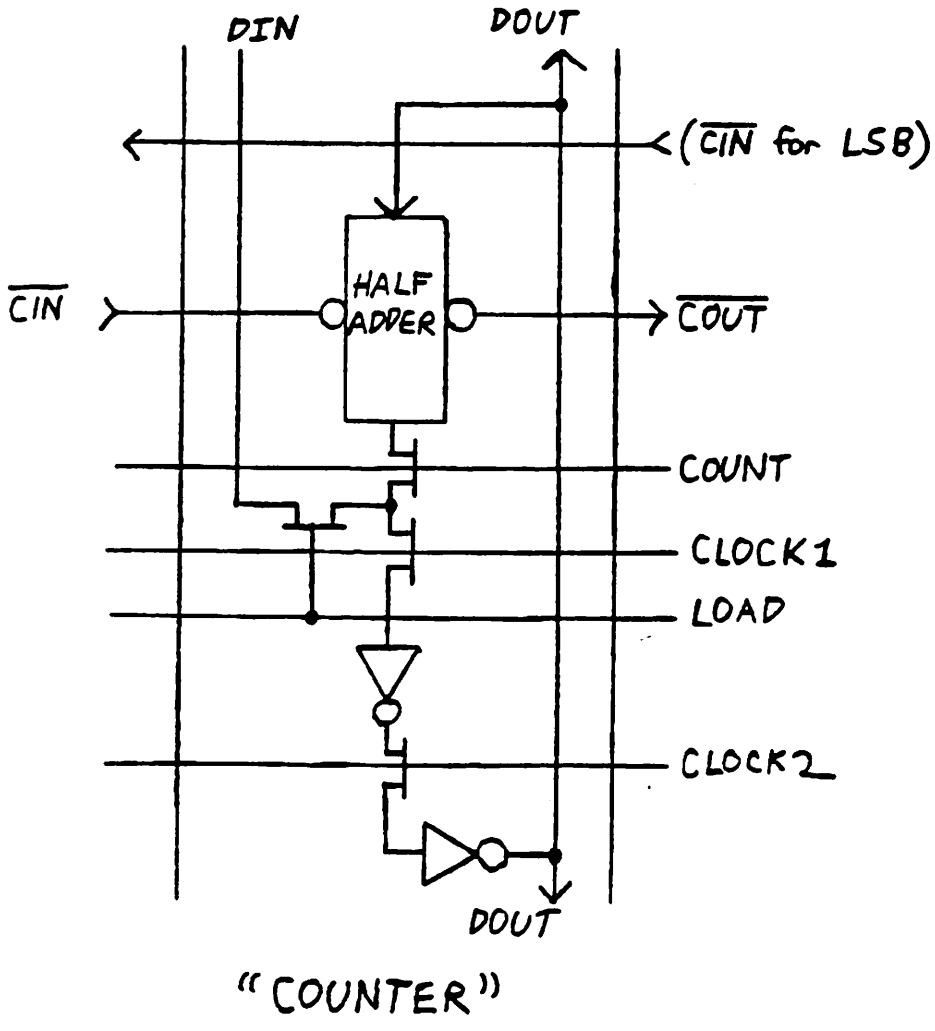
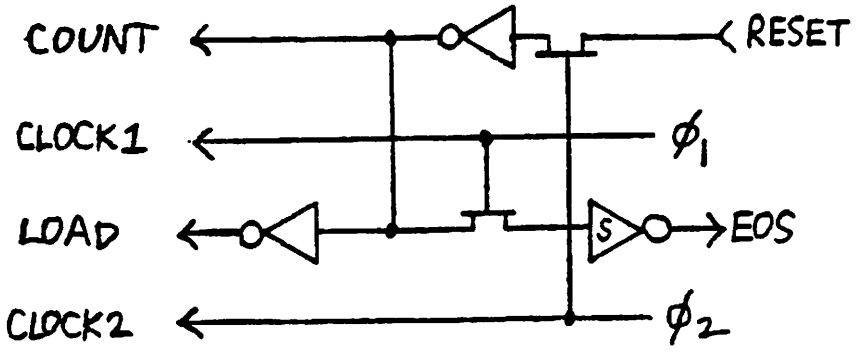
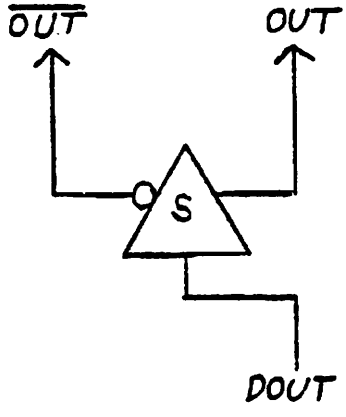


Fig.B.1 Counter bit-slice schematic



«PC.CTL»



«PC.1»

Fig.B.2 Schematics of cells used in PC macrocell

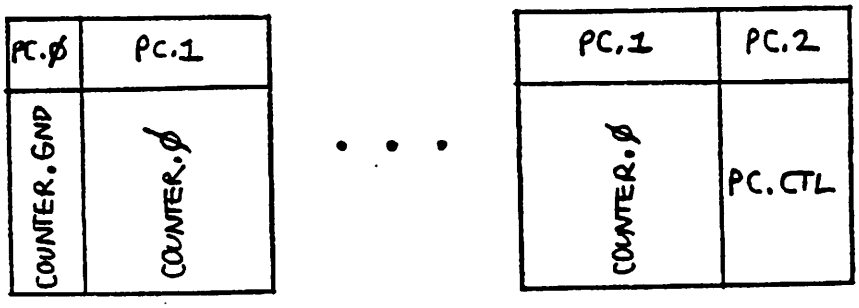


Fig.B.3 Organization of the PC macrocell

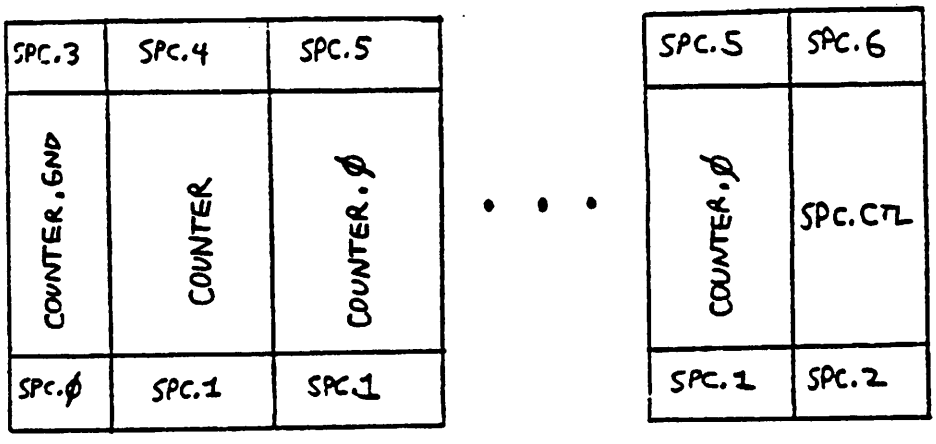


Fig.B.5 Organization of the SPC macrocell

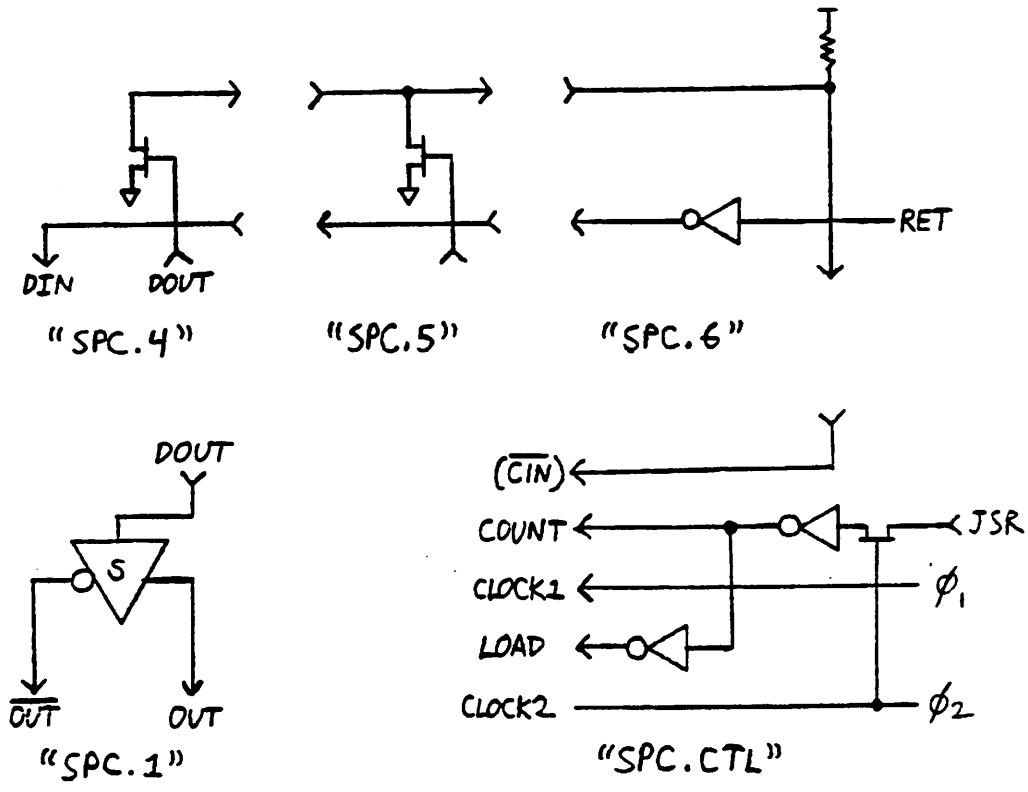


Fig.B.4 Schematics of cells used in SPC macrocell

The organization of cells within the SPC macrocell is given in Fig. B.5.

The cell *counter.gnd* contains no active circuitry, but it does connect the carry-in of the counter to a bus that goes back across the slices to the control cell.

B.3 The Address Arithmetic Unit

The AAU may contain either an IX counter, an IY counter, or both counters. Fig.B.6 is the schematic of a single bit slice for an AAU which has both counters. The cell names (except for the "decode" and "ydecode" sections, described below) are given along the left side of the schematic.

The IX counter must be -1 during the main program, 0 during the first iteration of the subprogram and so forth. So usually *counter.1* cells are used for the IX counter, so that it loads a -1 at the end of the sample. However, in the unusual case where there is no main program, *counter.0* cells are used here instead so that a zero is loaded. This begins the subprogram immediately.

The *aauinvert* cell provides true and complement outputs of the IX counter. This has two purposes: (1) The "decode" section can decode the value of IX to generate a "test" output to the FSM macrocell; (2) The inverted output is feeds an inverting input of the *aauadd* cell.

The decode section is empty unless a "IX<k>" or "IX[k]" reference is made in the FSM definition. For each such reference a row is included in the decode section, which involves placing a zero, one, or don't care cell in each bit-slice of the AAU. In this way an arbitrary function of the value of IX is created. Schematics for these cells are given in Fig. B.7.

The inverted output of IX feeds an "aauadd.even" or "aauadd.odd" cell. The adders themselves are identical circuits to the ones given in Fig.2.9 in Chapter 2. A nor-gate was included to control the indexing of the AAU input with the IX

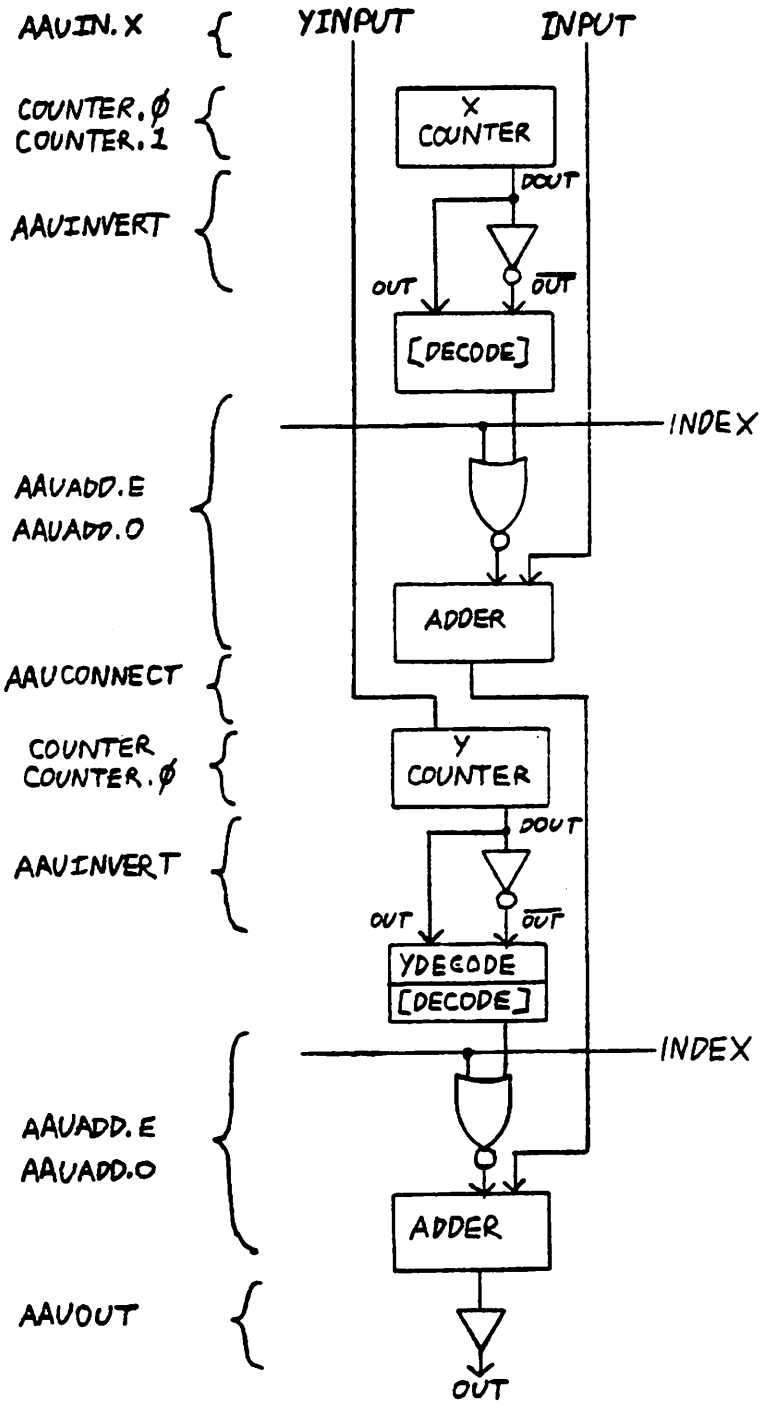


Fig.B.6 AAU bit-slice

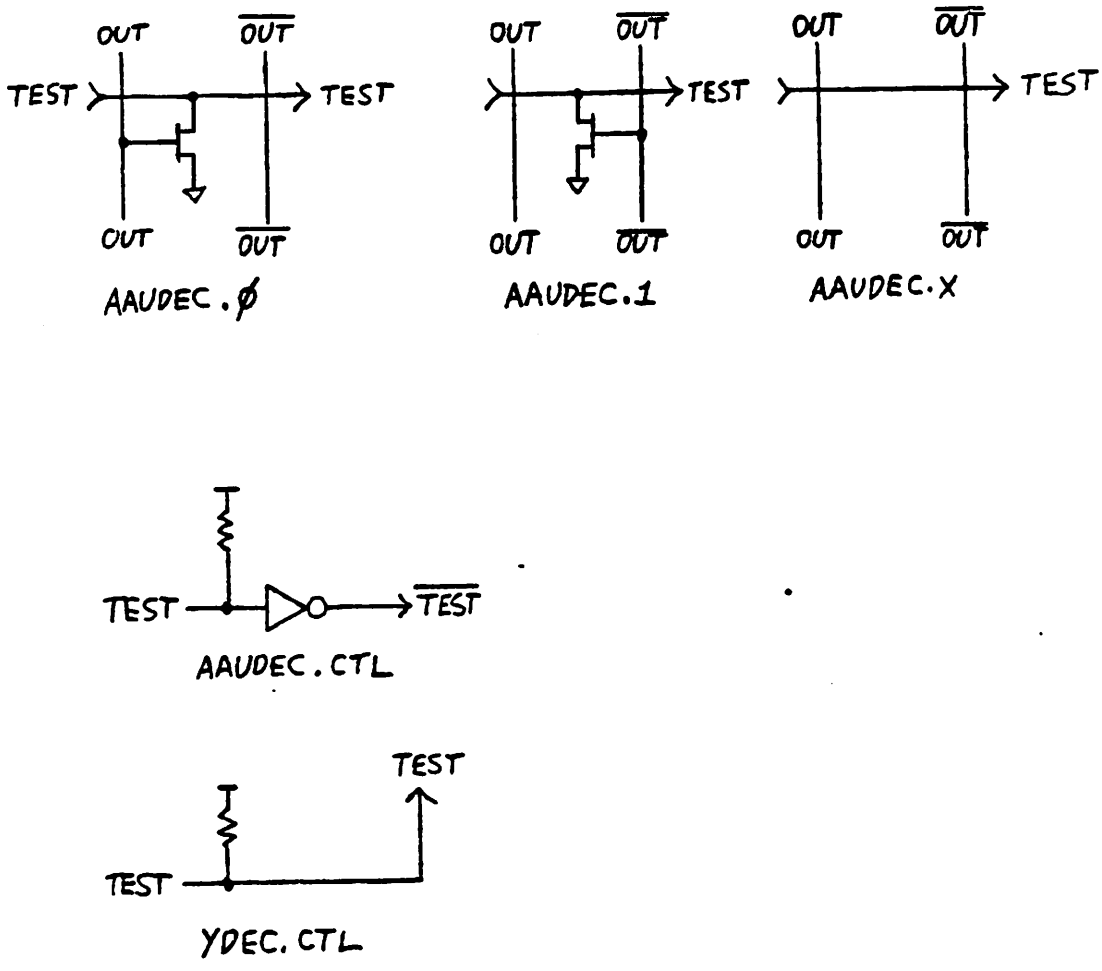
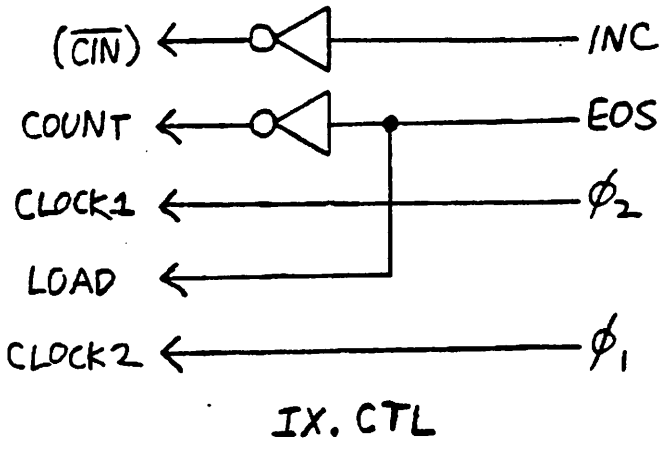
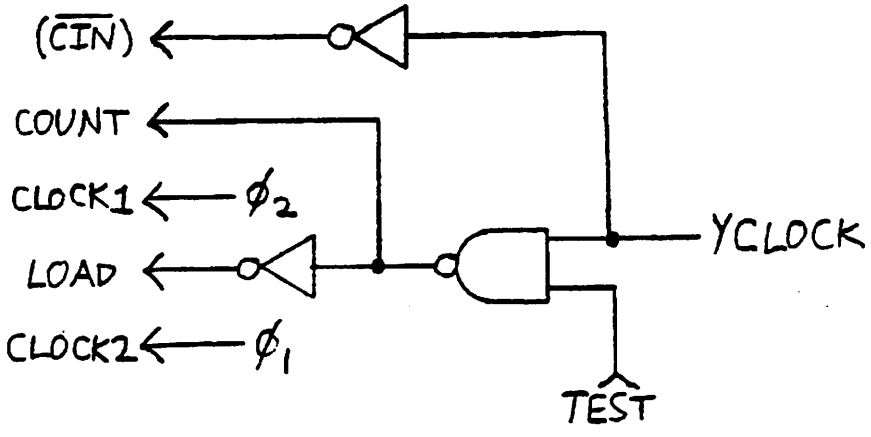


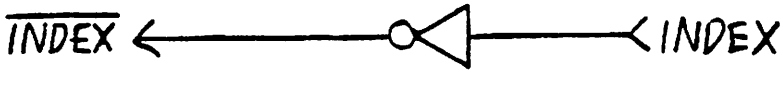
Fig.B.7 Scematics of cells used in AAU decode section



IX.CTL



IY.CTL



AAUADD.CTL

Fig.B.8 Schematics of AAU control cells

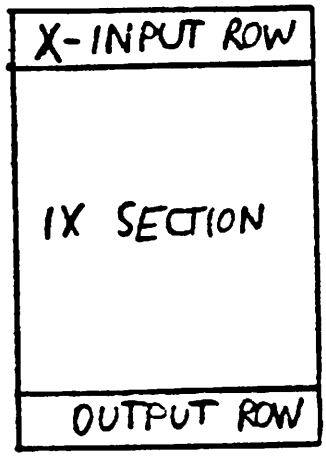


Fig.B.9 Organization of the AAU macrocell (IX indexing only)

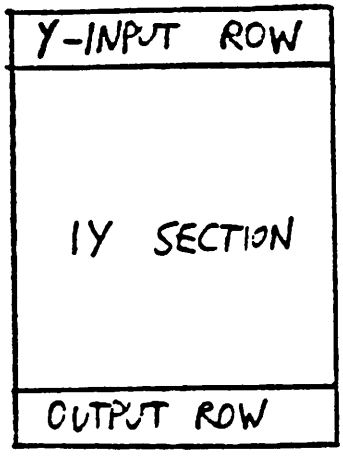


Fig.B.10 Organization of the AAU macrocell (IY indexing only)

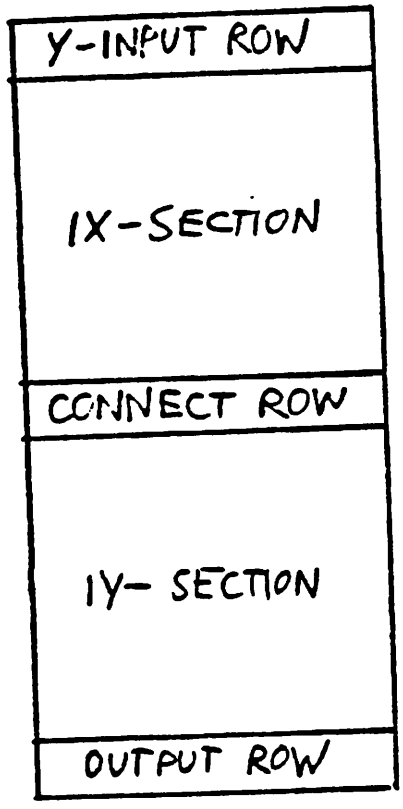


Fig.B.11 Organization of the AAU macrocell (IX and IY indexing)

counter.

The IY counter and its decode section and adder are very similar to the IX, but the sequencing is different. A "ydecode" section is included in both pointer and counter modes.

If the IY indexing is in counter mode, the "ydecode" section is programmed with the modulus of the counter and *counter.0* cells are used for the bit slices. The counter is clocked with the signal EOS. This has the effect of incrementing IY each sample until its maximum value is reached, then resetting it.

If pointer-mode addressing is used, the "ydecode" section is programmed with "don't cares" in every bit, and *counter* cells are used in the bit slices. Then whenever the IY counter is clocked, it is loaded from its inputs, which in this case come in externally to the AAU from the AUIO parallel bus. Note that in this mode, the IY counter is used as a register and never counts.

A "decode" section is included for the IY counter to provide "test" outputs for the FSM (as with the IX) and also to provide "IY=0" signals for the host interface if IY-indexed host I/O is being performed.

The *aauadd* cells following the IY counter index the address with the value of IY. Following this is an output buffer, *aauout*.

Schematics for the control cells for the IX and IY counters, and the *aauadd* cells are given in Fig.B.8.

Figs. B.9, B.10, and B.11 show the organization of AAU macrocells with IX indexing only, IY indexing only, and both types of indexing, respectively. The following describes in exact terms the cellular organization of the AAU macrocell. In every case the number of bit-slices in the AAU is equal to N, the number of address lines.

The "output row" always consists of the cells:

aauout.gnd

N instances of *aauout*

aauout.ctl

The "connect row", if included, is similarly composed of the *aauconnect.gnd*, *aauconnect*, *aauconnect.ctl* cells.

The y-input row consist of the cells:

aauin.gnd

aauin.y (N instances)

aauin.ctl

The "x-input row" consists of the cells:

aauin.gnd

aauin.x (N instances)

aauin.ctl

The "ix section" and "iy section" contain the following rows, listed from bottom to top:

ix section:

adder row

[decode]

invert row

counter row

iy section:

adder row

[decode]

ydecode

invert row

counter

The adder row consists of the cells:

aauadd.gnd

N instances of *aauadd.e or aauadd.o* (alternating)

aauadd.ctl

The "counter row" contains cells as follows:

counter.gnd

N instances of *counter*, *counter.0*, or *counter.1* (see below)

ix.ctl or *iy.ctl* (for ix section and iy section, respectively)

The "invert row" contains *aauinvert.gnd*, *aauinvert*, *aauinvert.ctl* cells in the usual fashion.

The [decode] section contains zero or more rows of the following form:

aaudec.gnd

N instances of *aaudec.0*, *aaudec.1*, *aaudec.x*

aaudec.ctl

The ydecode section contains one row of cells

aaudec.gnd

N instances of *aaudec.0*, *aaudec.1*, *aaudec.x*

ydec.ctl

B.4 Control ROM and Finite-State Machine.

The construction of the ROM and FSM are very similar. The ROM may have a split and-plane (as described in Section 2.3), while the FSM only has a single and-plane. The ROM has dual complementary inputs to its and-planes, while the FSM has registered single-ended inputs, and complementary drivers for the and-plane. In practice, some of the FSM outputs will be wired to some of the FSM inputs. However, this wiring is not part of the FSM macrocell itself.

