DESIGN OF A FINITE IMPULSE RESPONSE FILTER INTEGRATED CIRCUIT FOR PULSAR SIGNAL RECOVERY

by

Amar A. Kapadia

Memorandum No. UCB/ERL M91/74

3 September 1991

DESIGN OF A FINITE IMPULSE RESPONSE FILTER INTEGRATED CIRCUIT FOR PULSAR SIGNAL RECOVERY

by

Amar A. Kapadia

Memorandum No. UCB/ERL M91/74

3 September 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

DESIGN OF A FINITE IMPULSE RESPONSE FILTER INTEGRATED CIRCUIT FOR PULSAR SIGNAL RECOVERY

by

Amar A. Kapadia

Memorandum No. UCB/ERL M91/74

3 September 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Table of Contents

List of Figures

List of Tables

Introduction 1

1.1 A Perspective

Radio astronomy signal processing is an active area which has potential for numerous new applications of integrated circuits. For instance, pulsar signal processing applications typically require racks of boards to implement these systems. The Coherent Dispersion Removal Processor (CDRP) being developed by the Center for Particle Astrophysics is one such system. A main component of this system is a 1024 tap finite impulse response (FIR) filter integrated circuit which is presented in this report.

The FIR filter is intended for data rates from 1 kHz to 4 MHz, with three operational modes - 1024 tap, 512 tap, or 256 tap configurations (the product of data rate and taps cannot exceed 1024 tap×MHz). Moreover, the filter is programmable and the data and coefficients are full complex numbers. The word lengths of data and coefficients are rather small though, the data is two bits real and two bits imaginary, while the coefficients are three bits real and three bits imaginary. A feature to cascade more than one chip in sequence is also provided. This filter is a very large scale integrated (VLSI) circuit with a computation requirement of 7.2 GOPS, where each operation is a multiply or add. In addition to pulsar signal processing, a large tap filter such as this is also useful in a number of other fields like radar applications and pattern recognition of faint complex voltage signals.

1.2 Organization of the report

A brief introduction to pulsars and the problem of dispersion removal is given in chapter 2. This is followed by an overview of the CDRP system in chapter 3. Discussion on specifications and design considerations for the FIR filter is presented in chapter 4. Chapter 5 discusses quantization issues for the coefficients and data for the FIR filter. Architecture of the chip and its implementation is presented in chapters 7 and 8. Verification is given in chapter 9, and finally some conclusions are drawn in chapter 10.

The chip layout of the FIR filter is included in an appendix.

Introduction to Pulsars

2

2.1 What are pulsars?

Once a star exhausts all its fuel, its inevitable death can have three results. Small stars like our sun explode in a nova and degenerate into vapid white dwarfs. Bigger stars explode in a much more spectacular super-nova and then form neutron stars. Even larger stars are theorized to form black holes.

Neutron stars are extremely dense yet relatively small objects. Results have been found with these stars having a radii of about 10 km and masses comparable to the Sun's [1]. Some of these neutron stars are born with a strong dipole field. This special class of neutron stars are called *pulsars*. About 500 pulsars have been discovered since 1967.

In other words, pulsars are highly magnetized rotating neutron stars.They have the special property of emitting pulses with a striking regularity. The pulse, or the observed beam of radio emission rotates with the neutron star, much like the beacon of a lighthouse. We observe pulses that repeat at the star's rotation rate as the beam sweeps past the Earth. The star is rock stable thus sustaining a highly undisturbed and regular rotation period.

A special class of pulsars are millisecond pulsars. The distinguishing property of these pulsars is that their rotation periods are typically in the range of 1 to 10 ms. In comparison to long period pulsars, millisecond pulsars also have a more stable period.

2.2 Application of millisecond pulsars

Millisecond pulsars are very useful in a number of applications:

- Study of formation and evolution of neutron stars
- Study of evolution of binary star systems
- Probes for pulsar magnetosphere
- Pulsar timing

Pulsar timing is one application which is of interest not only to astronomers but also to other communities. The timing data from millisecond pulsars can be applied to the following applications [2]:

- Celestial mechanics
- Time Coordinate definition and gravitational wave detection
- Space reference definition and spacecraft navigation
- Interstellar turbulence

Needless to say, these applications are of high value, and a large amount of research effort is being conducted in this area.