Schematics for cells in the ROM/FSM and-plane are given in Fig.B.12. Complementary inputs enter the upper half of a ROM's split and-plane through the cell romtop.4. Complementary inputs enter the bottom half of a ROM's split and-plane, or a non-split and-plane, through the cell *rombtm.4*. Single-ended FSM inputs are registered and inverted by the cell *fsmdrv*.

The cells *pladec.0*, *pladec.1*, *pladec.x* program the and-plane. The complementary inputs bus vertically through these cells. The horizontal metal select lines, which are pulled up in pairs by the cell *romword.3*, exit the and-plane along its left edge and enter the or-plane.

The and-plane (and its upper and lower halves if split) must have an even

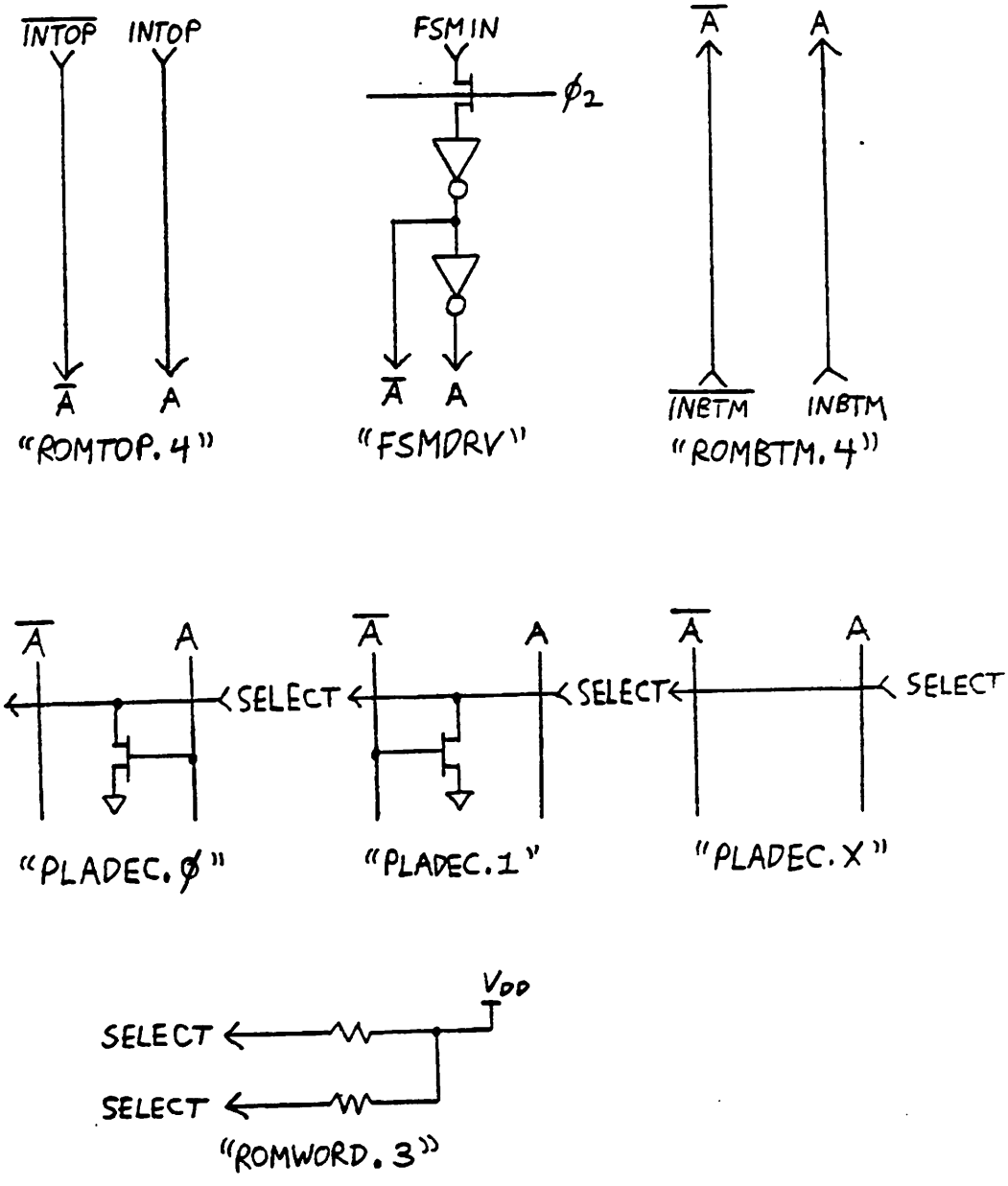
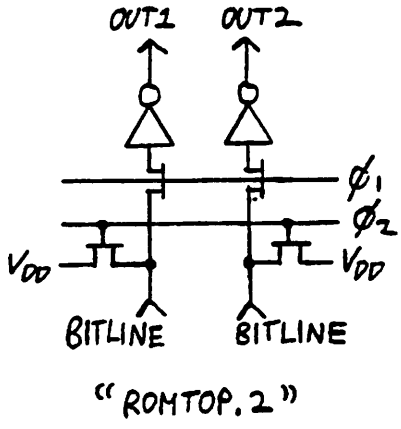
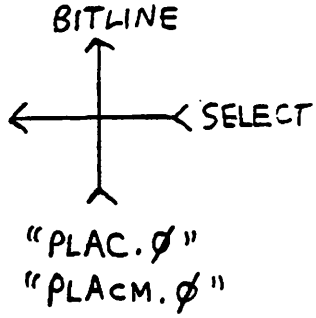
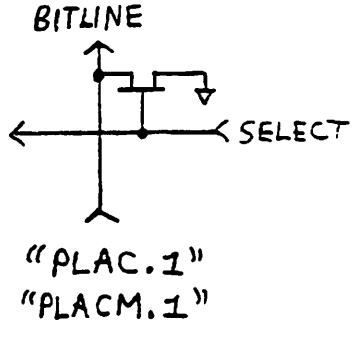


Fig.B.12 Schematics of cells used in ROM/FSM and-plane







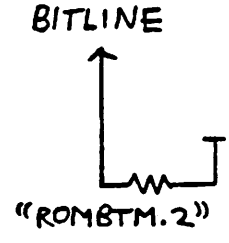


Fig.B.13 Schematics of cells used in ROM/FSM or-plane

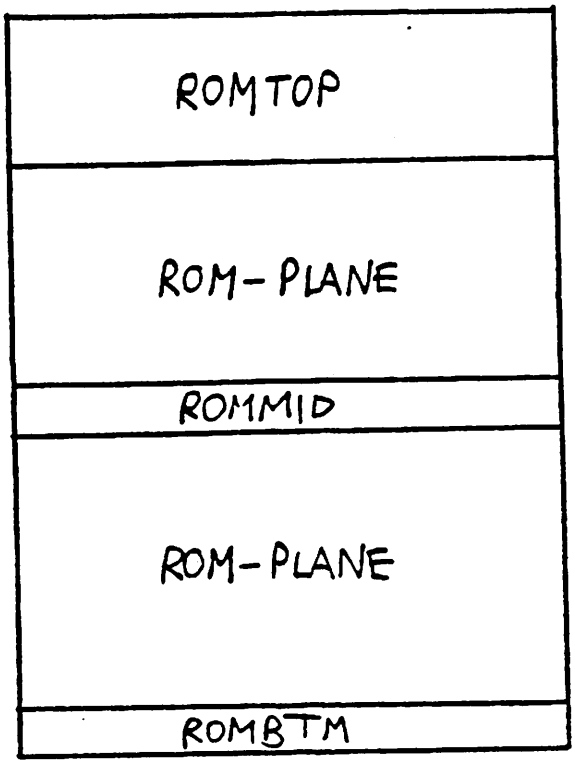


Fig.B.14 ROM organization

number of rows. Also, the SPC addressing the upper and-plane may have fewer bits than the PC addressing the lower and plane. If this is the case, the unused columns of the upper and plane must be filled with *pladec.x* cells.

The or-plane of the ROM and FSM macrocells is the only place in the cell library where any sort of mirroring is used. Because of this, there is no mirroring feature built into the tiling software. Instead, mirrored versions of or-plane cells are contained in the cell library, and the compiler first pass specifies the correct versions of the cells in the intermediate file.

Schematics of cells used in the ROM/FSM or-plane are given in Fig. B.13.

The cells *plac.0*, *plac.1* and their mirrored versions *placm.0*, *placm.1* program the or-plane. The select lines are polysilicon in the or-plane, and the metal bit-lines are pulled up in the cell *rombtm.2*. The bitlines are also precharged in the dual output register *romtop.2*. The use of large devices for precharging is needed for fast access times, and the use of pullups prevents precharge clock feedthrough from destroying the signal level if there are only a few words in the ROM (and the bitline capacitance is low).

The output registers were laid out in pairs because of the narrow bit-line pitch. There must be an even number of ROM outputs.

In both the and-plane and the or-plane of a ROM or FSM, diffusion (N+) wires are used to supply ground connections to the cells. These wires run vertically through the and-plane, and horizontally through the or-plane. However, it is necessary to connect these diffusion wires to metal ground lines at intervals, to prevent excessive voltage drop. Therefore, "dummy" rows are included which contain these ground lines.

In the and-plane, a "dummy" row is included for every sixteenth row. This requires a set of dummy cells *dummy.1*, *dummy.2*, *dummy.3*, *dummy.4*, *dummy.5*, through both planes.

In the or-plane, every 20th column of data is followed by a ground-line column, composed of ".g" cells: *rombtm.g*, *plac.g*, *placm.g*, *rommid.g*, *romtop.g*.

Where ground-line columns and dummy rows intersect, a *dummy.g* cell is placed.

The true and complement address lines bus through the *dummy.4* cells, while the bitlines bus through the *dummy.2* and *rommid.2* cells. The select lines bus through the *plac.g*, *placm.g* and *romword.2* cells.

The following discussion describes the cellular structure of the ROM macrocell in more exact terms. Fig. B.14 shows the general organization of a ROM macrocell.

The rows "rombtm", "rommid", and "romtop" vary only according to the number of output lines (N) and address inputs (K). From left to right, the cells in "rombtm" are:

rombtm.1

N/2 instances of *rombtm.2*

rombtm.3

K instances of *rombtm.4*

rombtm.5

In addition, prior to the 11th, 21st, 31st etc. instance of *rombtm.2* is an instance of *rombtm.g*.

The rows "rommid" and "romtop" are organized identically. "rombtm" and "romtop" are always required; "rommid" is omitted if the upper (subprogram) ROM-plane is empty. *rombtm.2* contains pullups for the bitlines. *romtop.2* contains the output registers. Aside from these, there is no active circuitry in these three rows. Upper and lower address input lines bus through the cells *romtop.4* and

rombtm.4 respectively.

Every sixteenth word in a ROM-plane is followed by a dummy word. Other than this, a ROM-plane is simply a stack of ROM-words. Some of the cells are different in alternating ROM-words to increase density by mirroring. "Even" ROM-words, meaning the first (bottommost) ROM-word and every second ROM-word above it, contain the following cells from left to right:

romword.1

N instances of either *plac.0* or plac.1

romword.2

K instances of either *pladec.0*, *pladec.1* or *pladec.x*

romword.3

In addition, prior to the 21st, 41st, 61st etc. instance of *plac.0* or *plac.1* is an instance of *plac.g*.

Odd ROM-words are identical except that the cells *plac.0*, *plac.1*, *plac.g*, *romword.1*, *romword.2*, *romword.3* are replaced by *placm.0*, *placm.1*, *placm.g*, *romwordm.1*, *romwordm.2*, *romwordm.3*.

It was found convenient to design the cells *romwordm.1*, *romwordm.2*, *romwordm.3* without including any geometries. These cells have a non-zero size for tiling purposes.

Whenever it is necessary to add a word to a ROM-plane in order to make the number of words even, the cells "placm.0" and "pladec.x" are always used to program the added row.

The cells in a dummy word are:

dummy.1

N instances of *dummy.2*

dummy.3

K instances of *dummy.4*

dummy.5

In addition, the cell *dummy.g* is placed prior to the 21st, 41st, 61st etc. instance of *dummy.2*.

The cell structure of an FSM differs from that of a ROM in two ways:

1) The cells *romtop.4* in the top row are replaced by the cell *fsmdrv*. These cells contain a register and drivers for the decoder (and-plane).

2) An FSM always has only a single ROM-plane.

B.5 Arithmetic Unit

The AUIO macrocell consists of the bit-slice arithmetic unit, described in this section, and the processor I/O section, described in Section B.6. The processor I/O section (which is always included, since any processor requires I/O) sits on top of the arithmetic unit, and connects to the bit-slices through the "mbus". The physical organization is illustrated by Fig. B.15.

A block diagram of the arithmetic unit is given in Fig.3.7 in Chapter 3. A schematic of the arithmetic unit bit-slice is given in Fig.B.16. Each of the cells *au* contain circuitry for an odd and an even bit slice; the cell *au.ctl* contains circuitry for the two most significant bits and additional control circuitry. Schematics for these additional control circuits are given in Fig.B.17. In these schematics, the

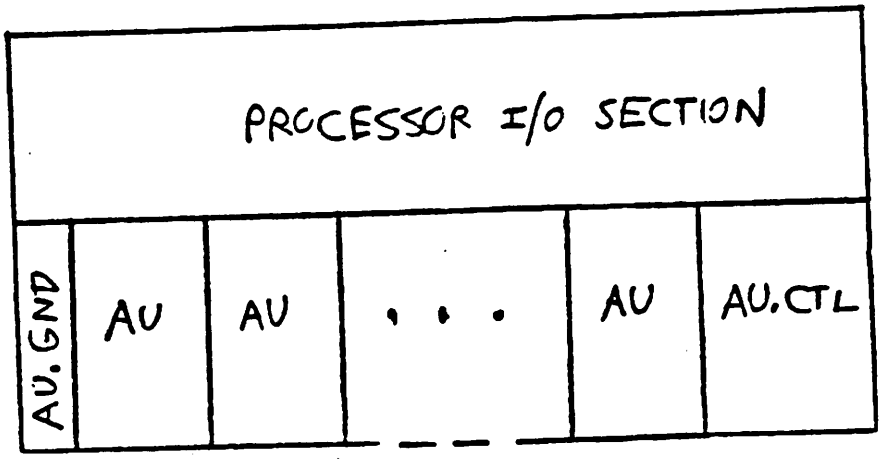


Fig.B.15 AUIO organization

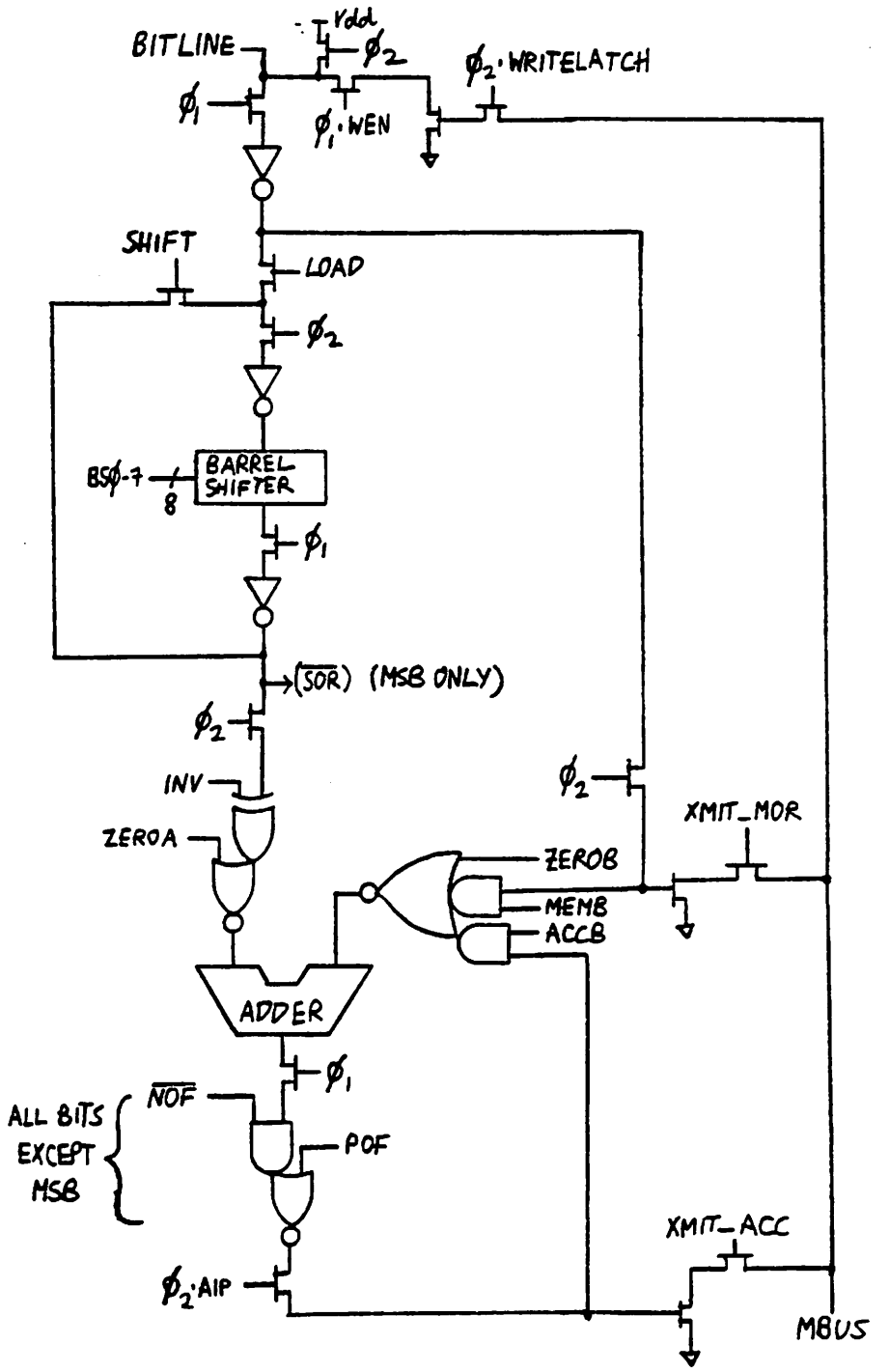


Fig.B.16 Arithmetic unit bit-slice schematic

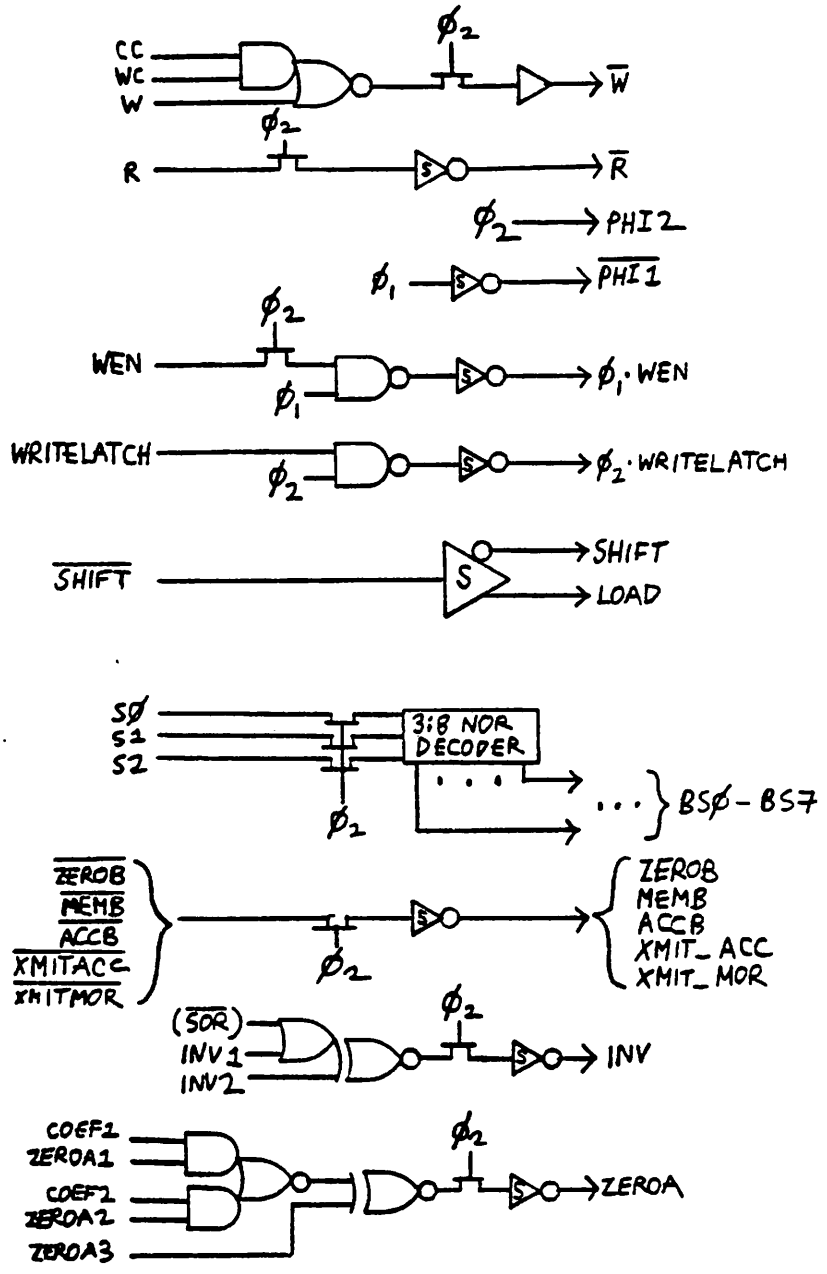
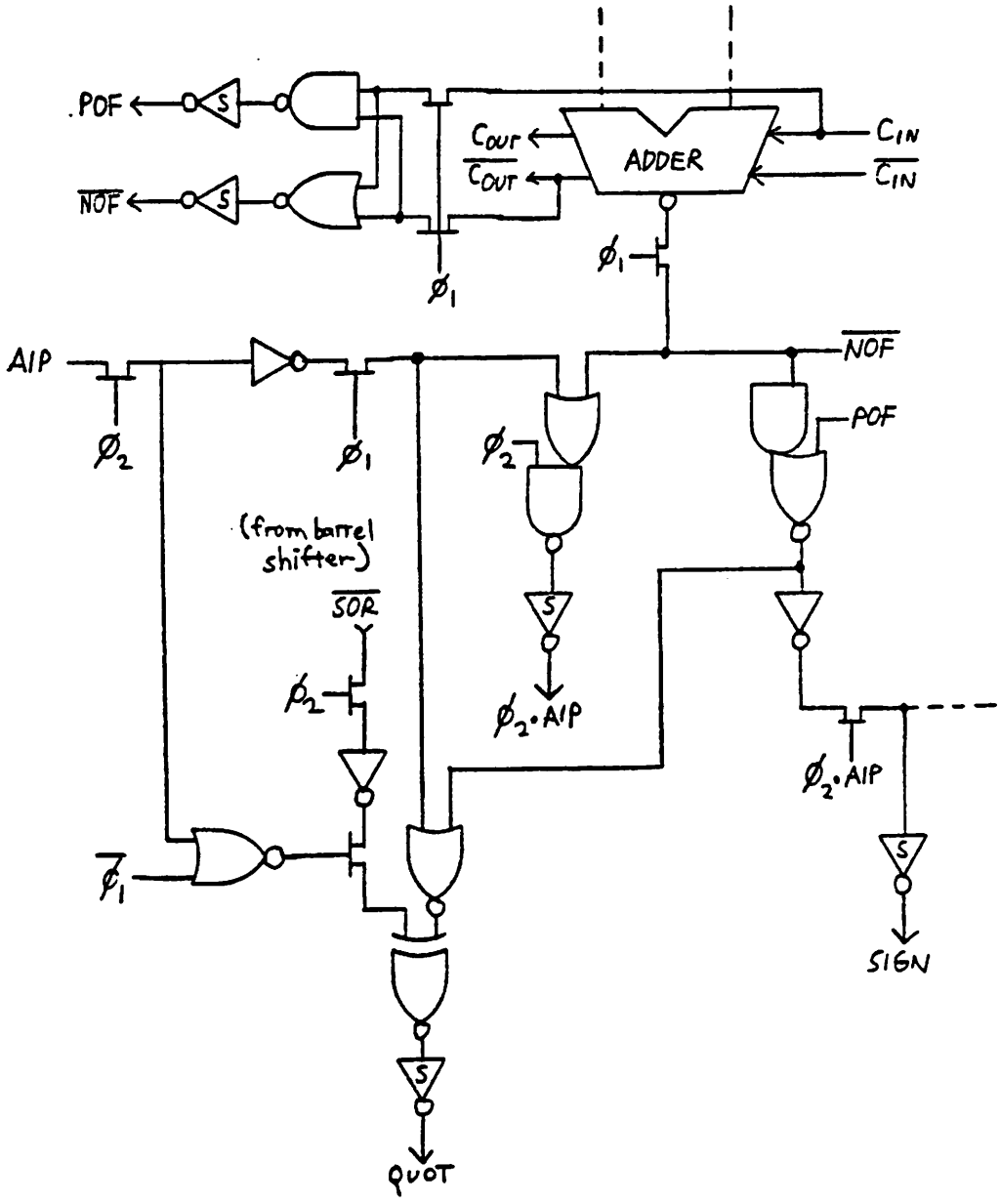


Fig.B.17 Additional circuitry for arithmetic unit control slice (1/2)



(additional circuitry for arithmetic unit control slice 2/2)

physical ordering of signal lines differs between the drawings and the actual cell layouts. I should therefore be noted that the following signals, named in these schematics, originate from the left side of the *au.ctl* cell and bus across the *au* cells:

phi1.wen

phi2.writelatch

shift

load

bs0, bs1, bs2, bs3, bs4, bs5, bs6, bs7

inv

zeroa

zerob

memb

accb

xmit_mor

pof

nof*

phi2.aip

xmit_acc

In addition, there are cell-to-cell connections in the adder carry chain and in the barrel shifter.

The cell *au.gnd* connects the true and inverted carry-ins of the LSB of the adder in the adjoining *au* cell to GND and Vdd, respectively.

The arithmetic unit connects to the RAM macrocell throught the "bitline" terminals in the bit slices; and through the four terminals "phi2", "phi1*", "read*"

and "write*" along the bottom of the control cell *au.ctl*.