2.3 Model for pulsar signals

The model for the pulsar signals, as we observe them, is shown in figure 1. The pulsar signal at the star can be modelled as amplitude modulated noise. The amplitude, or the intensity of the nth pulse $I_n(\phi)$ varies chaotically within a given pulse, and from pulse to pulse. However, a long term average of many

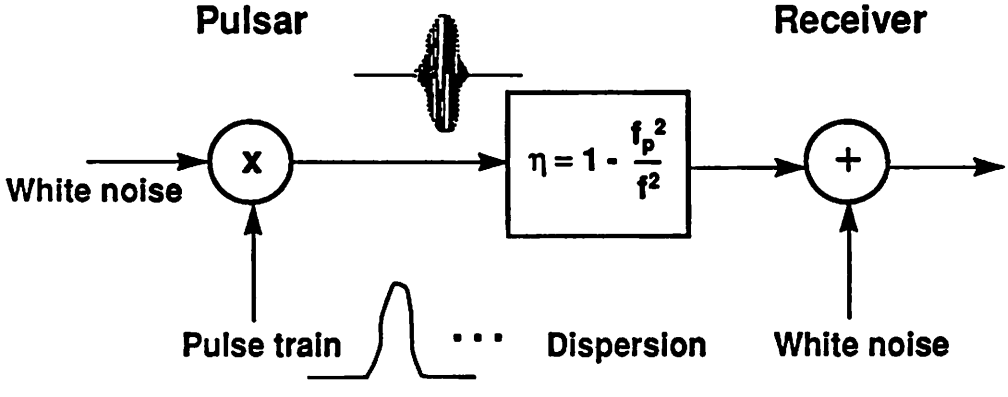


Figure 1: Model of the Pulsar Signal

pulses develops a stable pulse profile that is characteristic of each object. Figure 2 shows one such pulse profile. During its journey to the Earth, the pulse must travel through the interstellar medium which is interspersed with tenuous plasma. The tenuous plasma has an index of refraction η which has a frequency dependent term (see figure 1). The frequency varying index of refraction is responsible for the undesirable effect of dispersion. Finally, the receiver can be modelled as additive white noise. The noise can be substantial, and as expected, pulsar signals are generally buried in large quantities of noise.

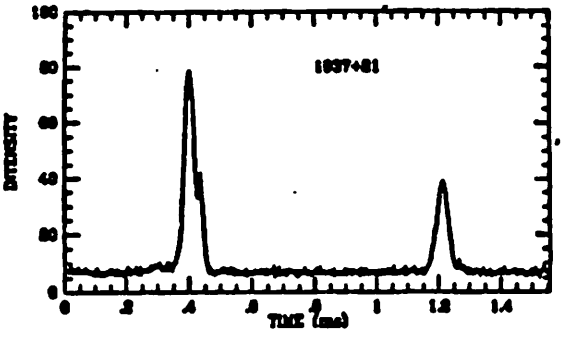


Figure 2: Profile of a Pulsar Signal

2.4 Dispersion in Pulsar Signals

The frequency dependent index of refraction of the interstellar medium results in non-linear dispersion of the pulsar signal. Figure 3 shows this effect. Across a narrow band BW, at radio frequency ν_o, the delay $\Delta\tau_D$ is given by

$$\Delta\tau_D = 8.3\ \mu s\ DM\ (pc\ cm^{-3})\ BW\ (MHz)/\ \nu_o^3\ (GHz)$$

where *DM* is the column density of electrons in the interstellar medium in astronomer's favorite units (1 pc [parsec] = 3×10^{16}m ≈ 3 light years).

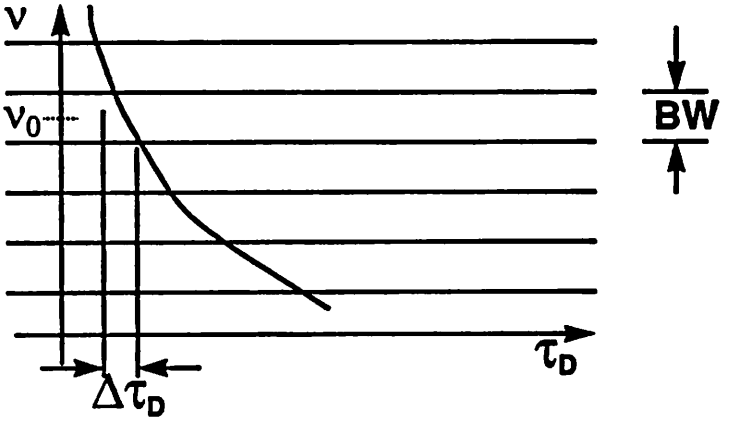


Figure 3. Dispersion in Pulsar Signals

2.5 Dispersion removal techniques

For any useful pulsar timing application, the dispersion from pulsar signals must be removed. There are two main techniques for removing dispersion - the incoherent technique and the coherent technique.

2.5.1 Incoherent dispersion removal (Pre-detection)

The frequency dispersion incurred by the pulsar signal in the ionized plasma along the line of sight can be removed by dividing the band B into smaller bands BW with a filter bank. Each signal can be detected and averaged synchronously with the period. Dispersion between bands is removed by shifting and adding [3]. This technique reaches a natural limit when $1/2BW$ exceeds the desired sample time which we take as $\Delta\tau_D$:

$$1/\ (2\ BW) = \Delta\tau_D \rightarrow \Delta\tau_D^2 = (8.3\ DM)\ /\ 2\nu^3$$

Choosing a $\Delta\tau_D$, or equivalently BW, one can find the minimum value of ν. This minimum presents a barrier which is undesirable. This is the main motivation for the complicated yet effective coherent dispersion removal technique.

2.5.2 Coherent dispersion removal technique (Post-detection)

In [4], Hankins and Rickett develop an analytical representation of how the dispersion only modifies the phase of the propagating electric field in a frequency dependent manner across the spectrum, and how the original pulse can be reconstructed by inverse filtering of the sampled voltage. Figure 4 displays the difference between the incoherent and coherent techniques and graphically demonstrates the superiority of the coherent technique. In the coherent technique, the sampled voltage can be passed through a finite impulse response (FIR) prior to detection to remove the dispersion. Hankins attempted to do this with a real-time 1-bit system during his Ph.D. research in 1970. Later Hankins and others have used Fourier techniques in software. Recently he and his colleagues have constructed and put to regular use a Reticon-based FIR dispersion remover [5].

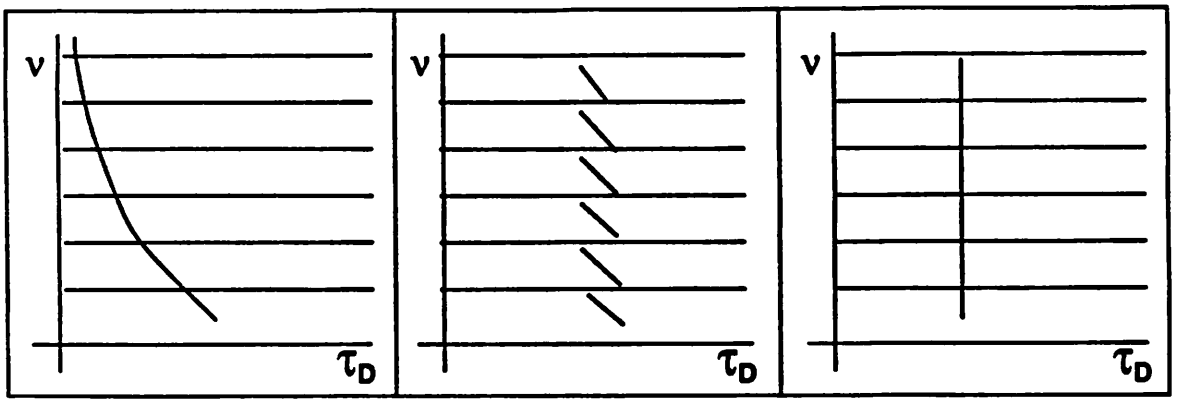


Figure 4: Dispersion Removal Techniques

The FIR restoring function for signals in a small fractional bandwidth at a radio frequency ν_0 can be written as:

$$h_+(t) \approx exp\left[\left(-j\frac{\pi\nu_0^3}{8.3DM}\right)t^2\right] \times exp\left(j(\pi/4)\right)$$

Alternatively, the dispersion can be removed in the frequency domain by taking the FFT of the input signal, multiplying it with the frequency response of the interstellar medium and subsequently taking the inverse FFT of the product. The narrow band frequency response is given by:

$$H_+(\nu_o + \Delta\nu) \approx exp\left(j\frac{2\pi\nu^2 D}{\nu_0^3}\right)$$

where D (sec Hz^2) = DM (pc cm^{-3}) / (2.41×10^{-16}). These approximate relations are derived from limiting the interstellar medium transfer function taylor series expansion to the quadratic term [4]. Although the higher terms are generally neglected, they can be incorporated in the FIR filter if one wishes because there is no requirement of symmetry in the coefficients. The number of samples required for the FIR with critical sampling of the complex data at a rate *BW* is:

$$N = BW \times \tau_D = \frac{8.3DM(BW)^2}{\nu_0^3}$$

The rapid growth of N with BW indicates that a channelization into sub-bands approach to an FIR coherent processor is essential. A single FIR filter for a bandwidth BW of 100 MHz at ν = 1 GHz and DM = 100 requires 8×10^6 taps while a bandwidth of 1 MHz requires only 8×10^4 taps.

After studying various requirements, the Center for Particle Astrophysics decided upon a specification of 1024 taps FIR filter with a 1 MHz channel bandwidth. The detailed specifications of the filter are mentioned later.