Control signals from the control ROM enter *au.ctl* from the right side. These control signals control both the arithmetic unit and the RAM macrocell. Also entering the right side of *au.ctl* are the coefficient inputs "coef1" and "coef2". Leaving the right side of *au.ctl* is the serial output "quot", and the signal "sign", which may be used as an input to the FSM macrocell.

There are four pipeline registers in the arithmetic unit: MOR, SOR, ACC and MIR. Arithmetic unit operation is best understood by the effect of the various control inputs on the contents of these registers. A description of the various control lines for the arithmetic unit follows. Control signals whose names end with "*" are active low; others are active high.

zerob* -- forces adder B inputs to be zero

memb* -- AND's adder B inputs with MOR

accb* -- AND's adder B inputs with ACC

(with none of the above three asserted, the B inputs are all ones)

writelatch -- selectively loads MIR from the "mbus".

(MIR differs for the other three registers in that it is transparent and selectively loadable. The other three registers are master-slave and are loaded every cycle as follows: RAM bitlines into MOR; shifter output into SOR; adder output into ACC. The latter assumes the "aip" control is not used.)

r,w,wc,cc -- control inputs for RAM decoder are
derived from these. "r" selects a read cycle; "w" a

write cycle: "wc" and "cc" are gated together for a conditional write cycle

wen -- enables MIR (inverted) onto bitlines

shift* -- when asserted, SOR is fed to shifter inputs instead of MOR

s0, s1, s2 -- these three encode (0 to 7) the number of places the shifter's output is shifted right from its input.

inv1, inv2 -- these control the value of a variable INVOUT (which is gated into the adder A inputs) as follows:

inv1	inv2	INVOUT
0	0	~ |SOR|
0	1	|SOR|
1	0	~ SOR
1	1	SOR

zeroa1, zeroa2, zeroa3 -- these control the gating of INVOUT into the adder A inputs as follows:

zeroa1	zeroa2	zeroa3	Adder A input
0	0	0	0
0	0	1	INVOUT
0	1	0	INVOUT & coef2
0	1	1	INVOUT & (~ coef2)
1	0	0	INVOUT & coef1
1	0	1	INVOUT & (~ coef1)

xmitacc*, xmitmor* — enable ACC or MOR onto "mbus"

aip -- accumulate if positive. If asserted, ACC is loaded with the adder output only if it is positive. To allow for a two's complement divide, aip also controls the "quot" output.

B.6 Processor I/O Section

In order to simplify programming of the multiprocessor macrocell-based IC's, interprocessor and off-chip I/O is abstracted by the use of *global variables* in the input design file. This simplification is not without hardware cost, and the complexity of the I/O hardware reflects the fact that the silicon compiler must distinguish between several types of I/O.

All interprocessor communication is through the use of *scalar* global variables, as opposed to the *array* global variables that are permitted for host I/O.

Communications between processors, and between processors and the host interface, may be classified according to the type of the source and destination. Possible sources are: the processor "mbus"; the processor "quot" output; and the host interface. Possible destinations are: the processor "mbus"; the processor "coef1" or "coef2" inputs; and the host interface.

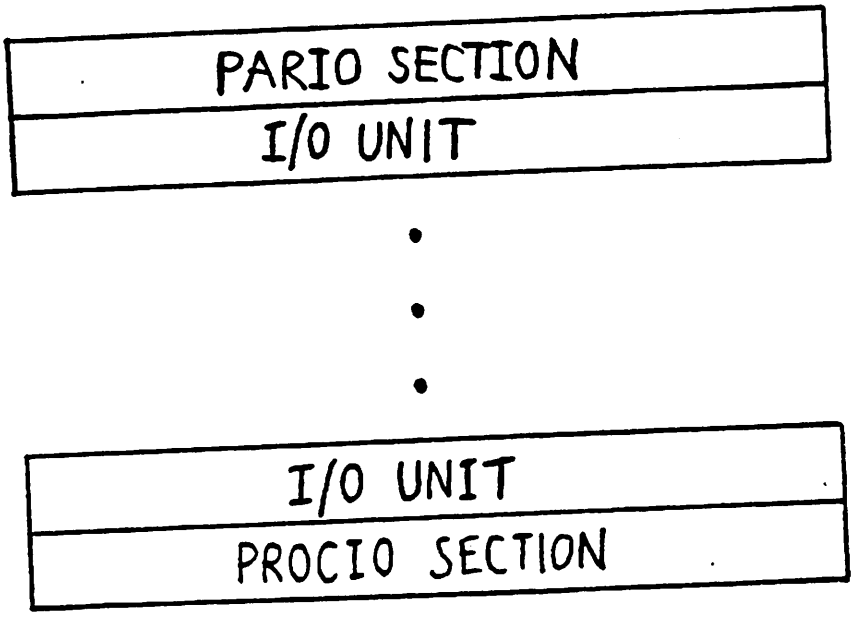


Fig.B.18 Processor I/O section organization

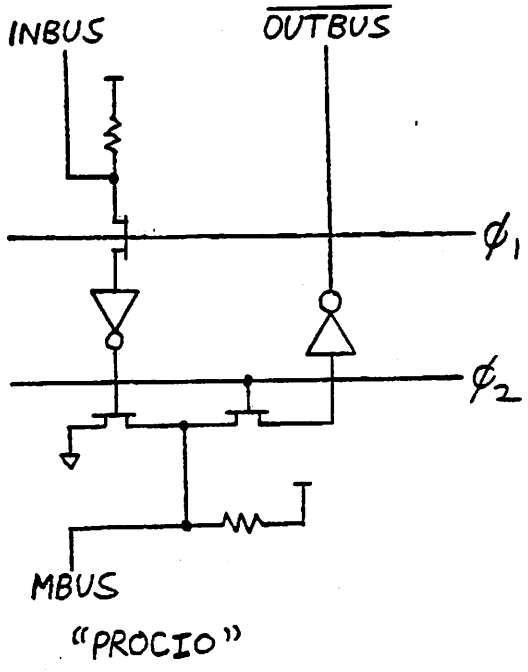


Fig.B.19 Schematic of "procio" cell

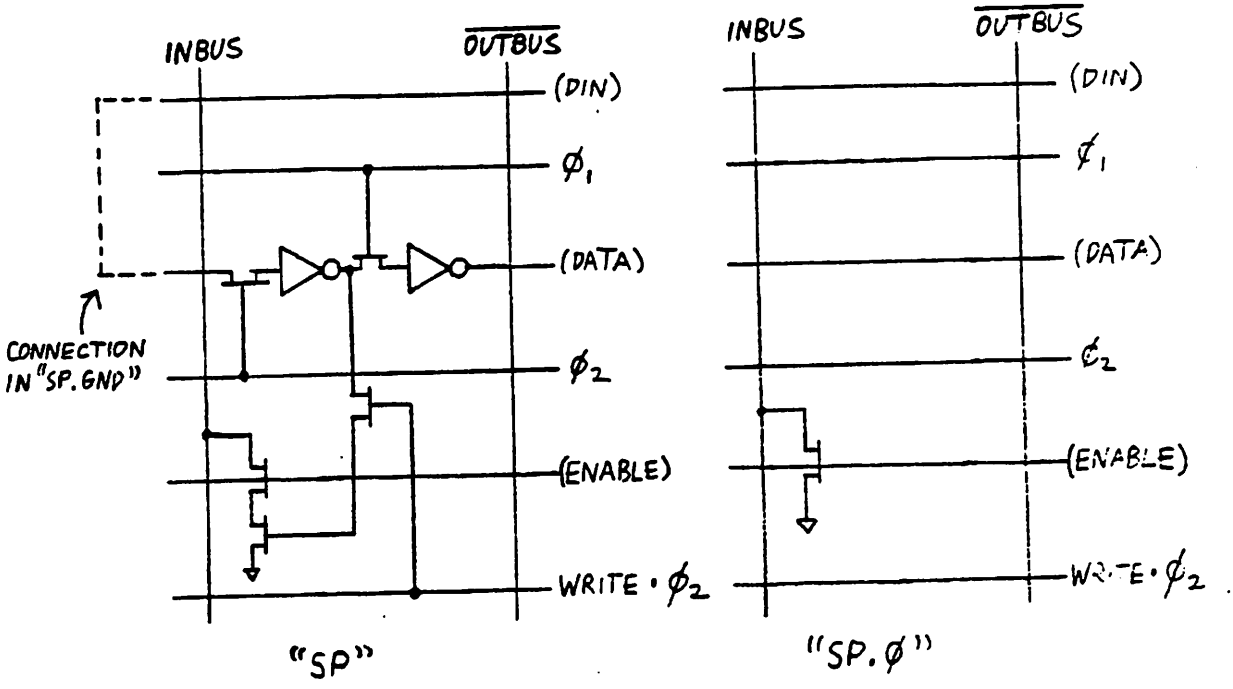
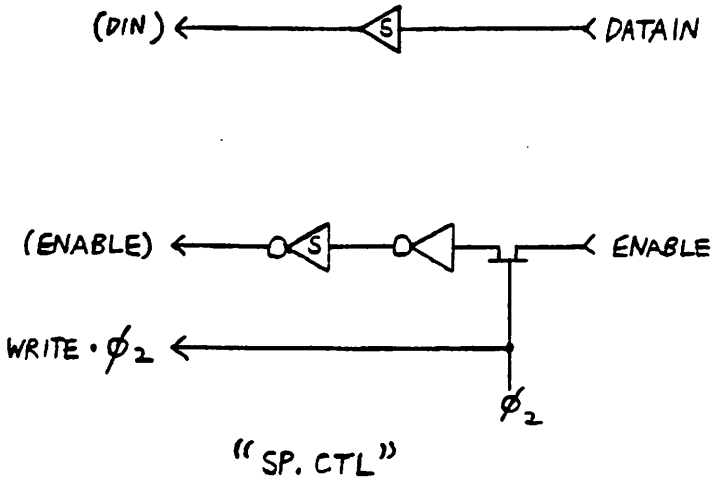


Fig.B.20 Schematic of the cell "sp"



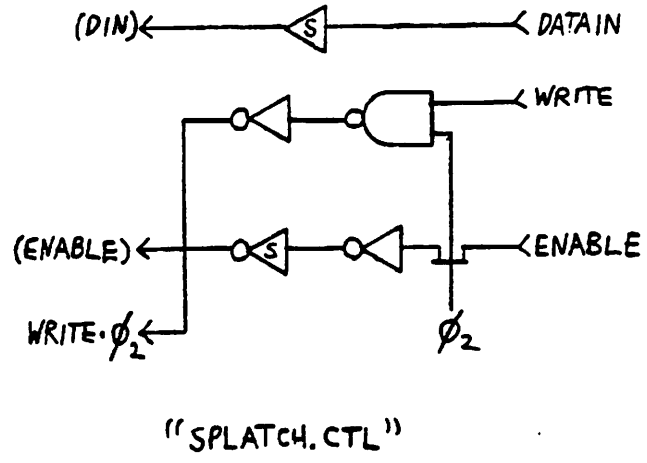


Fig.B.21 Schematics of control cells for SP and SP latch units

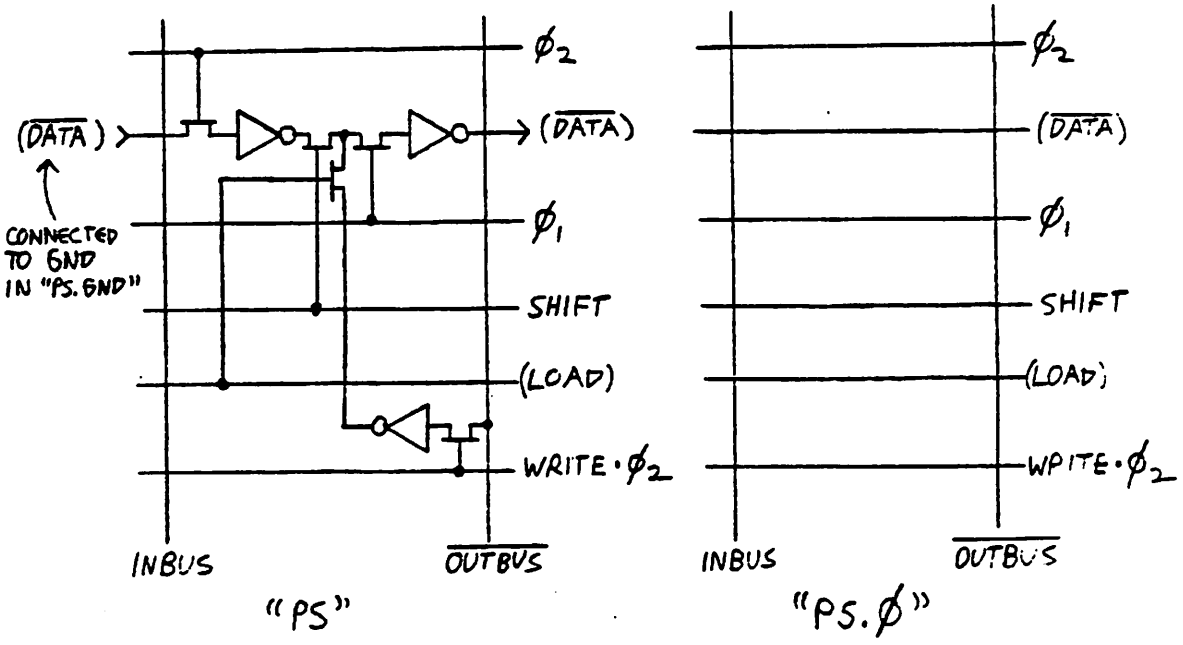


Fig.B.22 Schematic of the cell "ps"

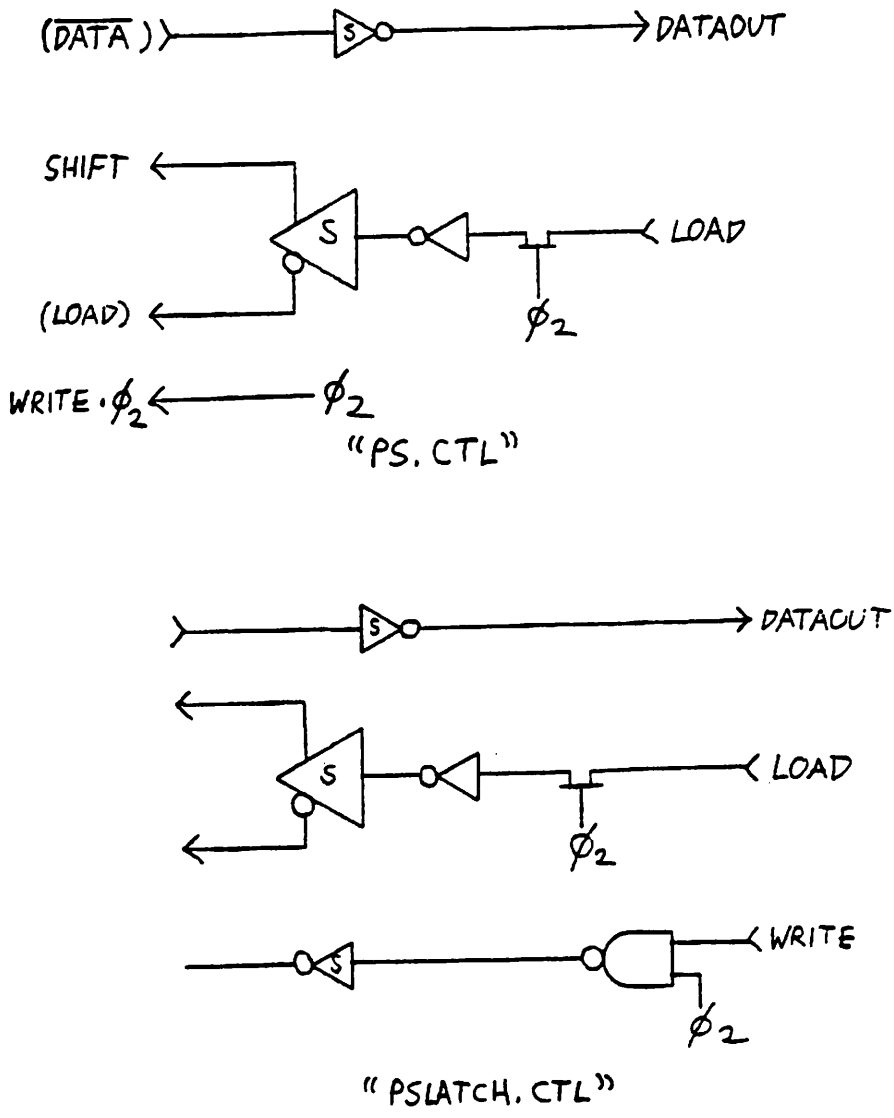


Fig.B.23 Schematics of control cells for PS and PS latch units

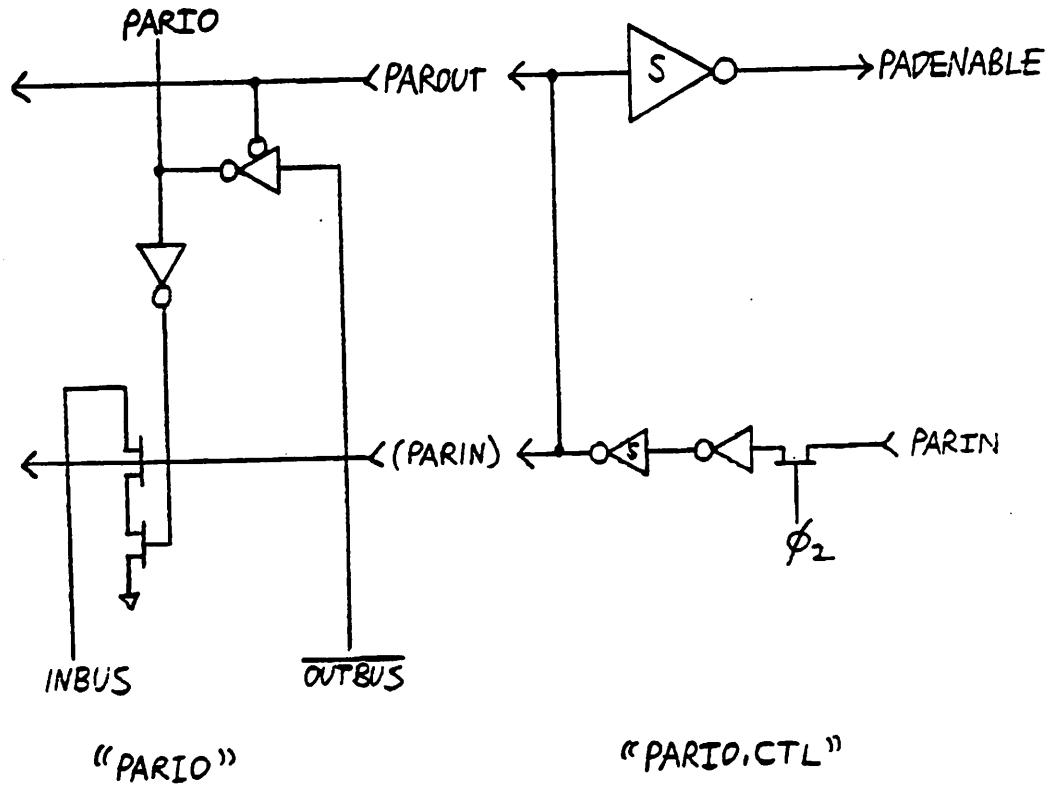


Fig.B.24 Schematics of cells in the "pario" section

In general, for every global variable declared in the source program for a chip, a serial communication line is set up. (The exceptions are globals used for parallel signal I/O.) Each serial communication line requires a "transmit unit" and a "receive unit" (hardware items consisting of serial-parallel and parallel-serial registers) as follows:

	source	destination	xmit unit	receive unit
G	mbus	mbus	PS	SP latch
N	mbus	coef	PS latch	---
G	mbus	host output	PS	host interface
G	quot	mbus	---	SP latch
	quot	coef	*NOT ALLOWED*	
G	quot	host output	---	host interface
N	host input	mbus	host interface	SP
N	host input	coef	host interface	---
	host input	host output	*NOT ALLOWED*	

(G means transmitted when generated, N when needed)

A given global is allowed to have exactly one source, but may have more than one destination. If two of the source-destination pairs for a single global are listed with a "G" in the above table, the transmit unit (if any) may be shared for both destinations. Otherwise, separate transmit units are used, and the I/O structure resembles that of two separate globals.

The individual processors are configured with the appropriate complement of SP, SP latch, PS, and PS latch units as described by the above table. These units become part of the processor I/O section.

The processor I/O section (Fig. B.18) consists of a "procio" section; a stack of zero or more I/O units (in any order); and an optional "pario" section, included only if parallel input or output (including pointer-mode iy indexing) is included.

The I/O units are the PS, PS LATCH, SP, and SP LATCH units mentioned above.

For an N-bit processor the "procio" section consists of the following single row of cells:

procio.gnd

N instances of *procio*

procio.ctl

A schematic of *procio* is shown in Fig.B.19.

There are four types of I/O units: the input units SP and SP LATCH; and the output units PS and PS latch.

An SP unit consists of a single row of cells:

sp.gnd

N instances of *sp* (or *sp.0*, see below)

sp.ctl

An SP LATCH unit also consists of a single row of cells:

sp.gnd

N instances of *sp* (or *sp.0*, see below)

splatch.ctl

Schematics for these cells are shown in Figs. B.20 and B.21. Note that the cell *sp.gnd* makes a connection to the input of the serial-parallel converter.

Both input units have an "enable" control input, the effect of which is to enable the data onto the mbus on the following clock cycle.

The SP LATCH unit has in addition a "write" control input, the effect of which is to store the shifted input in a temporary latch. For a K-bit global, the "write" control is asserted K-1 clock cycles following the cycle on which the MSB of the serial data is on the external serial input.

Simultaneous assertion of "write" and "enable" results in the new data being strobed onto the mbus.

For the case in which the processor word width is N bits, the global is K bits, and N > K, some of the cells *sp* in an SP or SP LATCH unit are replaced by *sp.0*. If the global is left-justified, the first (N-K) *sp* cells in the row are replaced by *sp.0*. If the global is right-justified, the last (N-K) *sp* cells in the row are replaced.

Please note:

(1) By default, a global is left-justified (justified towards the MSB side) unless declared as right-justified in the source file.

(2) A right justified global is an unsigned integer -- it is padded out with zero's to the left (no sign extension).

A "PS" unit consists of a single row of cells:

ps.gnd

N instances of *ps* (or *ps.0*, see below)

ps.ctl

A "PS LATCH" unit consists of:

ps.gnd

N instances of *ps* (or *ps.0*, see below)

pslatch.ctl

The PS LATCH unit has a "write" control input. This is asserted the same cycle that the data to be outputted is on the mbus, storing the data in a temporary latch.

Both output units have a "load" control input. For the PS unit, this is always asserted the same cycle that the data to be outputted is on the mbus. For the case of the PS LATCH, the "load" control may be asserted on that cycle or any time thereafter.

Asserting both "write" and "load" for a PS LATCH unit sends the newly written data to the serial output. In both cases, the MSB of the data appears on the external serial data line on the cycle following the cycle on which "load" is asserted.

The cell *ps.0* may replace *ps* in the same fashion as *sp.0* replaces *sp* to allow for globals with shorter wordlengths than the processor.

Schematics for cells used in IO units are shown in Figs. B.22 and B.23.

To perform parallel I/O, the PARIO section is included. Parallel input occurs from the off-chip signal data bus to the processor I/O section. Parallel output occurs from the processor I/O section to either the off-chip bus or the APU's Y-inputs (pointer mode addressing).

A PARIO section consists of a single row of cells: *pario.gnd*, *pario*, *pario.ctl* with *pario.0* replacing *pario* in the usual fashion. Schematics for these cells are given in Fig.B.24.

A parallel output operation is done by disabling "parin" (control input to *pario.ctl*). This is done the same clock cycle that the desired data appears on the

mbus. If the destination of the word is the off-chip bus, an additional control signal, active the same cycle, is sent to an output pad for use as an off-chip strobe. There is one such additional control signal per global used for signal output. No additional control signal is used if the destination is the AAU.

A parallel input operation is done by asserting "parin" the cycle before the data is needed on the "mbus". An additional control signal for each signal input global is sent to an output pad for use as an off-chip strobe.

The control line "padenable", derived from "parin", enables the pads used for the signal I/O bus. The logic of this signal is that the signal I/O bus is enabled every cycle except those for which it must be disabled for an input operation. This allows the processor's mbus to be monitored at all times, useful for testing purposes.

B.7. RAM

Described in this section is the processor data memory (RAM). Although the term "RAM" usually refers to read-write memories, in this case some of the memory locations may be programmed with read-only constants, the result of "constant" declarations in the processor microcode definition. The RAM bitlines are routed to the bottom of the AUIO, and all RAM control lines come from the control section of the AUIO block. RAM address inputs originate either from the AAU outputs, or directly from the control ROM in the event there is no AAU.