2.6 The FIR filter approach

As mentioned earlier, there are two basic methods of implementing coherent dispersion removal. First is the FIR filter method, while the second is the FFT method. Both have their own advantages. It may be generalized though, that the FFT is a board level (or part of a board) solution while the FIR filter is a chip level solution. To do an FFT of 1024 points, using commercially available parts, about two chips would be required. Also, FFT must be complemented by an inverse FFT doubling the number of chips. Finally a lot of memory and glue logic portions are required. All these components make the FFT a much more bulky solution. If we were interested in processing much more data than 1024, FFT would definitely be the optimal solution because of the $N\times\log_2 N$ computational requirement as opposed to N^2 for an FIR filter.

Comparing the various advantages and disadvantages, the Center for Particle Astrophysics selected the FIR filter approach. Observing the massive computational requirement of 7 GOPS, a custom integrated circuit was required. Using current day general purpose digital signal processors, only 30-50 taps could be implemented for a 1 MHz data rate which falls short of the 1024 tap goal. With regards to the state-of-the-art programmable FIR filters, a 30 MHz FIR filter with 16 taps and 8 bits quantization was reported [6]. Also, LSI Logic markets a 20 MHz FIR filter with 64 taps and 8 bit quantization and a 15 MHz FIR filter with 1024 taps, 1 bit data, and $1\frac{1}{2}$ bit coefficients [7].

Overview of the CDRP system

3

The Coherent Dispersion Removal Processor machine being developed at the Center for Particle Astrophysics utilizes the technique described in the previous chapter. The block diagram of this system is as follows:

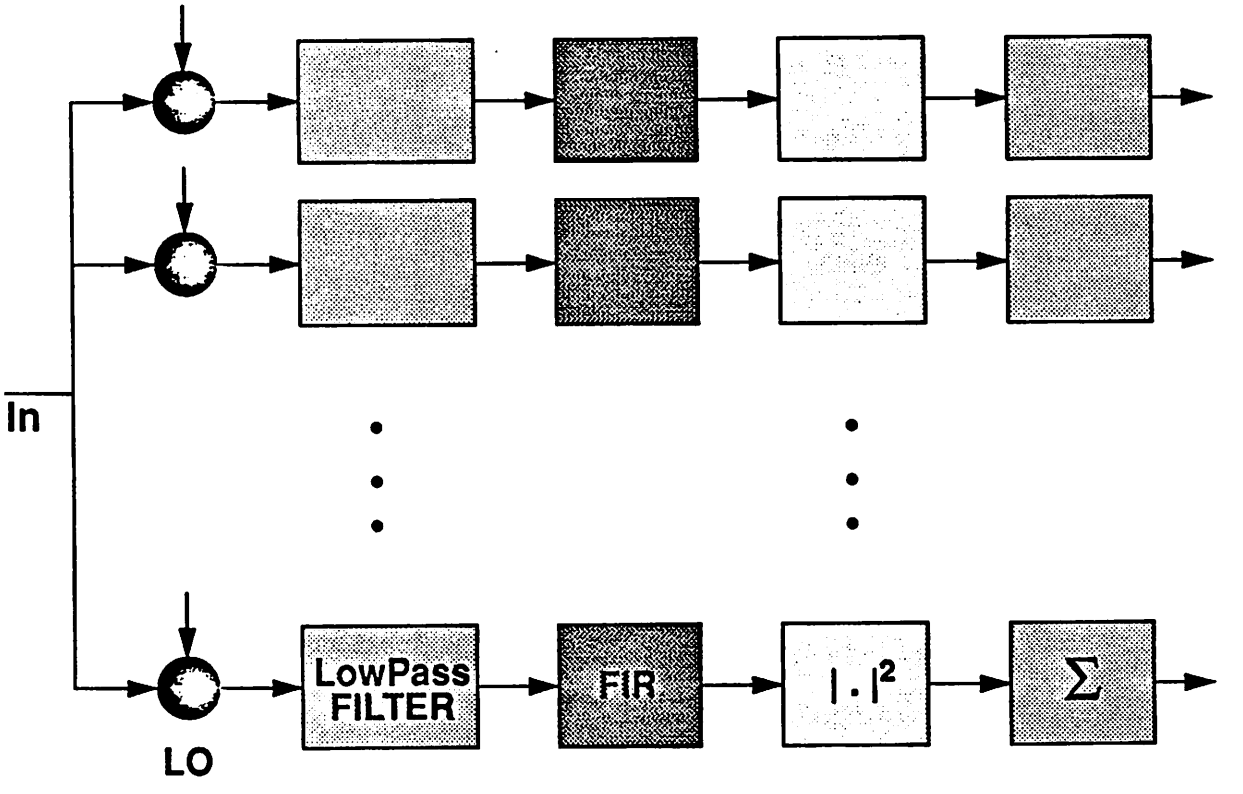


Figure 5: CDRP System

The local oscillators combined with the low pass filters are used for demodulation and channelization of the input signal. Quadrature demodulation is used, thus yielding complex signals. Quadrature demodulation has a number of advantages over simple demodulation [8]. The main reason for complex demodulation is that when the signal is mixed down to base-band, both positive and negative frequencies are present. The only way to sort them is to use to mixers, one sine and the other cosine, yielding complex data. It is possible to use a single sideband mixer to mix the IF band to only positive frequencies, but this is a more complicated and expensive device and one needs to sample a real voltage at twice the rate given by the bandwidth. Hence, typically the input signal bandwidth is 100 MHz, which is subsequently broken into 1 MHz bands. Each band is fed to a dedicated FIR filter chip which removes the dispersion from the band. Every band is then detected. Following this is signal averaging, which sums a number of periods. Over the time of averaging, a pulse profile finally develops and emerges above the noise level. The rest of the report discusses the design of the FIR filter chip which was the focus of this research project.

Design Considerations for the FIR Filter

4

4.1 Specifications and design goals

The main requirement for the FIR filter chip was agility. Based on this requirement the Center for Particle Astrophysics laid out the following specifications:

- 1024 taps operation with up to 1 MHz data rate
- 512 taps operation with up to 2 MHz data rate
- 256 taps operation with up to 4 MHz data rate
- Programmable non-adaptive coefficients
- Ability to cascade the chips

Based on these requirements, the main design trade-off was between speed and area. It was desirable to have a chip as small as possible but it is useless to have a small chip with an unreasonably high clock rate. Based on current day technology, we decided to clock the chip at 32 MHz and aim for a die area of about 11×11 sq. mm.

4.2 Design Issues and Methodology

4.2.1 Clocking Scheme

As mentioned, the target clock speed for the chip was 32 MHz. Most of the cell library uses a two phase non-overlapping clocking strategy. This strategy

is also easiest to implement, which resulted in our selection of this scheme for the chip.

4.2.2 Logic Design vs. Circuit Design

An initial estimate revealed that the final chip would be core limited and would also be an oversize device. This underlined the need to conserve silicon and have a layout as compact as possible. Library cells provide the advantage of extremely fast design time, but often fall short with regards to performance and area. Considering this, both logic design and circuit design approaches have been used in the chip. Circuit design has been done only when library cells did not satisfy the specific requirements.

4.2.3 Static vs. Dynamic Design