RAM's by their nature contain large numbers of small cells. Suppose that the RAM has M N-bit words and K address inputs. The bottom row consists of the following cells, from left to right:

rambtm.0

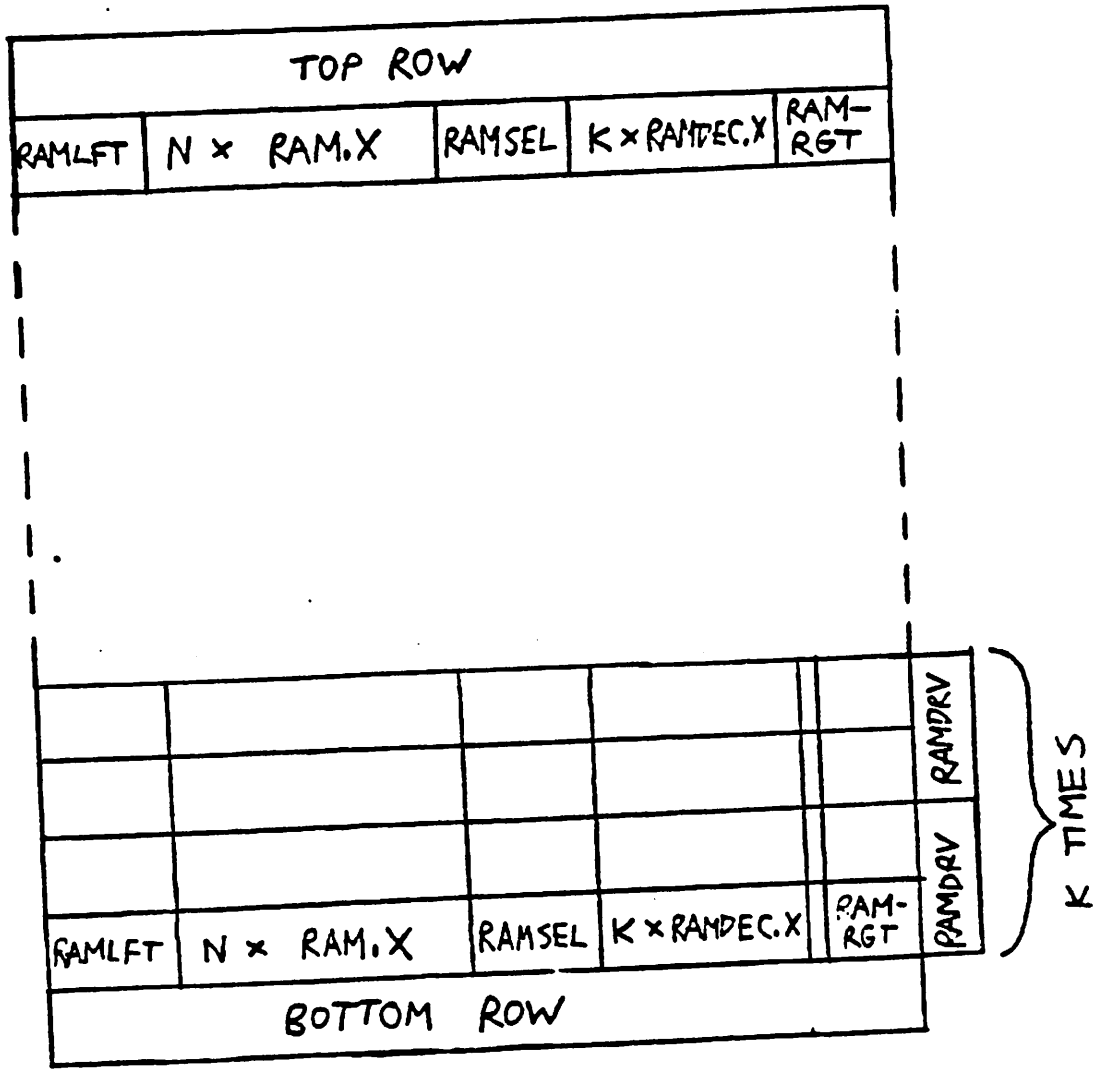


Fig.B.25 Organization of the RAM macrocell

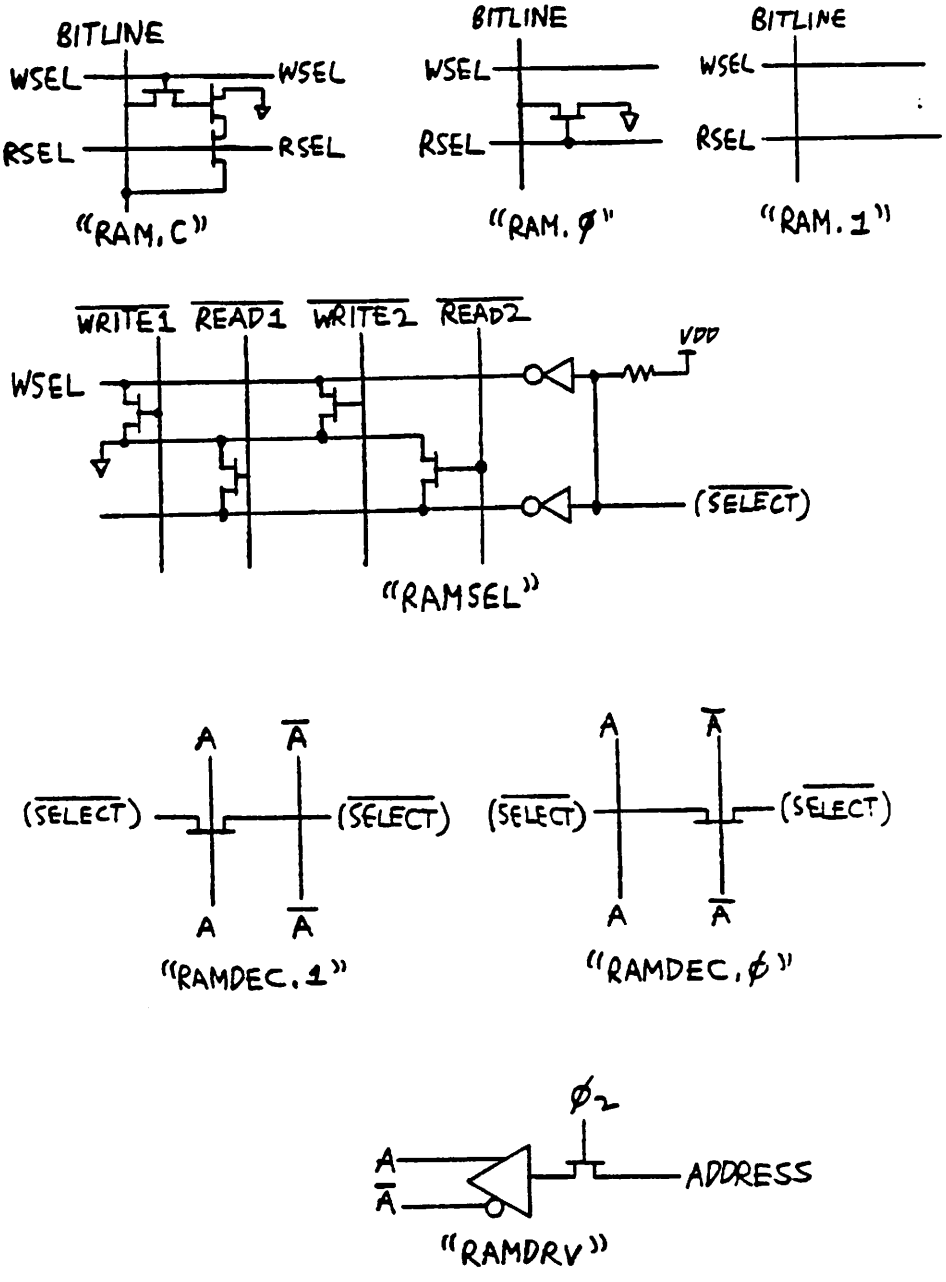


Fig.B.26 Schematics of cells in the RAM macrocell

rambtm.1 (N times)

rambtm.2

rambtm.3 (K times)

rambtm.4

Above the bottom row are M rows, one for each word, as follows:

ramlft

ram.c, *ram.0*, *or ram.1* (N times)

ramsel

ramdec.0 or ramdec.1 (K times)

ramrgt

[optional address input cells; see below]

Finally there is a top row:

ramtop.0

ramtop.1 (N times)

ramtop.2

ramtop.3 (K times)

ramtop.4

The overall organization is illustrated in Fig. B.25.

Some of the middle rows have cells on their right side that allow the address lines to be buffered and routed into the decoder. If there is one or more address line, cells are included in the first of the middle M rows. If there are two or more address lines, cells are included on the third of the middle M rows; three or more.

on the fifth; an so on up to seven address lines.

On the first of the middle M rows the address input cells are *ramcon.0* followed by *ramdrv*. On the third row, the cells are *ramcon.1* followed by *ramdrv*. This continues on up to the thirteenth row, where *ramcon.6* and *ramdrv* are included for the case of seven address lines (seven is the maximum allowed).

Because of this organization, is is not allowed for the number of words in the RAM to be one, three, or five.

The "ramcon" cells connect the complementary outputs of the *ramdrv* cells to the true and inverted address lines running vertically through the decoder. The "ramcon" cells are not illustrated in Fig. B.25; they extend from the left edge of the *ramdrv* cells over the decoder plane.

Cells used in the RAM are shown in Fig.B.26. The bitline is precharged during PHI2, and read accesses occur during PHI2*. Read data is strobed off the bitline by PHI1. Weak pullups are also used on the bitline, to restore charge lost due to coupling to the precharge clock.

Precharging is also used during a write. Note that following the trailing edge of PHI1, the WSEL signal must turn off the 3-T cell's pass gate prior to the rising edge of PHI2, when precharging begins. For this reason RAM operation depends on sufficient clock separation.

The decoder consists of a NAND decoder for the address lines, followed by two NOR gates for the RSEL and WSEL signals. Other inputs to the NOR gates originate from the cell *au.ctl*. These inputs, "read1*", "write1*", "read2*", "write2*", bus through the cell *ramtop.2*, which contains no active circuitry.

The read-only cells *ram.0* and *ram.1* are used to allow for constant locations within the RAM array.

B.8 I/O Sequencer

The I/O sequencer implements the finite state machine whose state sequence diagram is given in Fig.3.13 in Chapter 3. Functionally, the I/O sequencer is part of the host interface; physically it is located with the bonding pads in *padgroup2*, described in a later section.

The sequencer is contained in the cell *iocontrol*, which is not listed in the *descriptor* file because it appears only as a subcell of *padgroup2*. A small finite-state machine has three inputs (EOS, MOF and EOF) and two bits of state corresponding to the four states shown in Fig.3.13.

The state bits are decoded to produce five signals used by the various parts of the host input and output interfaces:

fs	first sample	(state 3)
ls	last sample	(state 2)
wnp	write not in progress	(state 0)
rnp	read not in progress	(state 1)
wnp+ls		(states 0 or 2)

In addition to its destination in the host interface, the "fs" signal is used to assert the RINT* (read interrupt) signal which goes to the host.

Fig.B.27 is a schematic of the cell *iocontrol*.

B.9 Host Output Interface

Fig.3.12, in Chapter 3, shows the Host Interface macrocell as being composed of the host output interface on the bottom, with the host input interface on top. One or the other of these may be excluded if only one type of host I/O is required. This section describes the host output interface.

Global variables used for host output originate from either the "mbus" or the "quot" output of a processor. In either case the data is generated at a time

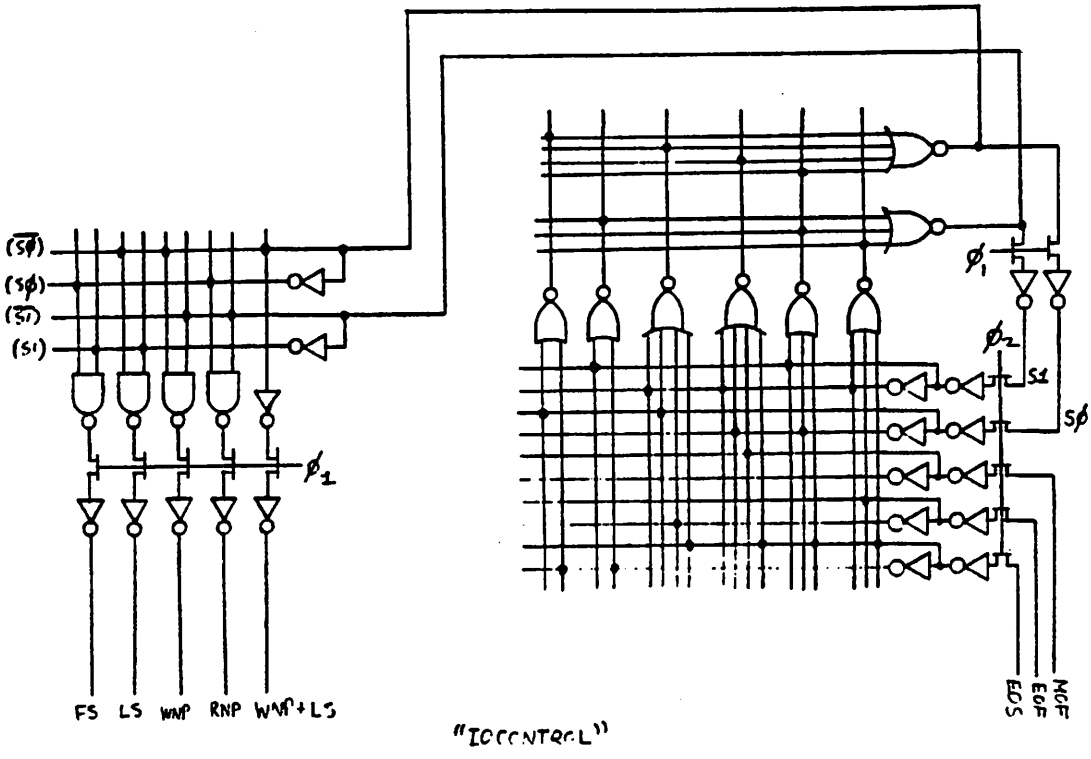
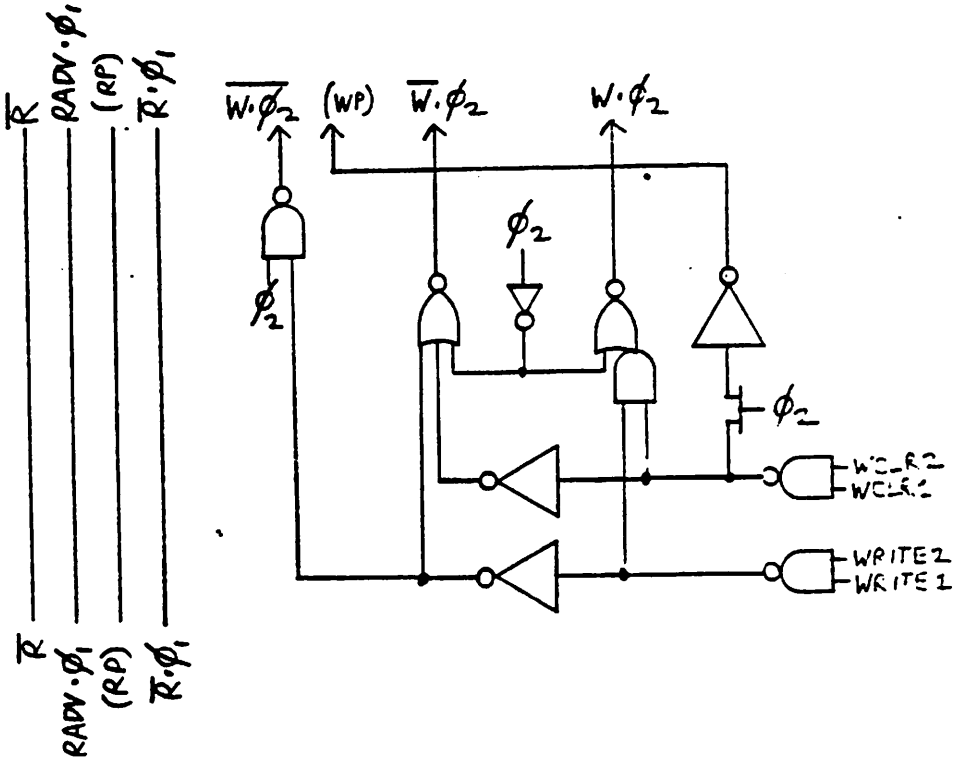


Fig.B.27 Schematic of the cell "iocontrol"

"HOSTOUT"



"HOSTOUT.CTL"

Fig.B.32 Host output section cell schematics

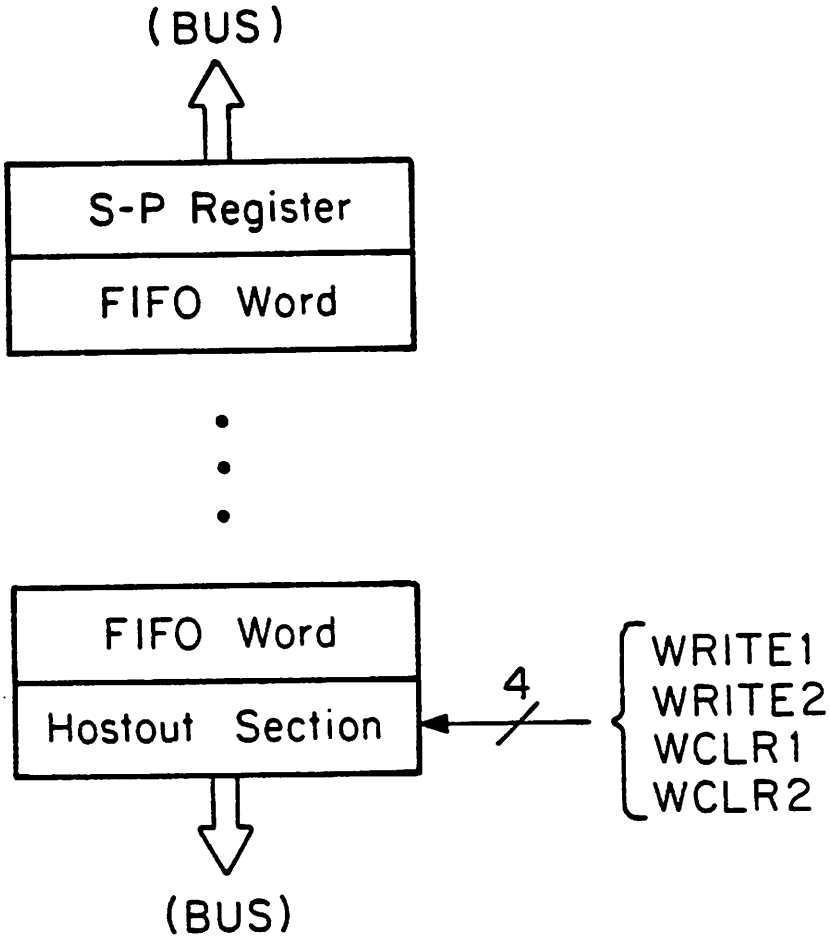


Fig.B.28 Organization of a host output FIFO section

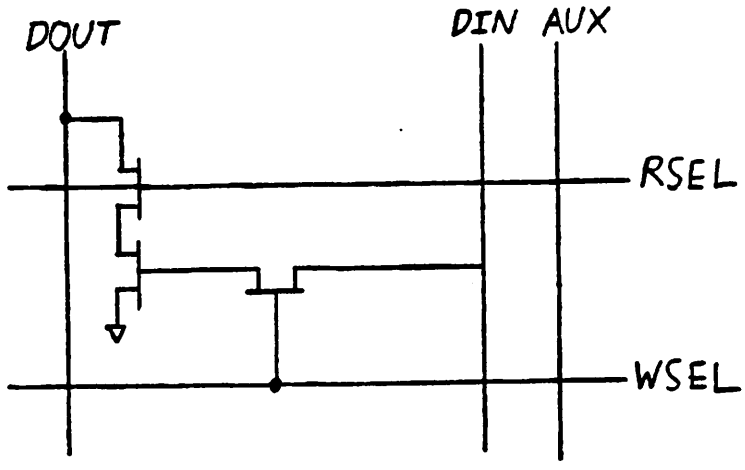


Fig.B.29 Schematic of the cell "fifo.c"

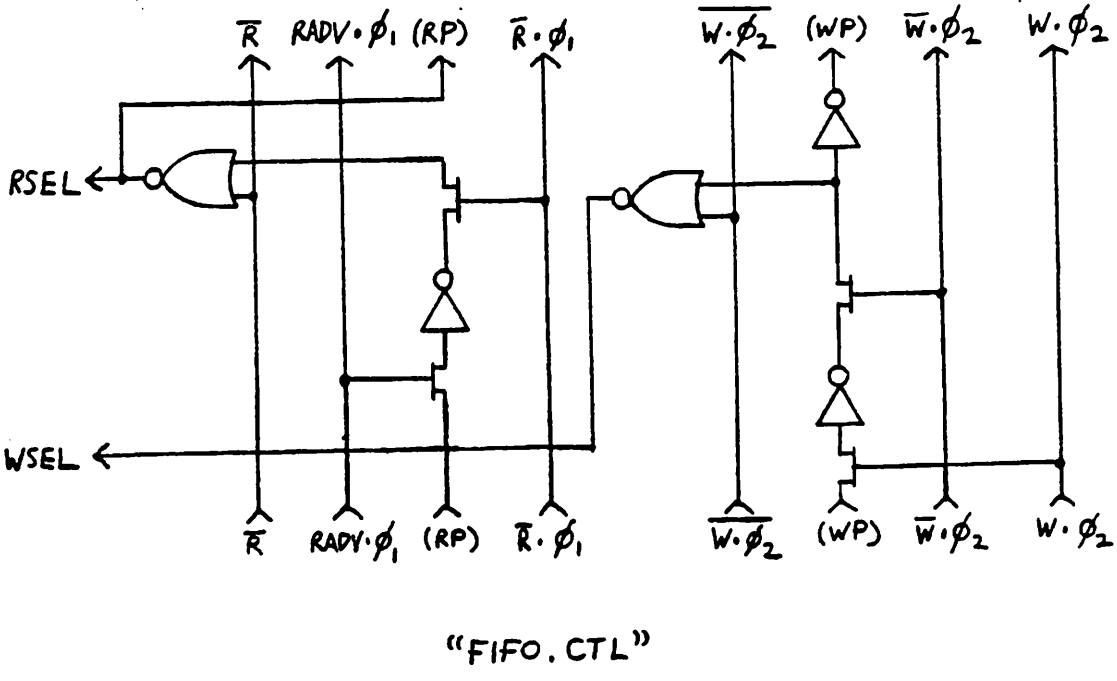
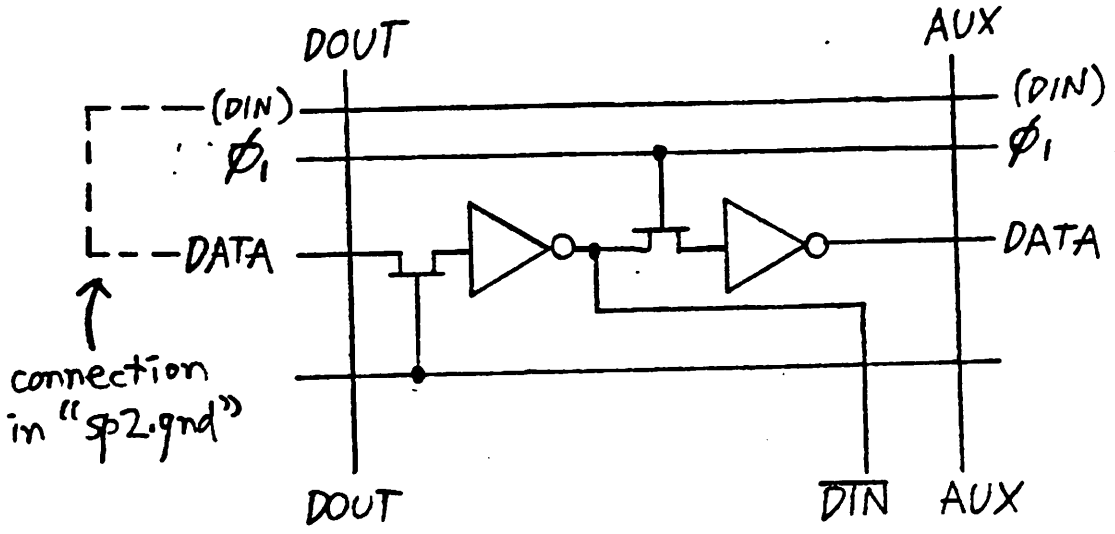
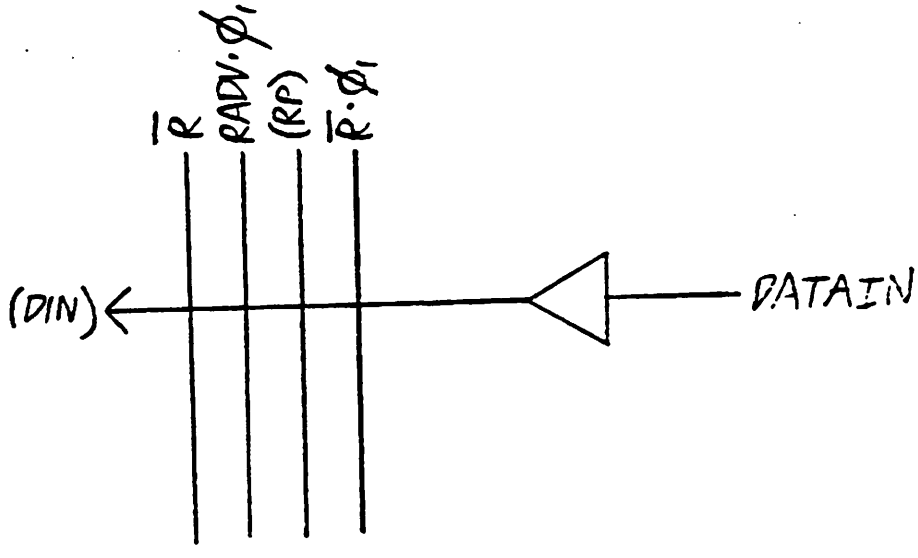


Fig.B.30 Schematic of the cell "fifo.ctl"



"SP2"



"SP2.CTL"