In [9] Yeung mentions, "While dynamic circuits are usually superior to their static counterparts in terms of speed and area, they are also more difficult to integrate into the system due to their timing constraints. A static data-path cell is easier to use and more likely to be reused than a dynamic cell of equivalent function. In addition, the power spikes, increased clock loading and potential timing problems associated with dynamic designs may more than outweigh its advantages. Static design is further favored when there is no urgent need to conserve chip area."

Looking at these trade-offs, it is clear that for simplicity, a static design is favored. For most of the chip, therefore, a static approach was used. However, for the shift register section (described later), where area was of paramount importance, a dynamic C^2MOS approach was used. C^2MOS, as one may recall, also has the pleasant property of being fairly insensitive to clock skews.

4.2.4 Power Consumption

By using CMOS technology and by avoiding circuit techniques that consume large amounts of static power, power consumption of the FIR chip was minimized. Also, for the critical portion of the chip, the shift register section, C^2MOS logic was used, which has absolutely no static power.

4.2.5 Computer Aided Designs

To facilitate the design process, CAD tools were used extensively to assist logic design, layout, and circuit and logic simulation.

4.2.6 Layout Strategy

Manual layout was avoided as much as possible by using the *LAGER* [10] silicon compiler and a cell-based modular design. Manual layout was done only on the cell and sub-block level, above that the layout was generated automatically by using *LAGER*. Moreover, a bottom up approach was used in the layout process so that blocks of layout were first generated and simulated before they were assembled to form more complex blocks. Manual optimization of the machine generated layout was also performed after complete verification of the design.

4.2.7 Simulation

Simulation occupied a very significant portion of the design cycle. Simulation was done mainly at three levels. The algorithm used to implement the architecture was simulated using a high level flowgraph language - *SILAGE* and the programming language C. The various cells manually laid out were simulated using *SPICE*. The final transistor level simulation was done using *irsim*, an interactive event-driven simulator for MOS transistor circuits.

Quantization effects 5

5.1 Quantization

In order to reduce the complexity of the FIR filter, which in turn reduces the area and the speed constraints, it is very important to have an efficient quantization scheme. It would be desirable to have the data and the coefficients of the FIR filter quantized to the minimum number of bits without adding significant quantization noise. Eventually, data quantization was fixed at 2 bits real and 2 bits imaginary, while the coefficient quantization was fixed at 3 bits real and 3 bits imaginary.

5.2 Simulations

Extensive simulations were done to ensure that the quantization scheme was not adding disproportionate amounts of noise. Before presenting the quantization scheme and the simulation results, it would be useful to look at the simulation model used. Figure 6 shows the model which was implemented in the *SILAGE* flowgraph language. To understand the effect of the low-pass filter, noise

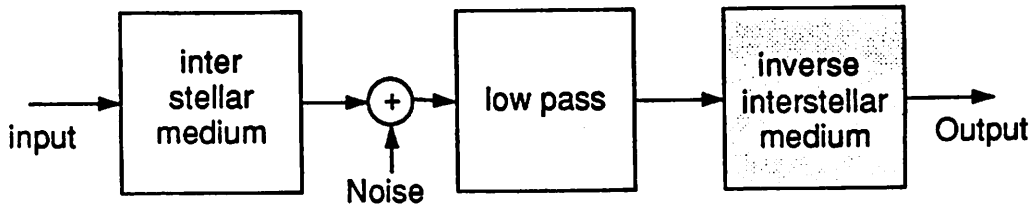


Figure 6: Quantization Simulation Model

was generated at 19 times the output rate. The sum of noise and the dispersed signal was low-pass filtered and subsequently decimated to the output rate.

5.3 Coefficient Quantization

The impulse response of the interstellar medium (see chapter 3) is pure phase with constant amplitude. This is also true for the inverse interstellar medium impulse response. In other words, all the coefficients derived from this transfer function are on the unit circle. This motivates a vector quantization scheme, figure 7a, where the coefficient pair is quantized to a pair with the shortest distance away from it. (Note that the quantized points themselves do not all lie on the unit circle because they are forced to be integers). The simulation results were quite promising for these quantization points. Figure 8 shows the results. In figure 8a, result is shown when an impulse is fed to the system with no quantization occurring in the IISM (inverse interstellar medium) FIR filter block.

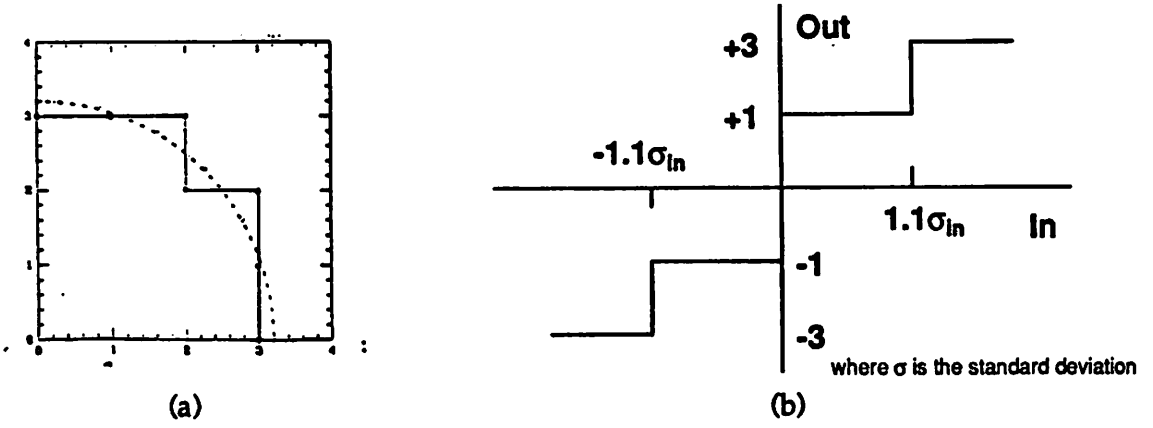
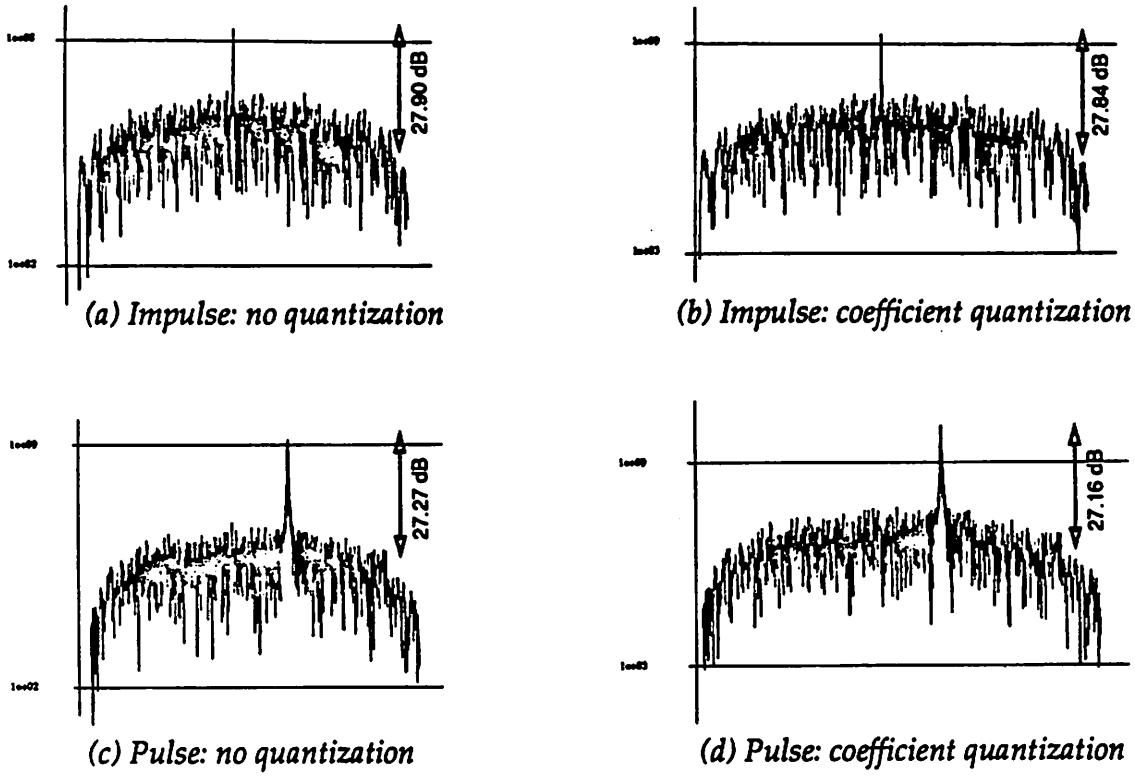