Fig.B.31 S-P register cell schematics

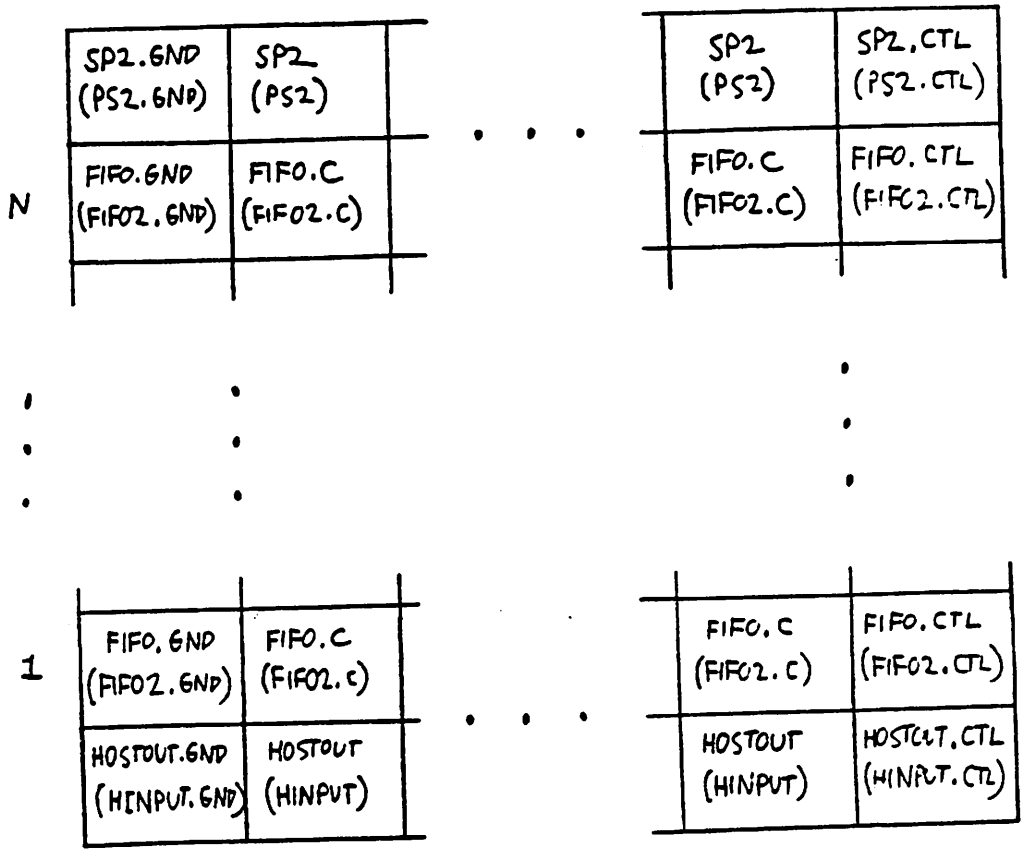


Fig.B.33 Illustration of FIFO section assembly

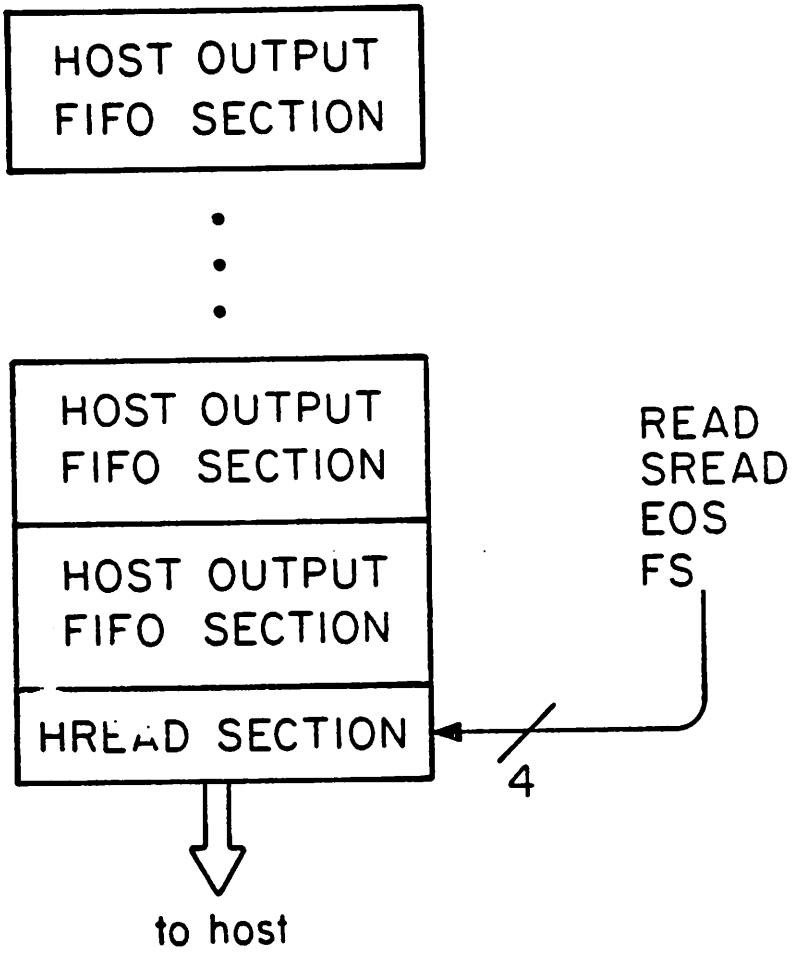


Fig.B.34 Organization of the host output interface

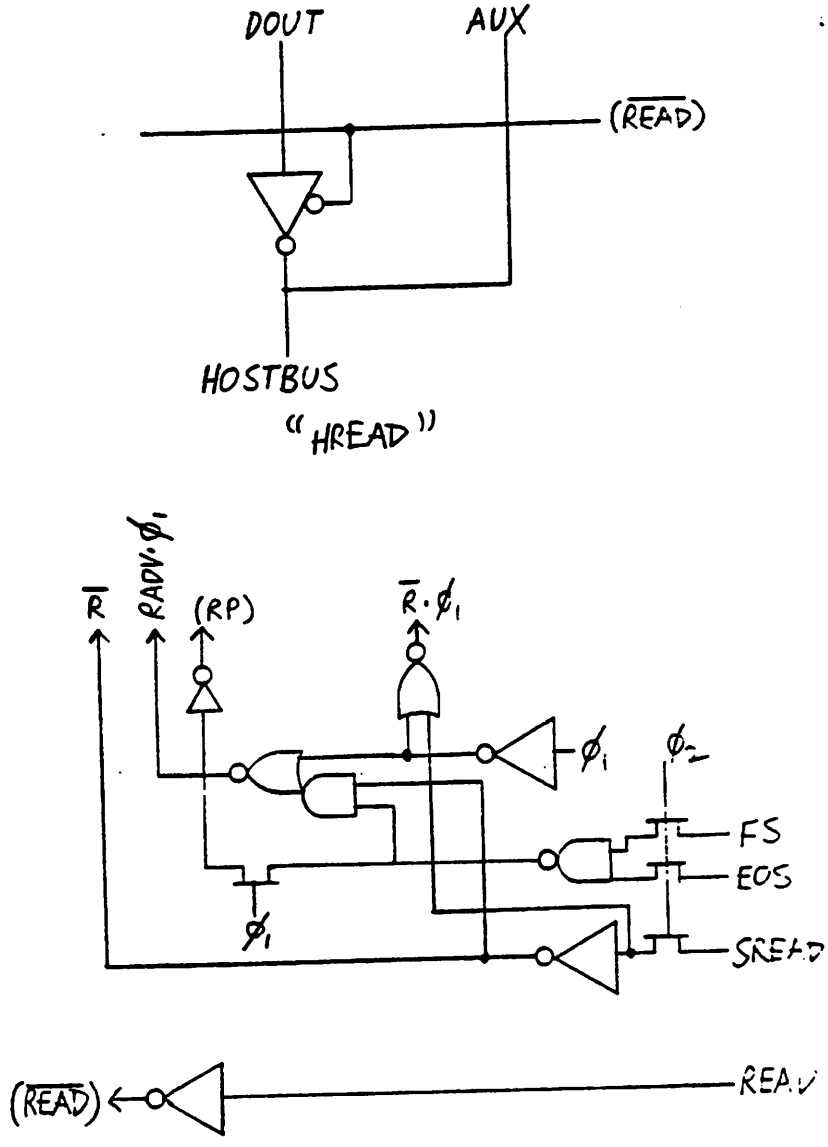


Fig.B.35 Host read cell schematics

determined by the processor.

Fig.3.14 in Chapter 3 shows how host output FIFO sections are combined into the host output interface. Fig.B.28 shows the structure of one of these sections. Suppose the host data bus is N bits wide. Each "FIFO word" is composed of the cells *fifo.gnd*, *fifo.c* (N times), *fifo.ctl*, whose schematics are given in Figs. B.29 and B.30. The "S-P register" contains the cells *sp2.gnd*, *sp2* (N times), *sp2.ctl* whose schematics are given in Fig.B.31. It should also be noted that the cell *sp2.gnd* makes a connection to the serial input of the *sp2* cell tits

The lower section, labeled "hostout", contains the cells *hostout.gnd*, *hostout* (N times), *hostout.ctl*. See Fig B.32 for schematics.

Fig. B.33 illustrates how the cells in a host output FIFO section are assembled. (Cell names in parenthesis refer to host input FIFO sections, discussed later.)

First the way in which data enters these FIFO sections will be discussed. In simple terms, the serial data enters an 8 or 16 bit S-P register, and a timing signal "write1" from the control ROM of the source processor tells the host output FIFO section when to write the data into the FIFO.

There are two complications: (1) The data may be longer than the 8 or 16 bits of the register; and (2) The data may be an array of words rather than a single word.

If neither of these complications is present, there is only one word in the FIFO section.

If the data is in an IX-indexed array, the "write1" control is asserted during the subprogram, rather than the main program, of the processor. In addition, there may be an additional assertion of "write1" in the main program shortly after the end of the last subprogram iteration, to strobe the last element of the array.

If the data is longer than 8 or 16 bits, enough words in the FIFO section are

included to hold the entire data. The "write1" signal from the source processor must be asserted several times at 8 or 16 cycle intervals.

In addition, the source processor must determine when to clear the FIFO write pointer so that the next write occurs at the first word. This is done either during the last sample of the frame or the first sample, depending on when the data is to be captured. For the case of scalar or IX-indexed globals, "wclr1" is a control line from the source processor, and "wclr2" is wired to either "ls" or "fs". The desired data (generated during the last sample) is written into the FIFO, and the FIFO's write pointer is subsequently allowed to increment past the last word of the FIFO section so that data from other samples in the frame is ignored.

The situation is more complex for IY-indexed data. Here, "wclr2" is asserted when the IY counter is zero, rather than just at the end of the frame. "wclr2" comes from a "test" output of the AAU for the source processor. To prevent data from changing while the host is reading it out of the FIFO, "write2" is wired to "rnp" (read not in progress) to suppress writing during this interval. Thus, the host gets a set of y-indexed values sampled near the end of the frame.

In cases other than the IY indexing, "write2" is always asserted (i.e., wired to Vdd).

Now consider the reading of the host output interface by the host. Reading is controlled by a row of cells, *hread.gnd*, *hread* (N times) and *hread.ctl*, situated below the FIFO sections. The arrangement of the "hread" section at the bottom of the host output interface is shown in Fig.B.34. Schematics of the cells *hread*, *hread.ctl* are given in Fig.B.35.

First note that the read pointer moves from section to section, as opposed to the write pointers which are independent for each section. This pointer is cleared at the end of the first sample by gating together "eos" and "fs". This is the same point in time when the read interrupt is generated.

The synchronized read signal "sread" is used to read words from the FIFO's. The read pointer increments each time a read is performed. The unsynchronized signal "read" enables the data onto the three-state host bus. (Note that the synchronized signal could cause a bus conflict were it used to enable the data.)

B.10 Host Input Interface

The host input interface (Fig.B.36) contains a "hwrite" section, plus one or more "host input FIFO sections". Each host input FIFO section (Fig.B.37) contains a "hinput" section; one or more "fifo2" words; and a "P-S register". Again assume the host data bus is N bits wide.

The "hinput" section contains the cells *hinput.gnd*, *hinput* (N times), *hinput.ctl*. The "fifo2" words contain the cells *fifo2.gnd*, *fifo2* (N times), *fifo2.ctl*. The "P-S register" contains the cells *ps2.gnd*, *ps2* (N times), ps2.ctl. Schematics for those cells with active circuitry are given in Figs. B.38 to B.41.

One feature of the host input interface is that it is possible to read a word out of a FIFO section more than once before advancing to the next word. This is why there are separate "read" and "radv" (read advance) controls for the "hinput" section. This allows a processor to reference a host input global more than once in its program without storing it locally. This feature has the limitation that the wordlength of the global variable must not exceed the width of the host data bus, since there is no way to move the read pointer backwards. Nevertheless the feature is sometimes useful.

If it is not desired to read the words more than once, the "read" and "radv" lines are tied together.

The "read" control line originates from the control ROM of the destination processor, and its timing is related to when the data is needed. The "rclr1"

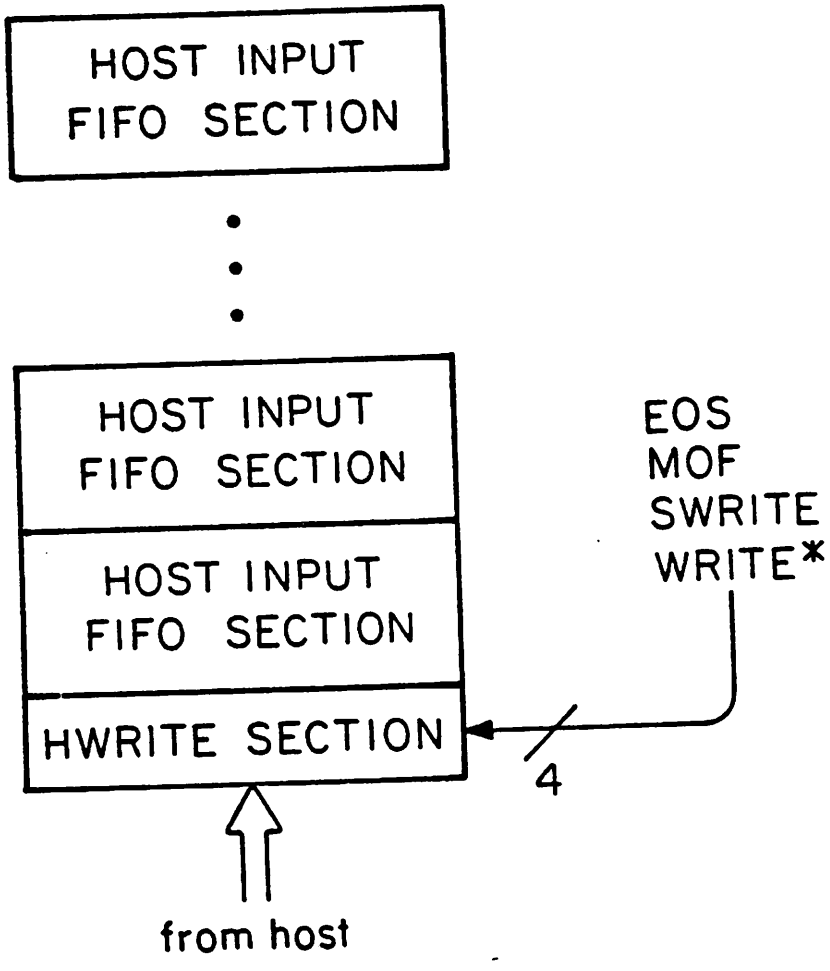


Fig.B.36 Organization of the host input interface

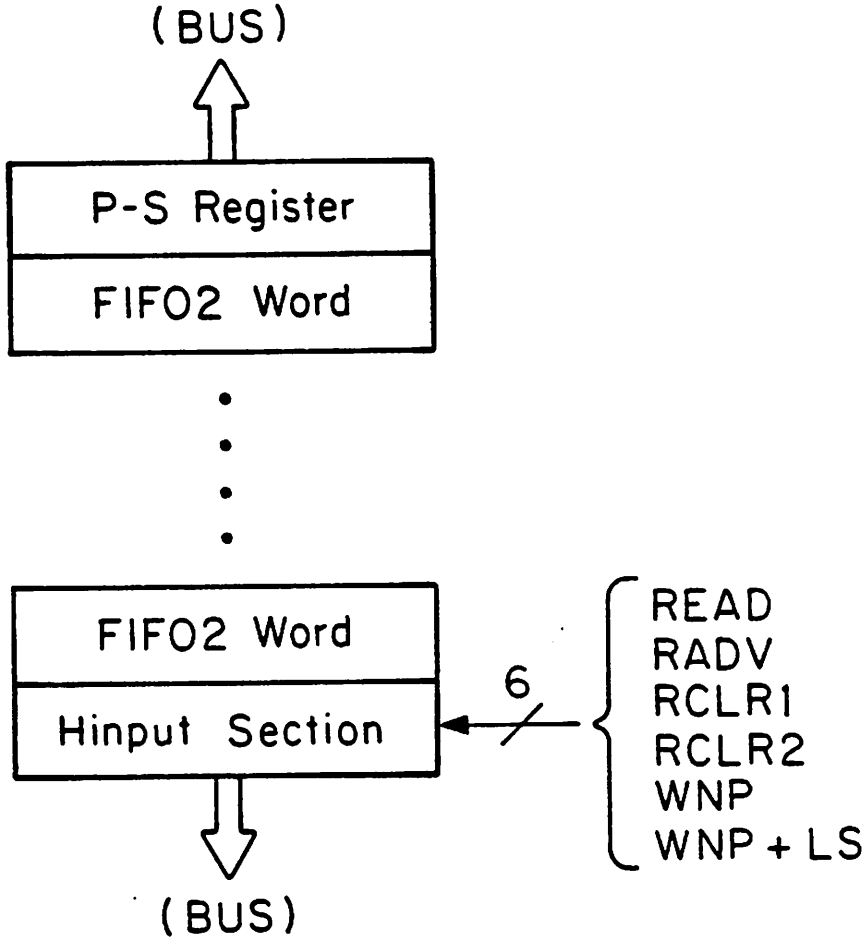
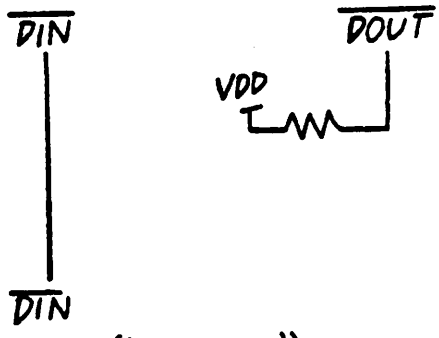


Fig.B.37 Organization of a host input FIFO section



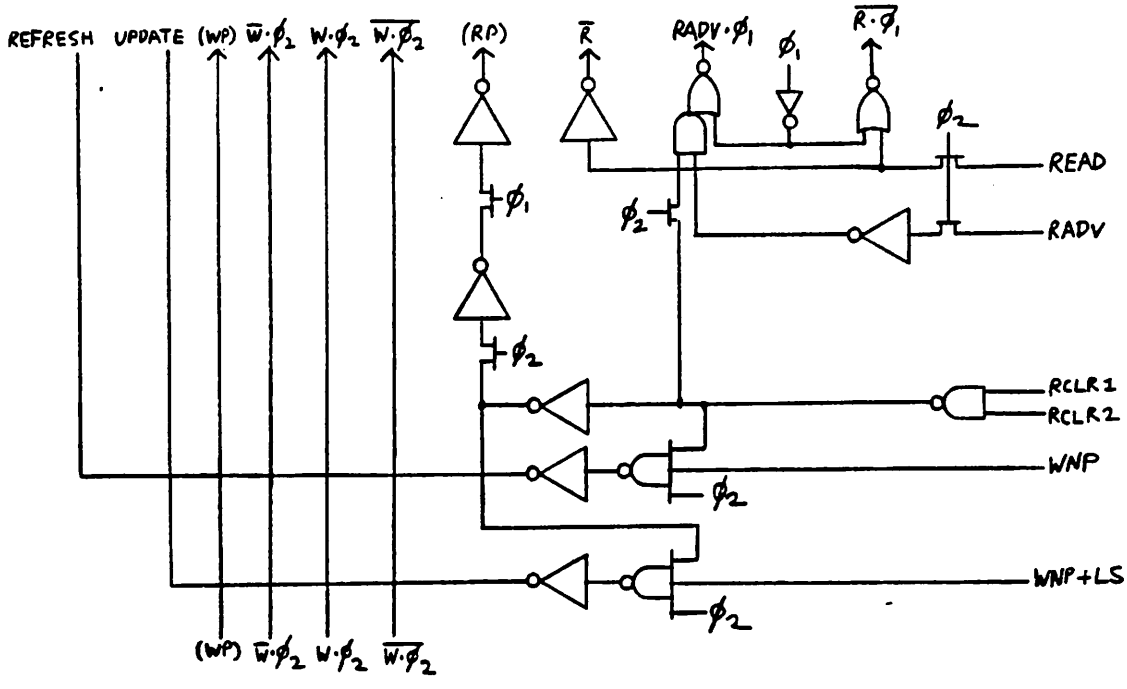


Fig.B.38 Host input section cell schematics

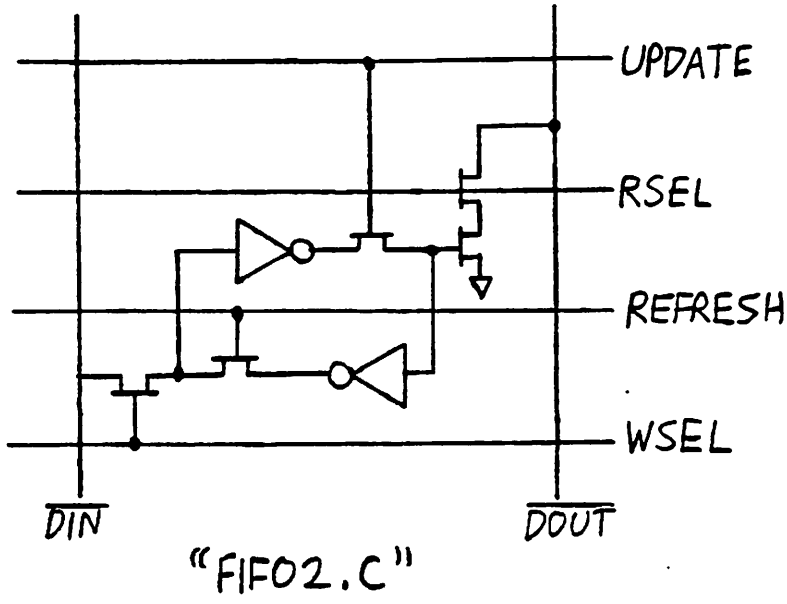


Fig.B.39 Schematic of the cell "fifo2.c"

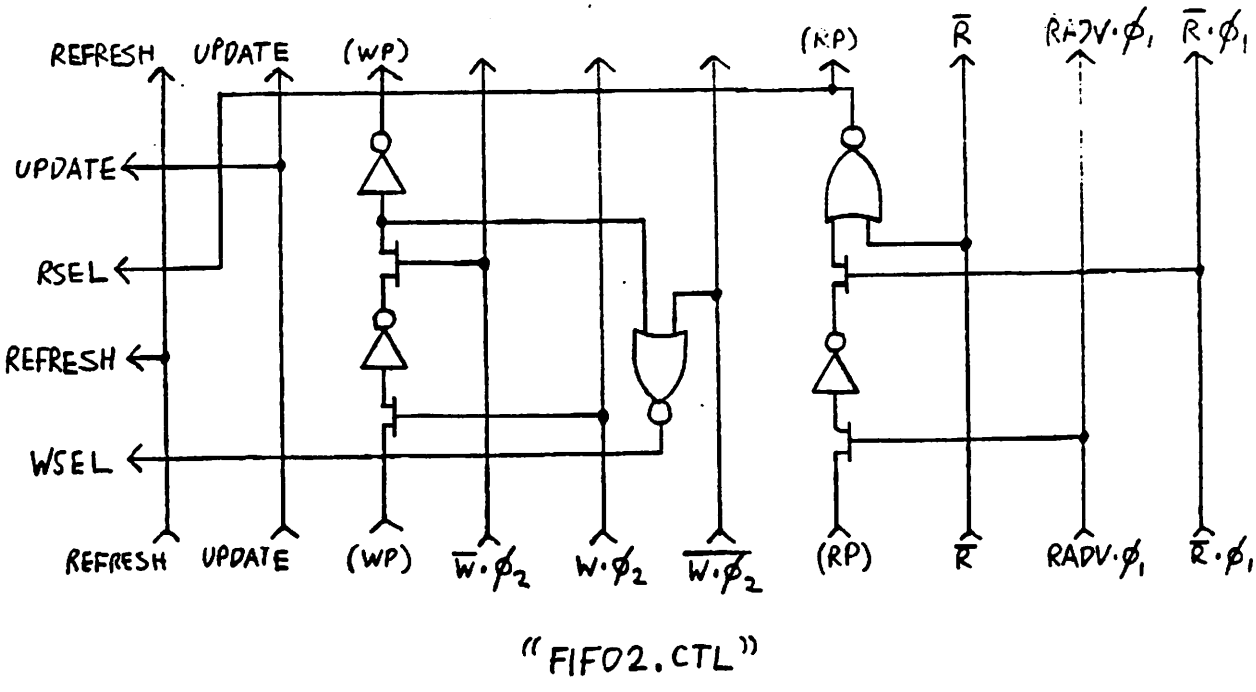
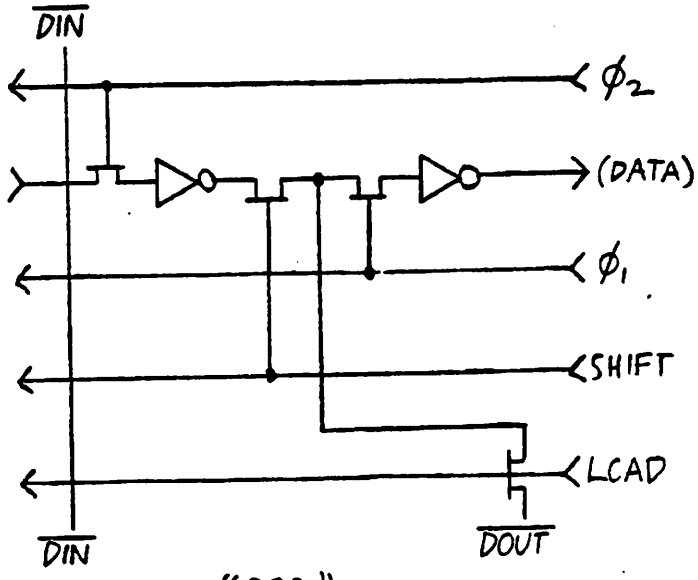
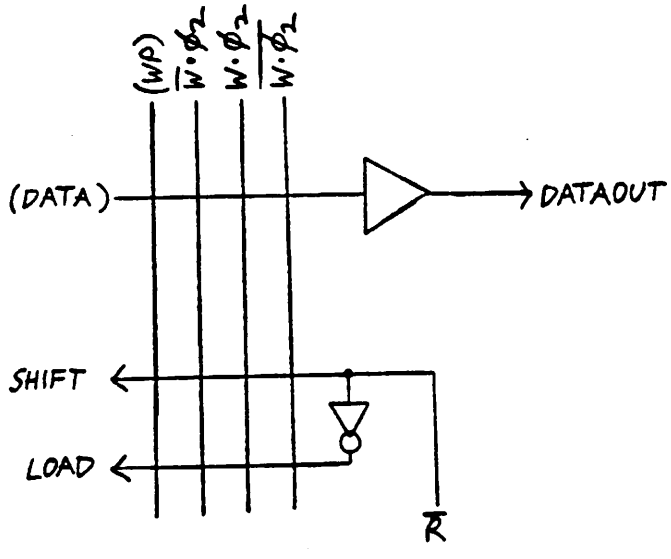


Fig.B.40 Schematic of the cell "fifo2.ctl"

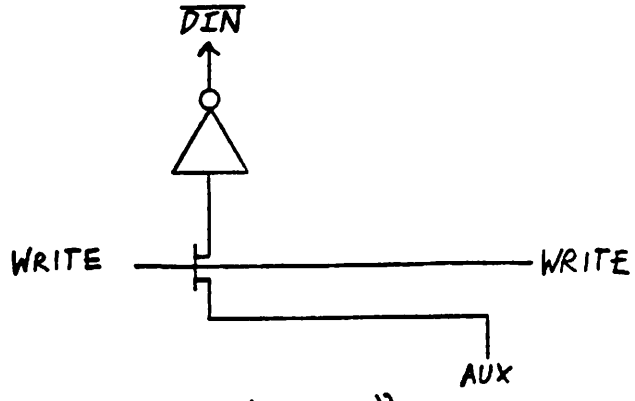


"PS2"

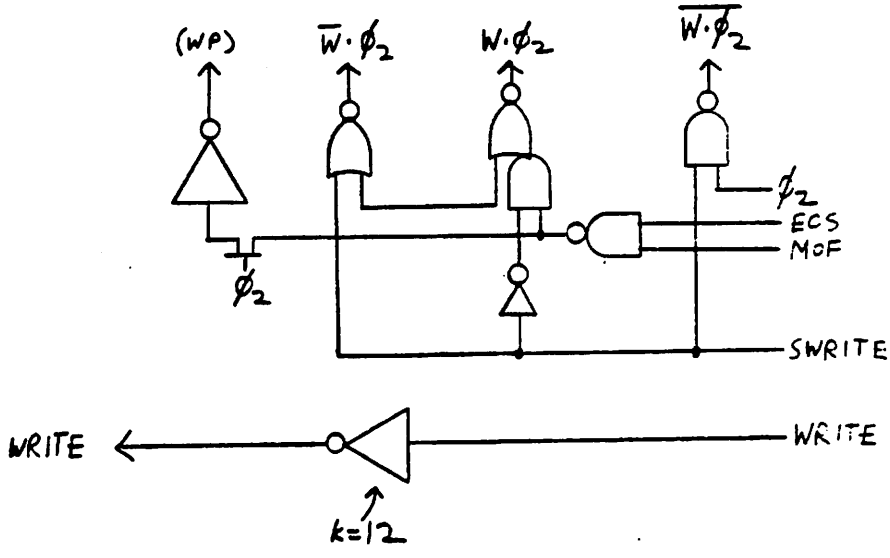


"PS2.CTL"

Fig.B.41 P-S register cell schematics



"HWRITE"



"HWRITE.CTL"

Fig.B.42 Host write cell schematics

control line not only clears the FIFO read pointer, so that the next word read is the first word in the FIFO, but controls a refresh operation to occur in the *fifo2* cells. This refreshing is necessary since the frame length may exceed a refresh interval. However, refreshing must be supressed while the host is writing into the "master" part of the FIFO. Thus refreshing occurs only if the "wnp" (write not in progress) signal is asserted.

A look at the schematic for the *fifo2* cell shows that the "update" operation must follow a "refresh" operation for the refresh to actually occur. When "wnp" is active, "refresh" is equal to the complement of "rclr1", while "update" is equal to "rclr1". The "update" control is also equal to "rclr" during "ls" (last sample), which has the effect of loading the new data into the "slave" part of the FIFO before the start of the new frame. By choosing the correct cycle for "rclr1", data referenced during any sample of the frame (including the first or last sample) will be identical.

The "rclr2" control is usually always asserted, with only "rclr1" in use. The exception is if the FIFO section stores a IY-indexed global. In this event, the "rclr2" is connected to a "IY=0" test output of the destination processor's AAU.

Writing from the host into the host input interface is straightforward. The "hwrite" section consists of the cells *hwrite.gnd*, *hwrite* (N times), *hwrite.ctl*, whose schematics are given in Fig.B.42. The FIFO write pointer is cleared by "mof" anded with EOS, which is when the write interrupt WINT* is asserted. The unsynchronized signal "write*" latches the incoming data, while the synchronized signal "write" controls the write cycle and advances the pointer.

Fig. B.33 (in the previous section) illustrates the assembly of a host input FIFO section. (Parenthesized cell names apply to host input FIFO sections, those without parentheses to host output FIFO sections.)

B.11 Bonding Pad Circuitry

Two package sizes are allowed: 40 and 64 pin DIP's. Because of the way pads are grouped, a 64 pin package is used if:

(1) The host interface is 16 bits

(2) The signal IO bus, plus the signal strobes, exceeds 19

(3) The number of signal strobes exceed 9

Otherwise, a 40 pin package is used. The signal IO bus, plus the signal strobes, may not exceed 31. The signal strobes may not exceed 15.

(The above could be refined to allow some projects with a 16 bit host interface to still fit in a 40 pin package.)

Pads are divided (somewhat arbitrarily) into four groups:

(1) The host IO bus

(2) Vdd, GND, PHI1, PHI2, READ*, WRITE*, RINT*, WINT*

(3) Up to 10 (16) bits of the signal IO bus for 40 (64) pin package

(4) GND, all strobes, and the remaining bits of the signal IO bus

These are referred to as *padroup1*, *padgroup2*, *padgroup3* and *padgroup4*.

The following procedure for positioning the pad groups around the edges of a chip has been used.

Following global place and route, the assembled padless circuit must have the signal IO bus along one edge, and the host IO bus along a separate edge. Place group (3) along the first edge, and group (1) along the second. Either these two edges are adjacent or opposite. If adjacent, place group (4) adjacent to group (3).

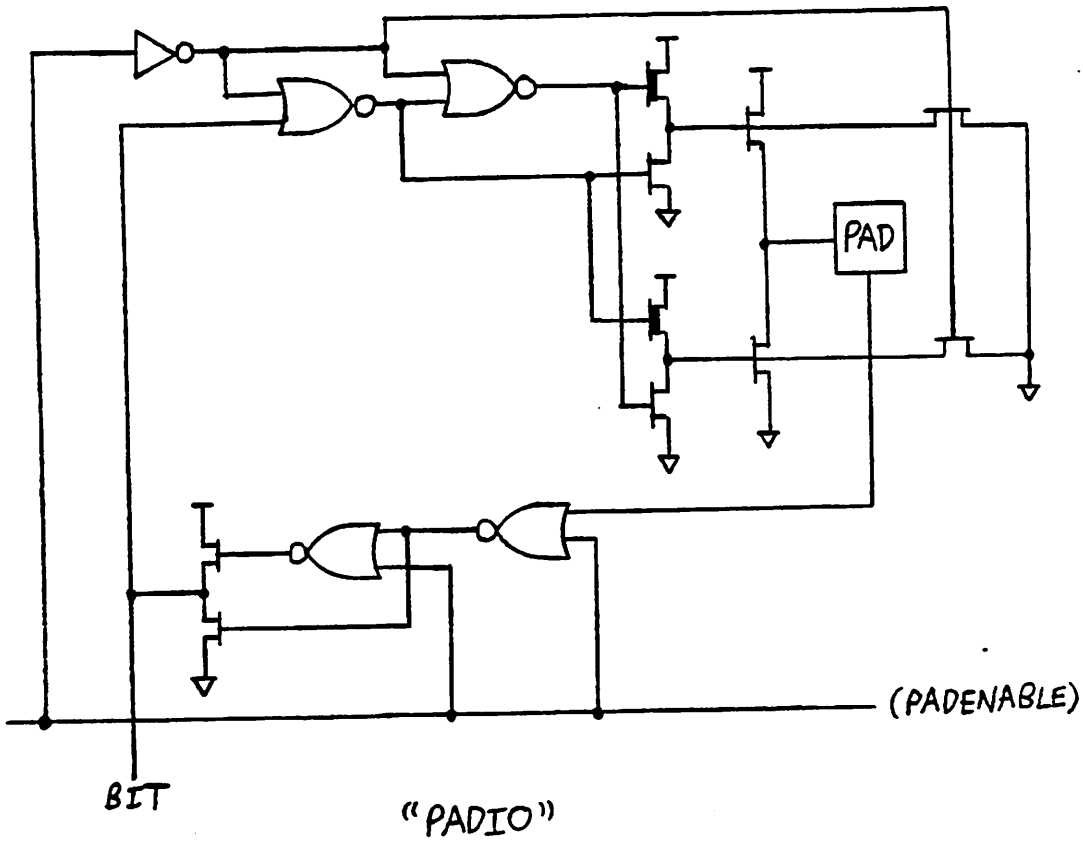


Fig.B.43 Schematic of "padio"

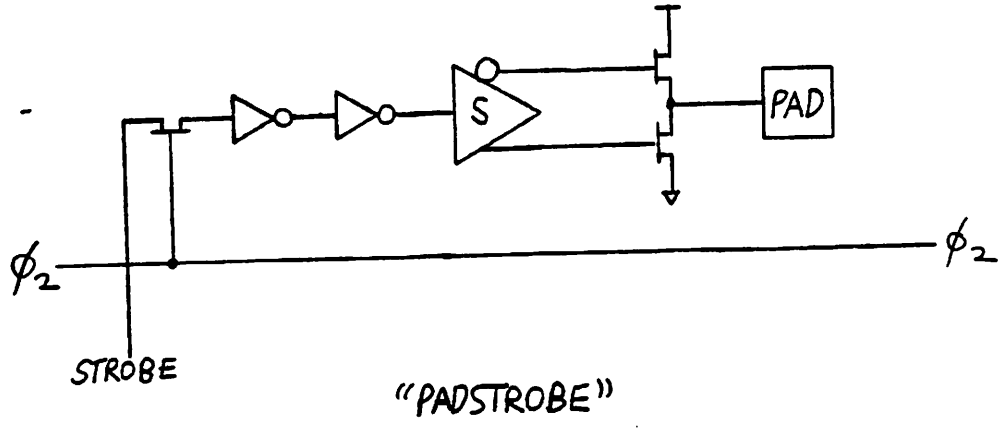


Fig.B.44 Schematic of "padstrobe"

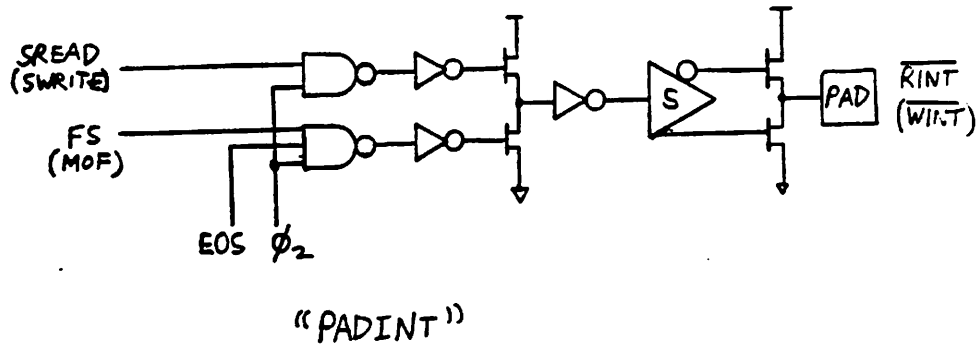


Fig.B.46 Schematic of "padint"

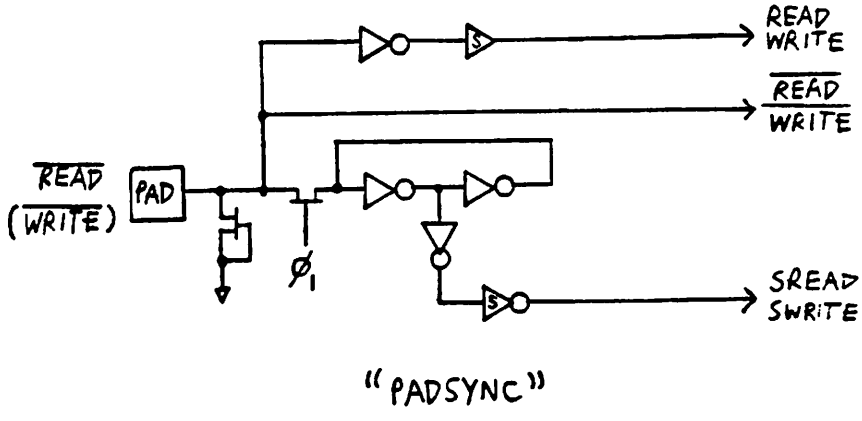


Fig.B.45 Schematic of "padsync"

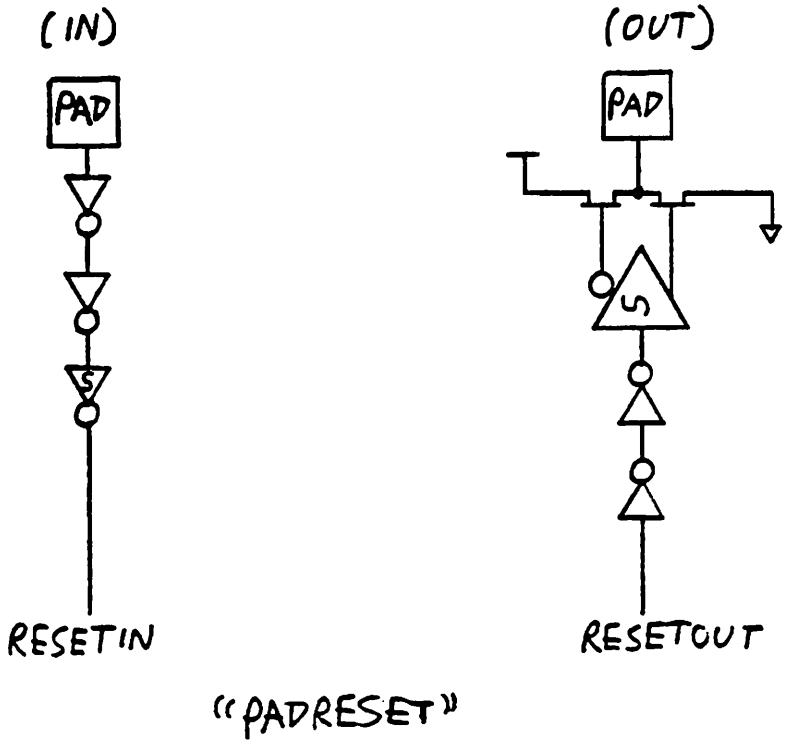


Fig.B.47 Schematic of "padreset"

If opposite, place groups (4) and (2) based on signal proximity.

It is assumed that pin-out is assigned by the software and may not be specified. Following the placement of the pad groups, three sets of permutable signals are identified:

The signal IO bus

The host IO bus

The strobes

For each set, signals may be assigned so as to to eliminate crossings within the wiring of that set.

The individual bonding pads come in the following varieties:

Tristate pad (Bidirectional, with an active high output enable)

Strobe pad (latches strobe on PHI2)

Synchronizing input pad (samples input on PHI1 into latch)

Interrupt output pad (with logic for setting/clearing interrupt)

Clock input pad (no drivers)

Ground pad for group (2)

Ground pad for group (4)

Vdd pad

IOC (I/O controller -- not a bonding pad, but part of group (2))

Cell names for these are *padio*,*padstrobe*,*padsync*,*padint*, *padclock*, *padgnd1*, *padgnd4*, *padvdd* and *iocontrol*. In addition, cell *padio.end* provides an "enable" terminal for a row of "padio" pads.

Circuitry for pads with active circuitry are given in Figs. B.43-45.

Each of the four groups has a single row of cells, providing terminals only along the bottom edge. GND and Vdd busses emerge along both left and right sides of the group.

Groups (1) and (3) contain the following cells:

a number of instances of padio

padio.end

Group (4) contains:

a number of instances of padstrobe

padgnd4

zero or more instances of padio

padio.end (if at least one instance of padio)

Note that "padgnd4" contains a terminal that connects phi2 to the strobe pads.

Group (2) is always fixed, consisting of the single cell "padgroup2". The cells contained in padgroup2 are kept available for future designs with more elaborate pad arrangements.

An additional pair of pads, in the cell *padreset*, may be attached to the left edge of padgroup4 and used for the "external sync" option described in Appendix C. Figs. B.47 is the schematic for *padreset*.

B.12 Descriptor File

The following descriptor file lists all the cells, and their terminal positions, for the NMOS cell library in the format described in Section 4.2 of Chapter 4. (A few

keywords not described in Section 4.2 are used in this version of the file.)

```
cell ram.0  17 19
cell ram.1  17 19
cell ram.c  17 19
cell rambtm.0  30 25
cell rambtm.1  17 25
cell rambtm.2  74 25
cell rambtm.3  14 25
cell rambtm.4  0 25
cell ramlft  30 19
cell ramsel  74 19
cell ramtop.0  30 12
cell ramtop.1  17 12
top bitline 2 12
cell ramtop.2 74 12
top write1* 9 12
top read1* 16 12
top write2* 30 12
top read2* 37 12
cell ramtop.3 14 12
cell ramtop.4 0 12
cell ramdec.0 14 19
cell ramdec.1 14 19
cell ramrgt 0 19
cell ramdrv 51 19
right address 51 17
cell ramcon.0 0 19
cell ramcon.1 0 19
cell ramcon.2 0 19
cell ramcon.3 0 19
cell ramcon.4 0 19
cell ramcon.5 0 19
cell ramcon.6 0 19
cell rombtm.1 23 23
cell rombtm.2 16 23
cell rombtm.3 16 23
cell rombtm.4 16 23
bottom inbtm* 6 0
bottom inbtm 14 0
cell rombtm.g 8 23
cell rombtm.5 60 23
cell rommid.1 23 8
cell rommid.g 8 8
cell rommid.2 16 8
cell rommid.3 16 8
cell rommid.4 16 8
cell rommid.5 53 8
cell dummy.1 23 8
cell dummy.g 8 8
cell dummy.2 8 8
cell dummy.3 16 8
cell dummy.4 16 8
cell dummy.5 53 8
cell romword.1 23 7
cell plac.g 8 7
cell placm.g 8 7
cell romword.2 16 7
cell romword.3 53 7
cell romwordm.1 23 7
cell romwordm.2 16 7
cell romwordm.3 53 7
cell romtop.1 23 112
cell romtop.g 8 112
cell romtop.2 16 112
top out1 2 112
top out2 10 112
cell romtop.3 16 112
cell romtop.4 16 112
top intop* 6 112
top intop 14 112
cell romtop.5 53 112
cell fsmdrv 16 112
top fsmin 4 112
cell plac.0 8 7
cell plac.1 8 7
cell placm.0 8 7
cell placm.1 8 7
cell pladec.0 16 7
cell pladec.1 16 7
cell pladec.x 16 7
cell counter 44 88
cell counter.0 44 88
cell counter.1 44 88
cell counter.gnd 30 88
cell pc.1 44 26
top out* 17 26
top out 25 26
cell spc.1 44 26
bottom out* 17 0
bottom out 25 0
cell aauout.gnd 30 22
cell aauout 44 22
bottom out 2 0
```

cell aauadd.gnd 30 117
cell aauadd.e 44 117
cell aauadd.o 44 117
cell aauadd.ctl 66 117
right index 66 92
cell aaudec.gnd 30 17
cell aaudec.0 44 17
cell aaudec.1 44 17
cell aaudec.x 44 17
cell ydec.ctl 66 17
cell aaudec.ctl 66 17
right test 66 10
cell aauinvert.gnd 30 10
cell aauinvert 44 10
cell aauinvert.ctl 66 10
cell aauconnect.gnd 30 8
cell aauconnect 44 8
cell aauconnect.ctl 66 8
cell aauin.gnd 30 7
cell aauin.x 44 7
top yinput 4 7
top input 41 7
cell aauin.y 44 7
top yinput 4 7
top input 41 7
cell aauin.ctl 66 7
cell ix.ctl 66 88
right eos 66 25
right inc 66 72
cell iy.ctl 66 88
right yclock 66 73
cell aauout.ctl 66 22
cell pc.0 30 26
cell pc.2 66 26
cell pc.ctl 66 88
right reset 66 75
right eos 66 44
cell spc.0 30 26
cell spc.2 66 26
cell spc.3 30 21
cell spc.4 44 21
cell spc.5 44 21
cell spc.6 66 21
right ret 66 2
cell spc.ctl 66 88
right jsr 66 75
cell hwrite.gnd 30 50
cell hwrite 44 50
bottom hostbus 41 0
cell hwrite.ctl 146 50
right eos 146 23
right mof 146 16
right swrite 146 9
right write* 146 2
cell hinput.gnd 30 72
cell hinput 44 72
cell hinput.ctl 146 72
right read 146 113
right radv 146 67
right rclr1 146 39
right rclr2 146 32
right wnp 146 22
right wnp+1s 146 4
cell fifo2.gnd 30 47
cell fifo2.c 44 47
cell fifo2.ctl 146 47
cell ps2.gnd 30 41
cell ps2 44 41
cell ps2.ctl 146 41
right dataout 146 23
cell sp2.gnd 30 44
cell sp2 44 44
cell sp2.ctl 146 44
right datain 146 6
cell fifo.gnd 30 32
cell fifo.c 44 32
cell fifo.ctl 146 32
cell hostout.gnd 30 42
cell hostout 44 42
cell hostout.ctl 146 42
right write1 146 14
right write2 146 22
right wclr1 146 29
right wclr2 146 37
right h 146 54
cell hread.gnd 30 47
cell hread 44 47
bottom hostbus 41 0
cell hread.ctl 146 47
right read 146 3
right sread 146 19
right eos 146 27
right fs 146 35
cell au 88 430
bottom bitline0 42 0
bottom bitline1 86 0
cell au.gnd 30 430
cell procio.gnd 30 38

```
cell procio 44 38
cell procio.ctl 137 38
cell sp 44 64
cell sp.0 44 64
cell sp.gnd 30 64
cell sp.ctl 137 64
right datain 137 59
right enable 137 5
cell splatch.ctl 137 64
right datain 137 59
right write 137 26
right enable 137 5
cell ps.gnd 30 59
cell ps 44 59
cell ps.0 44 59
cell pslatch.ctl 137 59
right dataout 137 55
right load 137 47
right write 137 21
cell ps.ctl 137 59
right dataout 137 55
right load 137 47
cell pario.gnd 30 56
cell pario 44 56
top pario 2 56
cell pario.0 44 56
top pario 2 56
cell pario.ctl 137 56
right parin 137 10
right padenable 137 22
cell au.ctl 225 430
bottom bitline0 42 0
bottom bitline1 86 0
bottom phi1* 99 0
bottom r* 106 0
bottom w* 113 0
bottom phi2 160 0
right wc 225 13
right cc 225 20
right w 225 27
right wen 225 41
right writelatch 225 59
right r 225 66
right xmitmor* 225 86
right shift* 225 93
right s0 225 100
right s1 225 107
right s2 225 114
right 1 225 151
right inv1 225 201
right inv2 225 217
right memb* 225 239
right zerob* 225 247
right coef1 225 254
right zeroa1 225 261
right coef2 225 268
right zeroa2 225 275
right zeroa3 225 282
right h 225 322
right quot 225 333
right sign 225 370
right aip 225 380
right accb* 225 400
right xmitacc* 225 419
cell padio 222 231
bottom bit 4 0
cell padio.end 10 231
bottom enable 6 0
cell padstrobe 212 231
bottom strobe 22 0
cell padgnd4 172 231
clock phi2 6 0
power gnd 103 0
cell padgroup2 1790 275
bottom read 4 0
bottom sread 20 0
bottom write 182 0
bottom swrite 198 0
bottom write* 230 0
bottom fs 980 0
bottom ls 1002 0
bottom wnp 1026 0
bottom rnp 1049 0
bottom wnp+ls 1072 0
bottom eos 1254 0
bottom eof 1261 0
bottom mof 1268 0
clock phi1 1548 0
clock phi2 1720 0
power gnd 891 0
power vdd 1377 0
cell padreset 388 231
bottom resetin 22 0
bottom resetout 198 0
```

Appendix C – Design File Description

This Appendix is excerpted from the reference manual for the silicon compiler [35].

C.1 Introduction

The design file gives a full description of a signal processing IC and serves as a single input to the emulator and silicon compiler, so that both design tools are consistent. A special purpose language has been developed. This language, which describes the system at an intermediate level, can be kept rather simple, due to the restriction to a well defined processor and interprocessor architecture. (The design file will be generated by a higher level compiler in a future phase.)

For the formal definition of the syntax of the design file, the syntax is described by the following notation: semantic constructs are denoted by English words between the angular brackets < and >. These words are suggestive of the nature or meaning of the construct. Production rules use ::= to define a construct as an expression or a combination of constructs; curly brackets { } indicate repetition any number of times including zero; square brackets [ ] indicate optional factors (i.e. zero or one repetition); parentheses ( ) are used for grouping. The vertical bar | is used for the or-ing of constructs.

The design file contains a detailed description of the basic blocks of the signal processing IC : the processors, the host I/O and signal I/O, the interprocessor communications. This determines the general format of the design file:

<design-file> ::= <globals> <I/O> (<proc> { <proc>}) [<constraints>]

In the next sections, the syntax and the contents of each of these blocks will be discussed in detail. This is preceded by a discussion of the number representation used in our architecture and its implications on the arithmetic operations.

C.2 Number representation and arithmetic operations

The two's-complement number representation has proven to be the most flexible way to handle negative numbers in a binary number system and is therefore used for all arithmetic operations in our structure. In this section, details of this representation are given. The way in which two's-complement multiplications and divisions are performed is described.

In the two's-complement number notation, the representation of a positive number is identical with its representation in an unsigned binary format. If a number, x, is negative, the two's-complement of x is given by Eq. (1):

$$Two's-complement\,(x) = 2^n - |x| \quad (x < 0) \quad (1)$$

One advantage of the two's-complement approach is that two's-complement addition is the same as the addition of two positive arguments.

Variable-variable multiplications

Suppose now that we want to multiply two two's-complement numbers x and y and that y is a fractional number ($-1 \leq y < 1$) with word-length n. It can be seen that the y can be expressed in terms of its two's-complement notation as shown in (2), where y_i is the i-th bit of the 2's-comp. representation.

$$val(y) = -y_0 + \sum_{i=1}^{n-1} \frac{y_i}{2^i} \quad (2)$$

This form allows a simple procedure for multiplication.

$$xy = -xy_0 + \sum_{i=1}^{n-1} \frac{x}{2^i} y_i \quad (3)$$

Equation (3) suggests a method of multiplying two two's-complement numbers in a parallel-serial fashion. Start with either 0 (when y is positive) or -x (y negative). Add x, shifted over i positions, if the i-th bit if the coefficient y is one. Note that the bits of the coefficient are needed serially on succesive cycles, MSB first, in order to control the addition of the shifted values. (Examples of how this procedure is microcoded are given in Section C.7.)

Example 1 : multiplication of 011000 (3/4) with 0011 (3/8)

```
000000      /* sign bit of coefficient is zero */
0000000     /* first significant bit zero */
00011000    /* bit 2 equals 1 : add x/4 */
000011000   /* lsb equals 1 : add x/8 */
---------
001001000   /* total = 9/32 */
```

Example 2 : multiplication of 011000 (3/4) with 1101 (-3/8)

```
101000      /* sign bit of coeffient is one . invert x */
0011000     /* first significant bit is one . add x/2  */
00000000    /* bit 2 equals 0 */
000011000   /* lsb equals 1 : add x/8 */
---------
110111000   /* total = -9/32 */
```

The result of multiplication has to be truncated to the wordlength of the processor, dropping the least significant bits (3 in the above examples). It is necessary to take this effect into account when determining the required number of bits in the data word for a given application and given performance criteria.

Multiplication with a constant

A different approach is used when the coefficient is a high precision constant. The number of cycles needed can be reduced drastically by representing the coefficient in the canonical signed digit form. In this form, a constant is represented in the form (4), where the x_i have values of ± 1, and the exponents n_i are chosen so as to minimize j, the total number of digits. This representation typically has one third the numbers of the digits as does a binary representation, with no loss in precision.

$$csd(x) = \sum_{i=0}^{j} x_i .2^{n_i} \qquad (4)$$

Example : the constant 0110111 (55/64) can be represented as :

$$2^0-2^{-3}-2^{-6}$$

When multiplying by the above constant, only two additions are required. It should be kept in mind that is only very effective for constant coefficients, where the canonical form can be calculated before the processor's code is written.

Division of two variables

Divide operations are implemented using long division. To find the quotient N/D with |D| > |N|, one first has to determine the sign of quotient (for four quadrant divisions). The division operation itself is performed as follows : |N| is loaded in the accumulator. On succesive cycles |D|/2, |D|/4 ... is subtracted form the accumulator, the result being accumulated only if it is positive. If so, a one is added to the quotient-result, otherwise a 0 is added. In this way, a sign magnitude representation of the quotient is obtained in a bit-serial fashion. The result can be transformed to two's-complement notation using some additional hardware.

Section C.7 describes the procedure to implement this long division in microcode.

C.3 General syntax description

Each block has the same basic syntax :

<block> ::= <keyword [parameters] CR> begin <statement> end

with

<statement> ::= <dataline:> { <dataline:>}

Note that only the semicolon is considered as a statement separator. Blanks, tabs and carriage returns are ignored. The only exception is the .keyword line, which has to be terminated by a carriage return (CR). Certain keywords may be followed by parameters.

Comments can be included as follows :

<comment> ::= /* put comment here */

These comments can be inserted everywhere in the design file.

C.4 The global-block

A global is a variable which is shared between different processors or which is shared with the outside world (host- and signal-I/O). All these variables are processed in bit-serial form and are buffered in temporary latches. The only exceptions are signal-I/O variables, which are parallel and unbuffered. It must be noted that the bit-serial transfer causes a delay of a number of clock cycles, on the order of the word length of the global variable.

Global variables which are used for host I/O may be declared as arrays. Input or output of array variables is indexed automatically with the ix-register if the reference to the global is in a subprogram, and with the iy-register if the reference is in the main program. Global array variables are stored in FIFO structures in the

Host Interface. Globals used for signal I/O or interprocessor communication are never arrays.

Globals may be indexed by the iy-register only in counter mode and not in pointer mode. These two modes of using the iy-register are described below.

The syntax of a global block :

<global-block> ::= .global CR begin <block-statement> end

<block-statement> ::= <data_line:> {<data_line:>}

<data_line> ::= <variable_decl> {,<variable_decl>} : [<justification>]

<variable_decl> ::= <name [dimension] <word_length>>

<justification> ::= left_justified | right_justified

("name" is a user-supplied variable name.)

Dimension declares the dimension of an array variable. If no dimension is specified, a scalar variable is assumed. Word_length defines the wordlength of the global. This denotes the number of bits which is transferred over the serial line. It defines the number of lines in the case of parallel transfer.

Justification denotes how the words are truncated if the wordlength of the processor and the global are different : left_justified preserves the most significant bits or pads the word with zero's at the right side (if the word has to be lengthened). Right_justified selects the least significant bits (and drops the sign-bit). This is e.g. important when positive counter values have to be transferred. left_justified has been selected as the default value.

IMPORTANT REMARK : The definition of a global results in the generation of

extra hardware at the transmitter and the receiver side. Excessive use of globals results in a untolerable growth of the processor-dimensions. The user should be careful and should try to keep the number of global definitions to a strict minimum in order to obtain an area-efficient design.

example :

```
.global /* interprocessor and I/O communications */
begin
   ka[10]<8>, ks[10]<8>; /* array variables with dimension 10 and
            wordlength 8 - left_justified        */
   pitch_in<8>, pitch_out<8> : right_justified;
     /* scalar variables with wordlength 8 -
       right_justified                    */
end
```

C.5 The I/O-block

This block defines which of the global variables are selected as I/O-variables.

Syntax:

<io-block> ::= .io <<host_word_length>>CR begin <block-statement> end

<block-statement> ::= <data_line:> {<data_line:>}

<data_line> ::= <file> : <variable> {,<variable>} : <io_specification>

<io_specification> ::= host_in | host_out | signal_in | signal_out

The host_word_length is specified on the same line as the .io-keyword and determines the size of the parallel bus, connecting the host-I/O unit and the host-processor. This equals normallly the wordlength of the host processor itself (8 or 16 bits) and by default is set to 8.

The file definition is only intended for emulation-purposes and determines where the input data can be found or the output data has to be written. The variable specification specifies which global variable (already defined in .global) is connected to a I/O-unit. The io_specification determines the type of I/O . signal-i/o is transferred through the parallel unbuffered bus, while host-I/O communicates with the host-processor through the FIFO-buffered host-I/O section.

example :

```
.io <16> /* 16 bit host interface */
begin
   host_in.d : ka.ks : host_in;
   host_out.d : pitch : host_out;
   speech_in.d : speech_in : signal_in;
end
```

C.6 The processor block

A processor can be considered as an assembly of different macrocells. Some of these cells have to be defined by the user (e.g. data-memory, finite state machine, arithmetic unit), others are partially or completely assembled by the compiler interpreting the user defined microcode (e.g. control section, I/O-units, address-arithmetic). This leads to following general format :

<proc> ::= .processor : <name <word_length>> CR begin <sub_blk> end

<sub_blk>::=<locals> <constants> <fsm> <main_program> <sub_program>

In the .processor line, a name is given to the processor and the wordlength of the arithmetic unit and the data memory is determined. Note that this wordlength is sufficient to assemble the complete aritmetic unit.

The syntax and meaning of the different subblocks is demonstrated in the following sections. Note that each of these blocks is optional. However either a main_program or a sub_program should be provided.

The local-block

In the hardware description, it has already been mentioned that the data memory of each processor can be a mixture of RAM and ROM. The RAM-memory locations are denoted as locals, the ROM-words are defined as constants (see constant-block).

Syntax :

<locals> ::= .localCR begin <data_line:> { <data_line:>} end

<data_line> ::= <name[dimension]> {.<name[dimension]>}

If no dimension is specified, the variable is considered to be scalar, otherwise an array of length [dimension] is reserved.

The constant-block

For definition, see local-block.

Syntax :

<constants> ::= .constantCR begin <data_line:> { <data_line:>} end

<data_line> ::= <name[dimension] = value {,value}>

Each data-line contains the definition and the initialization of only one constant-type. Arrays of constants of length [dimension] can be defined. In that case, the number of values has to equal the dimension of the array.

Example :

```
.processor : pitch <18>
/* implements the Gold pitchtacker algorithm */

begin
.local
begin
thresh[6], ppc[6], pp[7], lpp[6], signal[6];
ls, lp, lv, score, topscore, pitch, winner;
end

.constant
begin
TWO = 2;
BLANK = 24;      /* definition of blanking interval */
```

```
VOICED = 9;      /* speech if unvoiced if score < VOICED */
WINDOW1 = 8;     /* windows to compare pitches */
WINDOW2 = -14;
end

... definition of fsm and microcode

end
/* end of pitch_tracker definition */
```

The definitions in the above example make provisions for a data-memory of 43 words (38 RAM + 5 ROM).

The finite state machine

A limited form of decision-making is feasible with the definition of a finite state machine . This machine controls a conditional code-bit (cc), which in its turn governs the write operation. This results in a conditional write-instruction. The finite state machine operates in parallel with the processor.

The user defines the finite state machine completely. The state variables may be given arbitrary names, except for the following following reserved names : cc, mof and eof (cfr. host_io). Note that a conditional write operation is only possible when a cc-output has been defined and that only one processor may define the mof and eof bits.

Following signals can be used as input to the fsm : the states itself, the sign bit of the accumulator (TRUE if negative), the individual bits of the ix- and the iy-registers, and expressions of the form "ix=constant" or "iy=constant".

Syntax :

<fsm> ::= .fsm CR begin <command_def;> { <command_def;>} end

<command_def> ::= <cmd_name> : <equation> {, <equation>}

<equation> ::= <state> = <expression>

<expression> :== combination of <booleans> and <operands>

<booleans> :== <state>, sign , ix[c], iy[c], ix<i>, iy<i>

<operands> :== &(AND), |(OR) and !(NOT)

The <cmd_name> is used to reference to a specified fsm-command in the micro-code. An expression is evaluated from left to right with normal operator precedence : ! has the highest priority, and | the lowest. Parentheses can be used to change the precedence. "ix<i>" denotes the i-th bit of the ix-register, with ix<0> the least significant bit. "iy<i>" has a similar interpretation. "ix[c]" denotes an equality comparison between the ix-register and a constant. This is used, for example, to execute an operation only during the final iteration of a subprogram. "iy[c]" is used similarly.

Example :

```
.fsm /* finite state machine description */
begin
   SET : cc = sign;
  /* set condition code if acc >= 0 */
   AND_MINUS : cc = cc & !sign;
  /* set cc if acc < 0 and cc = TRUE */
   APV : cc = cc & (!ix<0>&!slp&lsp | ix<1>&slp&!lsp);
```

```
  VPE : cc = iy[0];    /* set cc if iy = 0 */
  SIP : cc = !slp & lsp;
 /* set cc if peak */
  SIV : cc = slp & !lsp;
 /* set cc if valley */
  SSL : lsp = slp, slp = sign;
 /* set slp (slope) if acc >= 0,
    set lsp (last_slope) to slp  */
end
```

C.7 The Microcode-block : Main Program and Subprogram

General block-syntax

The basic structure of the control-sequencer is quite strict. A processor starts a new sample interval with the execution of a main program, followed by a loop of x_mod subprograms. Note that all processors are synchronized in that they start a new sample at the same time. This means that a processor with a shorter program has to wait (execute nop's) until all other processors have finished their program. Both the main-program and the sub-program are optional.

The compiler infers the structure of the control sequencer from the microcode description. It counts the number of instructions in main- & sub-program and computes the number of cycles in a sample interval. This is done for all processors and the maximum value is taken as the modulus of the system's main program counter. The subprograms of two processors can be synchronized using the sync and the couple options (cfr. constraints).

Note that the microcode contains all the information needed for the generation

and assignment of the I/O-units and the address arithmetic unit. The compiler scans the microcode to check for the use of indexed addressing and for different kinds of I/O.

Syntax :

<main_program> ::= .main_pr <y_mod>CR begin <microcode> end

<sub_program> ::= .sub_pr <x_mod>CR begin <microcode> end

<microcode> ::= <simultaneous_instr;> { <simultaneous_instr;>}

<simultaneous_instr> ::= <instruction> {, <instruction>}

"y_mod" and "x_mod" determine respectively the modulus of the iy- and ix-index counters. As already stated before, the ix-counter counts the number of iterations of the subprogram and is incremented at the begin of a new iteration (ix = -1 in the main program). The iy-counter (in counter mode addressing) is incremented at the start of a new sample-cycle and is used for e.g. decimation. These registers (or counters) are basically used in the indexed read- and write operations (cfr. microcode definition). The default values of x_mod and y_mod are both 1.

If the y_mod field of the .main_pr line is "*", pointer mode addressing is implied. In this event, the iy-register may be loaded from the processor mbus. The new value is available in the iy register on the second instruction following the instruction in which the assignment to iy is performed.

Each microcode line consists of a number of simultaneously executed pipeline instructions. The emulator checks the consistency of these instructions. The lay-out generator transforms the assembly level description into binary used for ROM programming.

The assembler syntax

The set of available microcode instructions is split into groups of similar instructions. Members of the same group are mutually exclusive, while instructions of different groups can be executed simultaneously. There are the instruction groups:

memory
sor
acc (accumulator)
mbus
mir (memory input register)
output
aip (accumulate if positive)
coef (coefficient)
quot (quotient)

One may perform simultaneously a mor, sor, acc, mbus, mir, output, aip, coef and quot- instruction.

The Memory Instructions

All these instructions affect the mor (memory input resister). "loc_add" denotes the address assigned the local variable "local". The immediate index "ind" is used to address the different elements of an array variable. "ind" can be omitted when pointing to the first element of an array or in the case of scalar variables. The presence of a finite state machine defining 'cc' is assumed when

invoking a conditional write-operation. In the case of indexed addressing, the contents of the ix- or iy-register (defined in Part I section 3.6) is added to the actual address.

Note that in the syntax definitions, ":=" denotes a storage action (assignment of a value to a storage location).

instruction	action
default	mor := -1
r(local[ind])	mor := mem(loc_add + ind)
rx(local[ind])	mor := mem(loc_add + ind + ix)
ry(local[ind])	mor := mem(loc_add + ind + iy)
w(local[ind])	mem(loc_add + ind) := mir; mor := ~ mir
wx(local[ind])	mem(loc_add + ind + ix) := mir; mor := ~ mir
wy(local[ind])	mem(loc_add + ind + iy) := mir; mor := ~ mir
wc(local[ind])	if (cc) { mem(loc_add + ind) := mir, mor := ~ mir}
wxc(local[ind])	if (cc) { mem(loc_add + ind + ix) := mir, mor := ~ mir}
wyc(local[ind])	if (cc) { mem(loc_add + ind + iy) := mir, mor := ~ mir}
mor := ~ mir	mor := ~ mir (no memory action)

The sor instructions (shift output register)

Instructions affecting the sor-register.

instruction	action
sor := mor	unshifted load of mor-register
sor := sor	unshifted load of sor-register
sor := mor>>n	arithm. right shift of mor over n bits (0 <= n <= 7)
sor := sor>>n	arithm. right shift of sor over n bits (0 <= n <= 7)

The Accumulator Instructions

The syntax of an accumulator instruction is somewhat more complicated. The adder has two input busses (called abus and bbus). Both of these can represent a whole set of different actions. so that a large number of combinations is possible. To shorten the description of the instructions. we use a simplified syntax; the accumulator instructions can take one of the following forms :

acc := 'abus'

acc := 'bbus'

acc := 'abus' + 'bbus'

acc := 'bbus' + 'abus'

where 'abus' and 'bbus' represent respectively entries from the "abus"- and "bbus"-tables.

abus table:

instruction	action
0	abus = 0
sor	abus = sor
~ sor	abus = ~ sor (bit-inversion of sor)
|sor|	abus = |sor| (absolute value of sor)
~ |sor|	abus = ~ |sor| (bit inverted form of absolute val.)
coef.sor	if (coef == 1) {abus = sor} else {abus = 0}
coef.~ sor	if (coef == 1) {abus = ~ sor} else {abus = 0}

The order of the arguments in the coef-instructions is not significant: e.g. sor.coef is equivalent to coef.sor .

bbus table:

instruction	action
0	bbus = 0
mor	bbus = mor
acc	bbus = acc
mor&acc	bbus = mor&acc (bitwise AND-ing)
acc&mor	same as mor&acc

Most of these entries are clear from the above description. Some of them need however a more detailed specification.

— The output of the complementer is a 1's-complement (bitwise inversion) instead of a 2's-complement inversion. This results in an error of one least significant bit. E.g. the inversion of 0101 (5) yields 1010 (-6) instead of 1011 (-5). In digital filter implementations, this results in a small amount of additional noise. The effect must also be taken into account when using subtraction to perform a comparison.

-- The output of the adder is saturating. (cfr. hardware-description)

-- The coef-instruction (abus) is used for serial-parallel, variable-variable multiplications. One variable is loaded in the sor-register, while the second (coefficient) controls in a bit-serial way the output of the abus-multiplexer. In this way a shift-add multiplication is possible. More information can be found in the coef-instruction below.

The aip instruction

instruction	action
aip	accumulate if positive : load adder output in accumulator only if value is positive

This instruction makes the coding of a variable/variable division feasible (cfr. quotient-instruction). Besides the conditional accumulation, the aip-instruction adds a 1-bit to the quotient-result when positive, otherwise a 0-bit is added.

Note : the sign bit of the accumulator (used in the finite state machine) is checked BEFORE the aip-hardware. This makes a negative sign possible, even when a aip-instruction is executed.

The mbus instructions

These instructions assign the value of a certain register to the mbus. It is however important to know that the mbus does not provide any storage and is not a stage in the pipeline. Storage is provided in the mir-register or an external global.

instruction	action
mbus = acc	(default)
mbus = global	mbus = mor input command : loads external global

If the global (mbus = global) is defined as an array-variable, the iy- (ix-) register is used as the index-pointer to select the array-elements in the main- (sub-) program.

The latch-enable instruction (mir-register)

instruction	action
le	mir := mbus

The mir is a transparent latch. This means that the loaded value is immediately available and can be used in the same cycle for a write operation. Meanwhile the value is stored in the mir and remains there until the next "le" instruction. This in contrast with the other pipeline registers where the value is updated every cycle.

Output instructions

instruction	action
global := mbus	output of mbus to external global
iy := mbus	loads the mbus into the iy-index register

When the global (global := mbus) is an array global, the iy- (ix-) register is used as the index-pointer to select the array-elements in the main- (sub-) program. The new value for iy ('iy := mbus'-instruction) is available in the iy-register on the second instruction following the instruction in which the assignment to iy is performed.

Finite state machine instructions

instruction	action
<FSM-instruction>	adapt fsm-status (execute user-defined instruction)

With <FSM-instruction> the user-defined keyword (called <cmd_name> in previous syntax definitions) is executed.

The use of the finite state machine is illustrated with the following example : suppose that we want to find the largest of two numbers, stored in the local variables a and b, and that we want to store that number in the local c. We define a finite state machine with only one instruction SET- :

```
SET- : cc = sign; /* set cc if acc < 0 */
```

The following microprogram realizes the specified action :

```
r(a);     /* ld a in mor */
r(b), sor := mor, mbus = mor, le; /* ld a in sor and mir, ld b in mor */
w(c), sor := mor, acc := sor;  /* ld a in c and acc, ld b in sor */
acc := acc + ~ sor;   /* acc = a - b */
r(b), SET-;    /* lb b in mor, set cc if b > a */
wc(c), mbus = mor, le; /* ld b in c if cc ( b > a ) else keep a in c */
```

The coefficient instruction -- the multiplication operation

instruction	action
coef := global	feeds global into a P/S and starts shifting bits to the control of the abus-multiplexer

This instruction initiates a variable-variable multiplication. One of the variables, called the coefficient, is regarded as a fractional number (-1 <= coefficient < 1). The coefficient, which has to be defined as a global and is thus stored outside the processor's local memory, is parallel loaded into a P/S-converter and serially shifted into the control of the abus-multiplexer starting with the msb (sign-bit). When a 'one-bit' is presented, the contents of the sor-register is added to the accumulator. The contents of the accumulator is left unchanged in the case of a 'zero-bit'.

Basically, a variable-variable multiplication is performed in the following way. One variable is loaded in sor. On the next cycle, the coefficient is loaded into the P/S converter and the shifting is started. The msb (sign-bit) is presented to the multiplexer-control. The sor-value is shifted right repeatedly in the subsequent cycles, presenting a sequence of partial products to the adder input. The value of the coef-bit, present at that time, determines if this product is added to the accumulator value or not.

The following microcode fragment presents a typical 'multiply action. We want to multiply two variables a and b, stored as local variables in processor x. In a first step, b is transferred to c, defined as a left-justified global of wordlength 8 (the basic multiply action is going to take 8 cycles). Note that the transfer of the n-bit variable b to the 8-bit global c results in a truncation : only the 8 most significant bits of c are retained.

Next, the serial shift-in of c in the data-path is started. The rest of the algorithm is identical to the two's-complement multiplication, defined in section 2.

```
/* multiplication - example */
r(b);
r(a), mbus = mor, c := mbus; /* load b in the global c */
sor := mor;   /* load a in sor */
sor := sor>1, acc := coef.~ sor, coef = c;
/* sign-bit of c (== b) shifted in coef :
  if (coef == 1) load acc with ~ sor, else acc=0 */
sor := sor>1, acc :=  acc + coef.sor; /* bit 2 of c */
sor := sor>1, acc :=  acc + coef.sor; /* bit 3 of c */
sor := sor>1, acc :=  acc + coef.sor; /* ...       */
sor := sor>1, acc :=  acc + coef.sor;
sor := sor>1, acc :=  acc + coef.sor;
sor := sor>1, acc :=  acc + coef.sor;
acc := acc + coef.sor;              /* lsb of c   */
/* result of multiplication in accumulator */
```

Note that in a variable-variable multiply, the local data is always multiplied by a global coefficient. Thus, if the coefficient is stored locally it must first be assigned to a global as in the above example. Alternatively, the coefficient could originate from a different processor, where the global assignment is made.

A multiplication with a signed constant is however much simpler : Consider e.g. an 8-bit constant : 11100001 (represented in 2's complement representation). The following microcode fragment performs the multiplication of the mor-value with this constant.

```
/* multiplication with constant - example */
sor := mor;
```

```
sor := sor>1, acc := ~ sor; /* negative sign-bit, invert sor */
sor := sor>1, acc := acc + sor; /* 2nd 1-bit */
sor := sor>5, acc := acc + sor; /* 3rd 1-bit */
acc := acc + sor; /* 4th 1-bit = lsb */
/* result in accumulator */
```

The multiplication takes only four cycles here, the number of one-bits in the coefficient.

The quotient instruction – the division operation

instruction	action
global := quot	load result of division in global

Divide operations, by means of long division, are implemented using the "accumulate if positive" control option for the accumulator, combined with the global := quot instruction. This procedure results in a quotient, stored in a global variable (= fifo!) with a wordlength as defined in the global-definition. The bits of the quotient are obtained sequentially. The global := quot instruction closes the divide operation and stores the result in the global.

To find the quotient N/D with D >= 0 and |D| > |N|, the absolute value of N is loaded in the accumulator and -D/2 is loaded into the sor. The sign of N is automatically checked and routed into the quotient (= sign of the sor-register two cycles before the first aip). On successive cycles D/2, D/4, ... is subtracted from the accumulator, the result being accumulated only if it is positive. A one (zero) bit is routed into the quotient-S/P if the result is positive (negative). This results in a sign-magnitude representation, which is converted automatically into a 2's

comp. representation for the global.

Example : (see section 2 for numerical example)

```
/* division - example . ka is defined as 8-bit global*/
r(N);
r(-D), sor := mor; /* N in sor - sign bit is tested */
sor := mor>1, acc = |sor|; /* load absol. val. of N in acc */
/* start division */
sor := sor>1, acc := sor + acc, aip; /*sign bit routed in quot*/
sor := sor>1, acc := sor + acc, aip; /*bit 1 in quot */
sor := sor>1, acc := sor + acc, aip;
sor := sor>1, acc := sor + acc, aip;
sor := sor>1, acc := sor + acc, aip;
sor := sor>1, acc := sor + acc, aip;
acc := sor + acc, aip;
ka := quot; /*division finished, quot in global ka */
```

Note : this example assumes that the negative value of the denominator D is available. This simplifies the code and also allows for a considerable noise reduction. Instead, we could have used the acc := ~ sor + acc -command for the subtraction, but this would have resulted in a higher noise-level.

The nop instruction

instruction	action
nop	No Operations (executes the default commands)

Following instructions are executed by default (when not overwritten by another command) : mor := -1, sor := sor, acc := acc, mbus = acc, mir := mir. The major effect of these defaults is to refresh the values present in the pipeline registers.

C.8 Constraints

This block has been added to give the programmer a certain amount of control over the setup of the timing (and of the control sequencer) of the different processors. Other constraints than the ones listed can be considered on user's demand.

Syntax :

<constraints> ::= .constraints CR begin <data_block> end

<data_block> ::= <data_line:> { <data_line:>}

<data_line> ::= sync : <master> <slave> |

<couple : <master> <slave> |

external_sync

Where <master> and <slave> are processor-names.

The *sync* option is used when the subroutines of two processors are communicating and thus have to be synchronized : this is the case when the subroutine of the master processor sends data (in the form of globals) to the subroutine of the slave. In order to avoid a sample-delay in this data-transfer, the sync-constraint

has been implemented : the emulator (& compiler) checks the globals (and their delay), send from master to slave, and adjusts the timing of the slave so that the slave can access the data from the master in the same sample, and this without timing conflicts. It is clear that this constraint only makes sense when both processors have an equal number of subroutine iterations. Therefore, nops are added to the shortest subroutine.

The *couple* option includes the sync-option, but puts some more constraints on the subroutine alignment : in this case, the slave-subroutine not only receives data from the master, but also wants to send data back. This data has to be present at the master-side before it is accessed in the NEXT iteration of master- subroutine. This asks for a somewhat more complicated alignment of the subroutines and the inclusion of extra nop-instructions.

The *external_sync* option allows for a synchronisation of the processors to an external clock. To achieve this, the compiler routes the reset-control lines of the processors to the outside world (the reset-line initiates a new sample-cycle and starts the execution of the first instruction of the instruction-rom).

C.9 Conclusions

The different aspects of the design file description of a concurrent signal processing system have been discussed. This description serves as the input to the emulator and compiler development tools.

Appendix D – Design File Examples

D.1. A Simple Two-Processor Example

```
/* This simple example of a design file describes a two processor circuit. The cir-
cuit acts as follows. The first processor has a constant stored in local memory. It
reads out this constant, and sends it over to the second processor, using a global
variable. The second processor accepts an external signal input. This signal is
divided by two, added to the constant from the first processor, and used as a sig-
nal output. */

/* First three globals are declared, each eight bits. The three globals are the signal
input, signal output, and a global written by the first processor and read by the
second. */

begin
   sigin<8>, sigout<8>, temp<8>; end

/* The "io" section indicates that the sigin and sigout globals are to be input and
output on the signal data bus, respectively. There is no host interface. */

begin
  infile : sigin : signal_in;
  outfile : sigout : signal_out; end

/* The first processor is declared with symbolic name "p1" and data bus width of
8 bits. */

begin

/* One memory location is declared, a constant "fifty". There are no local vari-
ables. */

begin
  fifty=50; end

/* The main program reads the constant out of memory and assigns it to the glo-
bal "temp". */

begin
  r(fifty);
  mbus = mor, temp = mbus; end

end  /* p1 */

/* The second processor is declared with symbolic name "p2" and data bus width
```

```
of 8 bits. */

begin

/* Two local variables are declared in the second processor. */

begin temp1,temp2; end

/* Now comes the main program for the second processor */ /* (Neither processor
has a subprogram.) */

begin

  /* First assign the signal input to "temp1" and the
  constant from the other processor to "temp2". */

  mbus = sigin, le, w(temp1);
  mbus = temp, le, w(temp2);

  /* A few read instructions which are basically no-op's */

  r(temp1);
  r(temp1);
  r(temp1);
  r(temp1);

  /* "temp1" is read, divided by 2, added to "temp2".
  The accumulator is then assigned to global "sigout". */

  r(temp1);
  r(temp2), sor:=mor>1;
  sor:=mor, acc:=sor;
  acc:=acc+sor;
  sigout = mbus;

  /* another no-op */

  r(temp1); end

end /* p2 */


/* end of design file */
```

D.2. LPC Vocoder design file

```
/* LPC Vocoder Design File
/* There are three processors:
/* filter, correlator and pitch_tracker */


.global /* interprocessor and i/o communications */
begin
   ka[10]<8>; /* host output reflection coefs. */
   ks[10]<8>; /* host input reflection coefs. */

   /* interprocessor communications */

   ka_connect<8>, ka_shift<8>; /* ka-connection + multiplier */
   plus<16>, min<16>;
   low_pass<16>;
   square<13>;

   /* signal inputs and outputs */
   speech_in<16>, speech_out<16>, residu<16>, excitation<16>;

   /* host output and input residual energy parameter */

   energy_out<24>;
   energy_in<16>;

   /* host input and output pitch periods */

   pitch_in<8>, pitch_out<8> : right_justified;
end

.io <8> /* io-description */
begin
   /audio/jan/dsp/speech/host_in : energy_in, pitch_in, ks : host_in;
   host_out : energy_out, pitch_out, ka : host_out;
   /audio/jan/dsp/speech/speech_in : speech_in : signal_in;
   speech_out : speech_out : signal_out;
   residu_out : residu : signal_out;
end

/*************************************************************/

.processor : filter <16>

 /* main_program : low_pass filtering and     */
 /* pre-emphasis + deemphasis of signals  */
 /* subprogram : lattice filter : synthesis   */
 /* and analysis      */

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

```
begin
 .local
 begin
    a, b[11], c, d[11], temp;
    e, f, g, h;
    ka_intern[11];
 end

 .main_pr <1>
 begin

/* copy c to d[10], output residual, input speech (in e) */
/* deemphasis filter */

r(a);
r(h), mbus = mor; /* residu = mbus; */
r(c), sor := mor>3, acc := mor;
w(d[10]), mbus = mor, sor := mor>1, acc := acc + ~ sor, le;
r(e), acc := acc + sor;
w(h), sor := mor, speech_out = mbus, le;

/* preemphasis filter */
sor := sor, acc := ~ sor;
w(e), mbus = speech_in, sor := sor>2, acc := acc + ~ sor, le;
sor := mor, acc := acc + sor;
r(b[0]), sor := sor, acc := acc + ~ sor;
w(temp), acc := acc + ~ sor, mbus = mor, le ;
w(a), le;
w(b[0]);

/* Lowpass filters, first and second stages, input excitation */
r(f);
r(e), sor := mor>2, acc := mor;
w(c), sor := mor>2, acc := acc + ~ sor, mbus = excitation, le,
residu = mbus;
r(g), acc := acc + sor;
w(f), sor := mor>2, acc := mor, le;

/* input last ka -coefficient and write in 10 */
w(ka_intern[10]), mbus = ka_connect, sor := mor, acc := acc + ~ sor, le;
acc := acc + sor;
w(g), low_pass := mbus, le;

end

.sub_pr <10>

begin
```

```
/* implements analysis and synthesis lattice filter */

/* compute ~ |(a + b[i])| and ~ |(a ~ b[i])| */
r(temp);
r(a), sor := mor;
r(a), sor := sor, acc := sor + mor;
      mor := ~ mir, acc := ~ sor + mor, le;
mor := ~ mir, sor := mor, le;
rx(d[1]), sor := mor, acc := ~ |sor|;
r(c), sor := mor, acc := ~ |sor|, plus := mbus;

/* compute c = d(i+1)*ks + c */
sor := sor>1, acc := mor + coef.~ sor, min := mbus, coef = ks;  /* sign bit */
sor := sor>1, acc := acc + coef.sor;
sor := sor>1, acc := acc + coef.sor;
sor := sor>1, acc := acc + coef.sor;
rx(ka_intern[1]), sor := sor>1, acc := acc + coef.sor;
r(a), sor := sor>1, acc := acc + coef.sor, mbus = mor, ka_shift := mbus,
  ka := mbus; /* output ka to host_interface */
r(temp), mbus = mor, sor := sor>1, acc := acc + coef.sor, le;
mor := ~ mir, sor := mor, acc := acc + coef.sor;

/* compute a = a - b(i)*ka */
w(c), sor := sor>1, acc :=  mor + coef.~ sor, coef = ka_shift, le;
sor := sor>1, acc := acc + coef.sor;
sor := sor>1, acc := acc + coef.sor;
sor := sor>1, acc := acc + coef.sor;
sor := sor>1, acc := acc + coef.sor;
r(temp), sor := sor>1, acc := acc + coef.sor;
r(a), mbus = mor, sor := sor>1, acc := acc + coef.sor, le;
mor := ~ mir, sor := mor, acc := acc + coef.sor;

/* compute b(i+1) = b(i) - a*ka, update temp */
mor := ~ mir, sor := sor>1, acc := mor + coef.~ sor, coef = ka_shift, le;
w(a), mbus = mor, sor := sor>1, acc := acc + coef.sor,le;
rx(b[1]), sor := sor>1, acc := acc + coef.sor;
w(temp), mbus = mor, sor := sor>1, acc := acc + coef.sor, le;
sor := sor>1, acc := acc + coef.sor;
rx(d[1]), sor := sor>1, acc := acc + coef.sor;
r(c), mbus = mor, sor := sor>1, acc := acc + coef.sor, le;
mor := ~ mir, sor := mor, acc := acc + coef.sor;

/* compute d(i) = d(i+1) - ks*c */
mor := ~ mir, sor := sor>1, acc := mor + coef.~ sor, coef = ks, le;
wx(b[1]), mbus = mor, sor := sor>1, acc := acc + coef.sor, le;
sor := sor>1, acc := acc + coef.sor;
sor := sor>1, acc := acc + coef.sor;
sor := sor>1, acc := acc + coef.sor;
sor := sor>1, acc := acc + coef.sor;
```

```
sor := sor>1, acc := acc + coef.sor;

/* input new ka-coefficient (from correl) for cycle i - 1  */
acc := acc + coef.sor, mbus = ka_connect, wx(ka_intern), le;
mor := ~ mir, le;
wx(d), mbus = mor, le;

end

end

/*********************************************************************/

.processor : correlator<26>

/* main_program : frame_counter and waveform - generator */
/* sub_program : correlator                              */

/*********************************************************************/

begin

.local

begin
energ_temp;
c[10], d[10];
sample_counter;
pitch_counter, pitch, voiced, unvoiced;
random, gain;
end

.constant

begin
INCREMENT = 1;
MASK = 32767;
SIX = 6;
FRAME_LENGTH = -90;
MIDDLE_OF_FRAME = -70;
end

.fsm

begin
SET_IF_PLUS : cc = !sign;
SET_IF_MINUS : cc = sign;
MINUS_FIVE : cc = cc & !sign;
CC_OR_MINUS : cc = cc | sign;
```

```
EXOR : cc = !cc&!sign | cc&sign;
EOF : eof = !sign, mof_flg = mof | (mof_flg & !eof);
MOF : mof = !sign & !mof_flg;
.end

.main_pr <1>
begin

/* Here we have a little piece of code which counts
samples for the 11.25 millisecond frame interrups. */

r(sample_counter);
r(INCREMENT), sor := mor;
r(FRAME_LENGTH), sor := mor, acc := sor;
r(MIDDLE_OF_FRAME), sor := mor, acc := acc + sor;
r(pitch_counter), acc := acc + sor, aip;
w(sample_counter), sor := mor, le, acc : = acc + sor, EOF;

/* increment pitch_counter - compare with pitch */
/* determine input-zone ( <5, =5, >5)
voiced = gain/4 if <5
* voiced = ~ gain if 5
* voiced = 0 if >5 */

r(INCREMENT), sor := sor, MOF;
w (pitch) , acc := sor + mor, mbus = pitch_in, le;
w(pitch_counter), sor := mor, le;
sor := mor, acc := ~ sor + mor;
r(SIX), sor := sor, acc := 0, SET_IF_MINUS;
wc(pitch_counter), acc := sor + mor, le;
w(voiced), sor := mor, acc := acc, SET_IF_PLUS;
w(gain), mbus = energy_in, acc := sor + acc, le;
wc(voiced), sor := mor>2, mbus = mor, le, MINUS_FIVE;
r(random), acc := ~ sor;
wc(voiced), sor := mor, le;

/* generate random number ( for unvoiced case ) */
/* technique : exor lsb with lsb -1 and circular shift */
/* first, we have to check that the initial random number
is different from zero : this tends to block the
generator. */

      r(INCREMENT), acc := sor;
      acc := mor&acc;
      r(random), sor := mor, acc := acc, le;  /* mor = -1 */
      w(unvoiced), sor := mor>1 , acc := acc + sor;
      r(INCREMENT), acc := sor, SET_IF_PLUS; /* check lsb */
      acc := mor & acc, le;
      mor := ~ mir, sor := mor, acc := acc;
```

```
        sor := mor, acc := acc + sor;
        r (MASK), acc := ~ sor, EXOR; /* exor lsb & lsb -1 */
        sor := mor, acc := acc & mor;
        w(random), sor := sor, acc := acc + ~ sor, le; /* left shift */

/* check if random number is zero */
/* if yes, transform to largest negative number */

        r(unvoiced), acc := acc + sor, le;
        sor := mor, CC_OR_MINUS;

/* adjust unvoiced to gain/8 or -gain/8 */

        r(gain), acc := sor + mor;
        wc(random), sor := mor>3, SET_IF_MINUS;
        r(pitch), acc := sor;
        sor := mor, le;
        w (unvoiced), acc := sor + mor;

/* make voiced-unvoiced decision : if pitch = 0 :unvoiced */
/* output final waveform to excitation */

wc (unvoiced), mbus = mor, SET_IF_PLUS, le;
r (voiced);
wc (unvoiced), mbus = mor, le;
r (unvoiced);
excitation := mbus, mbus = mor, le;

/* send last energy_value to host_out */

r (energ_temp);
mbus = mor, energy_out := mbus;
end

.sub_pr <10>
begin
/* square plus-signal */
mor := ~ mir, mbus = plus, le;
sor := mor, mbus = mor, square := mbus;
sor := sor>1, acc := coef.~ sor, coef = square;
sor := sor>1, acc := acc + coef.sor;
sor := sor>1, acc := acc + coef.sor;
sor := sor>1, acc := acc + coef.sor;
sor := sor>1, acc := acc + coef.sor;
sor := sor>1, acc := acc + coef.sor;
sor := sor>1, acc := acc + coef.sor;
sor := sor>1, acc := acc + coef.sor;
sor := sor>1, acc := acc + coef.sor;
sor := sor>1, acc := acc + coef.sor;
```

```
sor := sor>1, acc :=  acc + coef.sor;
mor := ~ mir, sor := sor>1, acc := acc + coef.sor, mbus = min, le;
sor := mor, acc := acc + coef.sor, mbus = mor, square := mbus;

/* square minus-signal */
sor := sor>1, acc := coef.~ sor, coef = square, le;
sor := sor>1, acc :=  acc + coef.sor;
sor := sor>1, acc :=  acc + coef.sor;
sor := sor>1, acc :=  acc + coef.sor;
sor := sor>1, acc :=  acc + coef.sor;
sor := sor>1, acc :=  acc + coef.sor;
sor := sor>1, acc :=  acc + coef.sor;
sor := sor>1, acc :=  acc + coef.sor;
sor := sor>1, acc :=  acc + coef.sor;
sor := sor>1, acc :=  acc + coef.sor;
sor := sor>1, acc :=  acc + coef.sor;
mor := ~ mir, sor := sor>1, acc := acc + coef.sor;
rx(c), sor := mor, acc := acc + coef.sor;

/* start lowpass filtering of squared signals */
mor := ~ mir, sor := mor>6, acc := mor + ~ sor, le;
rx(d), sor := mor, acc := acc + ~ sor;
wx(c), sor := mor>6, acc := mor + ~ sor, le;
sor := mor, acc := acc + ~ sor;
wx(d), sor := sor>1, acc := sor + acc, le;

/* accumulator contains now D-C - calculate now (C-D)/2 & (C+D)/2 */
sor := mor>1, acc := ~ sor, le;
mor := ~ mir, acc := ~ sor + acc; /* (C+D)/2 in acc */
w (energ_temp), sor := mor>1, le;
sor := mor>1, acc := |sor|;

/* start division */
sor := sor>1, acc := sor + acc, aip; /*sign bit av. */
sor := sor>1, acc := sor + acc, aip;
sor := sor>1, acc := sor + acc, aip;
sor := sor>1, acc := sor + acc, aip;
sor := sor>1, acc := sor + acc, aip;
sor := sor>1, acc := sor + acc, aip;
acc := sor + acc, aip;
ka_connect = quot; /*division finished */
end

end

/*****************************************************/

.processor : pitch <18>
```

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

/* implements the GOLD -pitchtrack algorithm (can be improved) */

begin

.local

begin
thresh[6], ppc[6], pp[7], lpp[6], signal[6];
ls, lp, lv, score, topscore, pitch, winner;
end

.constant

begin
TWO = 2;
BLANK = 24;
VOICED = 9;
WINDOW1 = 8;
WINDOW2 = -14;
end

.fsm /* finite state machine description */

begin
SET : cc = !sign;
AND_MINUS : cc = cc & sign;
APV : cc = cc & (!ix<0>&!slp&lsp | ix<0>&slp&!lsp);
VPE : cc = iy[0];
SIP : cc = !slp & lsp;
SIV : cc = slp & !lsp;
SSL : lsp = slp, slp = !sign;
end

.main_pr <6>

begin
/* finish up calculation of previous time */
r (score);
r (topscore), sor := mor>1;
sor := mor>1, acc := sor;
r (score), acc := acc + ~ sor;
ry (pp), acc := mor, SET;
wc (topscore), acc := mor, le;
wc (winner), le;

/* if VPE (every six samples) compare topscore to constant VOICED.
   set pitch to either winner or zero and reset topscore        */
```

```
r (winner);
r (VOICED), acc := mor, VPE;
wc (pitch), sor := mor, le;
r (topscore), sor := sor, acc := 0;
wc (topscore), acc := ~ sor + mor, le;
r (signal), acc := 0, AND_MINUS;
wc (pitch), acc := mor, le, SIP;
wc (lp), le, SIV;
wc (lv);
w (ls);

/* beginning of new sample . Compare new signal to last and set SSL,
clear score */

w (signal), sor := mor>1, mbus = low_pass, acc := 0, le;
w (score), sor := mor>1, acc := sor, le;
r (signal), acc := acc + ~ sor;
mor := ~ mir, mbus = mor, SSL, le;

/* send pitch to output port . Form signal[1] through signal[5] . */

r (lv), sor := mor>1, mbus = mor, le;
w (signal[1]), sor := mor>1, acc := sor;
r (lp), sor :=mor>1, acc := sor + acc;
w (signal[3]), sor := mor>1, acc := ~ sor, le;
w (signal[2]), acc := sor + acc, mbus = mor, le;
r (pitch), le;
w (signal[5]), sor := mor>1;
w (signal[4]), mbus = mor, acc := sor, le;
pitch_out := mbus;
end

.sub_pr <6>

begin

/* increment pitch_counter by two and set condition code
if greater than BLANK . Conditionally decay threshold .
And condition code with peak-valley indicator. */

rx (thresh);
rx (ppc), mbus = mor, le;
r (TWO), sor := mor;
r (BLANK), sor := mor, acc := sor;
wx (thresh), sor := mor, acc := sor + acc;
wx (ppc), sor := mor, acc := ~ sor + acc, le;
r (lp), sor := sor>6, acc := ~ sor, SET;
rx (signal), sor := sor>1, acc := acc + sor, mbus = mor,le;
w (lp), sor := mor>1, acc := acc + sor;
```

```
wxc (thresh), sor := sor, acc := acc + ~ sor, le, APV;

/* And condition code with result of (signal > threshold)
comparison, Conditionally update thresh, lpp, ppc, pp */

rx(pp), acc := sor, AND_MINUS;
wxc (thresh), acc := mor, le;
wxc (lpp), le;
rx (ppc), acc := 0;
wxc (ppc), acc := mor, le;
wxc (pp), le;
r (lv);
w (lv), mbus = mor, le;

/* add contribution for this channel  to score of current cand.
do three window comparisons with current candidate and pp,
lpp and pp + lpp . In each case, increment score with 2 if true */

r (pp[0]);
w (pp[6]), mbus = mor, le;
ry (pp[1]);
rx (pp), sor := mor>1;
r (WINDOW1), sor := mor>1, acc := ~ sor;
r (WINDOW2), sor := mor>1, acc := acc + sor;
r (TWO), sor := mor>1, acc := acc + sor;
r (score), sor := mor, acc := acc + sor, SET;
ry (pp[1]), acc := mor + sor, AND_MINUS;
wc (score), sor := mor>1, le;
rx (lpp), acc := ~ sor;
r (WINDOW1), sor := mor>1, acc := acc;
r (WINDOW2), sor := mor>1, acc := acc + sor;
r (TWO), sor := mor>1, acc := acc + sor;
r (score), sor := mor, acc := acc + sor, SET;
ry (pp[1]), acc := mor + sor, AND_MINUS;
wc (score), sor := mor>1, le;
rx (lpp), acc := ~ sor;
rx (pp), sor := mor>1, acc := acc;
r (WINDOW1), sor := mor>1, acc := acc + sor;
r (WINDOW2), sor := mor>1, acc := acc + sor;
r (TWO), sor := mor>1, acc := acc + sor;
r (score), sor := mor, acc := acc + sor, SET;
acc := mor + sor, AND_MINUS;
wc (score), le;
end

end

.constraints
```

```
begin
couple : filter, correlator;
end

/* end vocoder description */
```

Appendix E – Companion Tape

A companion tape for this dissertaion, in UNIX *tar* format, is available. Contained on the tape are the following:

(1) A layout database for the LPC Vocoder circuit (Chapter 2). Also included are microcode and emulation files.

(2) The cell library (Chapter 3 and Appendix B)

(3) Design Files, Intermedite files, and resultant CIF files for the examples (Chapter 5 and Appendix D).

(4) Source code and documentation for the silicon compiler and the emulator (Chapter 4 and Appendix C).

The various directories on the tape contain files (usually named *ReadMe*) which describe their contents. Interested parties may contact:

Prof. Robert W. Brodersen
Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720

References

1. L. R. Rabiner, B. Gold, *Theory and Application of Digital Signal Processing*, Prentice-Hall, Englewood Cliffs N. J., 1975, Ch. 6.

2. *Ibid.*, Ch. 10

3. N. L. Daggett, "A Computer for Vocoder Pitch Extraction," Tech Note 1966-3, Lincoln Laboratories, Lexington, MA (1966).

4. L. R. Rabiner, B. Gold, *op. cit.*, Ch. 8.

5. Nelson Morgan, *Talking Chips*, McGraw-Hill, 1984, pp. 75-78.

6. Y. Kawakami, et. al., "A Single-Chip Signal Processor for Voiceband Applications", *International Solid State Circuits Conference Digest*, San Francisco, 1980, pp.40-41.

7. J. R. Boddie, et. al., "A Digital Signal Processor for Telecommunications Applications", *International Solid State Circuits Conference Digest*, San Francisco, 1980, pp. 44-45.

8. S. S. Magar, E. R. Caudel, A. W. Leigh, "A Microcomputer with Digital Signal Processing Capability", *International Solid State Circuits Conference Digest*, San Francisco, 1982, pp. 32-33.

9. Peter B. Denyer, "An Introduction to Bit Serial Signal Processors for Signal

Processing". University of Bristol, U. K., July 1982.

10. Neil Bergman, "A Case Study of the F.I.R.S.T. Silicon Compiler", in R. Bryant, ed., *Proc., Third CalTech Conference on Very Large Scale Integration*, Computer Science Press, Rockville, MD, 1983.

11. S. Pope, J. Rabaey, R. W. Brodersen, "Automated Design of Signal Processors Using Macrocells", *Proc., Conf. on VLSI Signal Processing*, University of Southern California, Los Angeles, Nov. 1984.

12. Richard F. Lyon, "A Bit-Serial VLSI Architecture Methodology for Signal Processing", in J. P. Gray, ed., *VLSI 81* Academic Press, 1981.

13. R. Fellman, P. Hurst, R. W. Brodersen, "Switched Capacitor Circuits for Adaptive Filtering and Autocorrelation", *International Solid State Circuits Conference Digest*, New York, Feb. 1983.

14. Rabiner, L. R., Schafer, R. W., *Digital Processing of Speech Signals*, *Prentice-Hall*, Englewood Cliffs, N.J., 1978, Chapter 8.

15. *Ibid*, pp. 441-444.

16. T. E. Tremain, "The Government Standard Linear Predictive Coding Algorithm LPC-10", *Speech Technology*, pp. 40-49, Apr. 1982.

17. J. Burg, "Maximum Entropy Spectral Analysis", Ph.D. Dissertation, Stanford University, Stanford, CA, May 1975.

18. Kang, G. S., "Application of Linear Predictive Coding to a narrowband voice digitizer", *Naval Research Laboratory Report 7779*, Washington, D.C., 1974.

19. B. Gold, L. R. Rabiner, "Parallel Processing Techniques for Estimating Pitch Periods of Speech in the Time Domain", *J. Acoustical Society of America*, V. 34, No. 7, pp. 916-921, 1962.

20. L. R. Rabiner, R. W. Schafer, *Op Cit*, pp. 452-453.

21. D. Johansen, "Bristle Blocks: A Silicon Compiler", *Proc., 16th Design Automation Conference*, San Diego, June 1979.

22. Hwang, K., *Computer Arithmetic*, Wiley, New York, 1979, p. 149.

23. Bohm, C., and Jacobini, G., "Flow-diagrams, Turing Machines, and languages with only two formulation rules", Comm. ACM 9,5, May 1976, pp. 366-371.

24. K. H. Keller, A. R. Newton, "KIC2: A Low-Cost Interactive Editor for Integrated Circuit Design", *Digest of Papers, IEEE CompCon 82 Conference*, San Francisco, Feb. 1982, pp. 302-304.

25. Ritchie, D. M., Thompson, K., "The UNIX Time-Sharing System", *Bell Systems Technical Journal*, 57(6), pp. 1905-1929, 1978.

26. G. C. Billingsley, "Program Reference for KIC", Memorandum No. M83/62, Electronics Research Laboratory, University of California, Berkeley, CA, October 1983.

27. R. N. Mayo, J. K. Ousterhout, "Pictures with Parentheses: Combining Graphics and Procedures in a VLSI Layout Tool", *Proc., 20th Design Automation Conference*, June 1983.

28. Dan Fitzpatrick, private communications.

29. T. Yoshimura, E. S. Kuh, "Efficient Algorithms for Channel Routing", IEEE Transactions on Computer Aided Design, V. CAD-1, Jan. 1982, pp. 25-35.

30. C. A. Mead, Lynn Conway, *Introduction to VLSI Systems*, Addison-Weseley, Reading, Mass., 1980.

31. Mats Torkelson, private communication.

32. Jan Rabaey, private communication.

33. Jeremy Tseng, private communication.

34. D. Johansen, *op. cit*.

35. Jan Rabaey, "LAGER -- An Automated Layout Generating System for Digital Signal Processing Circuits -- User Manual Version 1.3", Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, CA, December 1984.

36. W. L. Abbott, "Design of a 300-Baud FSK Modem using Customized Digital Signal Processors", Memorandum No. M84/83, Electronic Research Laboratory,

University of California, Berkeley, CA, August 1984.