Figure 7: Quantization Schemes (a) Coefficients (b) Data

Figure 8b shows the result for the same input waveform with only the coefficients quantized in the IISM filter. Figure 8c and 8d show results of the same test on a narrow pulse. The additional noise for the impulse was 0.06 dB while for the pulse was 0.11 dB (The signal to noise ratio used is peak to average). This noise arises from two effects - the signal to noise ratio loss and dynamic range limit. Clearly, this amount of noise is within acceptable limits.



(a) Impulse: no quantization

(b) Impulse: coefficient quantization

(c) Pulse: no quantization

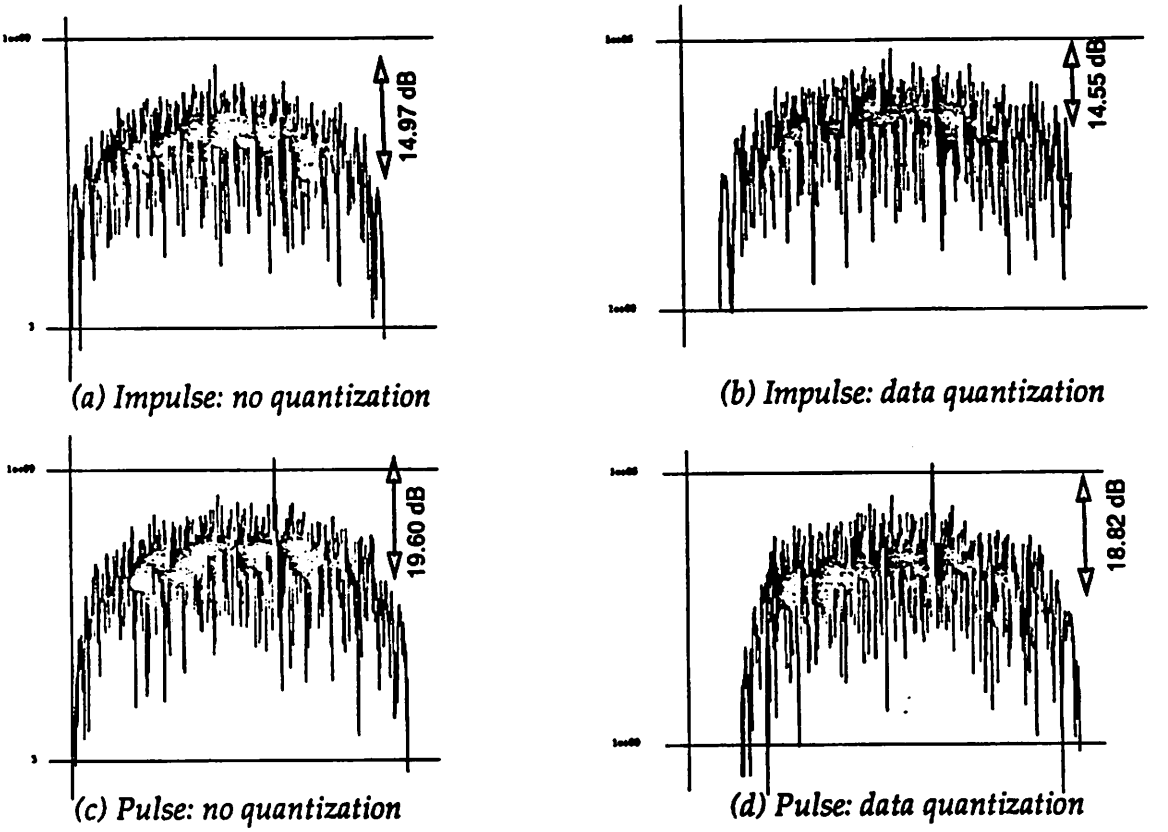
(d) Pulse: coefficient quantization

Figure 8: Coefficient Quantization Results

5.4 Data Quantization

The key fact used while deciding upon to the number of bits for data quantization was that the signal is highly noise dominated. Intuitively after a certain word length, further increase in the number of bits does not extract any more information. The commonly used quantization length in currently used digital correlators is two. Since the kind of data being dealt with is the same here, a two bit quantization scheme was used. The scheme is shown in figure 7b. The relationship between the input and the quantized output shown in the figure has been found to be satisfactory [11]. This scheme is used for both the real component of the data and also the imaginary.

The effects of quantization noise were studied and the results are summarized in figure 9. Similar to figure 8, figure 9a shows the result with an impulse fed to the system with no quantization in the IISM FIR filter. During this



(a) *Impulse: no quantization*

(b) *Impulse: data quantization*

(c) *Pulse: no quantization*

(d) *Pulse: data quantization*

Figure 9: Data Quantization Results

experiment, much more noise was added than in the previous experiment to get a more accurate picture of data quantization effects. For instance, the dispersed signal in the previous experiment was buried in noise with 5 times the amplitude, while 30 times in this experiment. Figure 9b shows the result with the same impulse fed to the system with data quantization Figure 9c and 9d summarize the same results for a pulse. additional noise in the two cases is 0.42 dB for the impulse case and 0.78 dB for the pulse case. As with the previous case, this noise is a result of signal to noise ratio loss and dynamic range limit. These numbers are acceptable and agree with the results from correlation experiments.

Architecture of the FIR

6

The main trade-off while deciding upon the architecture was between speed and area. Fewer parallel data-paths would result in smaller area, but a higher clock rate as opposed to an architecture with more parallel data-paths. The architecture finally selected is based on the work done by the Stanford DSP group [11]. It was decided to use a 32 MHz clock for the chip which seems like a reasonable target considering present day technology. Since the data rate is assumed to be 1 MHz, this constraint implied that the FIR filter chip would have 32 complex multipliers to do the 1024 multiplies required for the 1024 taps. The architecture is shown in figure 11; thus, there are '32 blocks' each implementing 32 taps, and in total implementing 1024 taps. Each of these blocks consists of 1 complex multiplier, 32 data shift registers, and 32 coefficients shift registers. Intercommunication between blocks is minimal. The data circulates for 31 clock cycles and during the 32nd clock cycle it has to be transferred between blocks. The data thus traverses in a snake like fashion. The coefficients keep circulating at all times (except coefficient load). Figure 10 shows a simple timing diagram to illustrate the inter-block communication. The above explanation holds for the 1024 tap case, and a number of tricks had to be introduced to keep a provision for different filter sizes (see specifications).

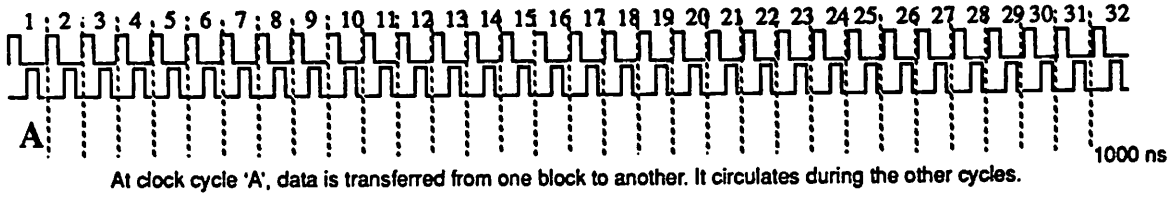


Figure 10: Simple Intercommunication Timing Diagram

The 32 real and 32 imaginary results from the multipliers have to be summed up. There are a number of ways to implement the summation, e.g. a sequential tree, a binary tree, or a wallace tree approach. The binary tree was selected because it is area efficient, is easy to pipeline, and has library support. To meet the speed requirements, the adder tree in fact has four levels of pipelining. The sum from the adder tree is passed on to an accumulator which outputs the data at the same rate as the input data rate. A controller and some output circuitry is also provided to facilitate cascading of more than one chip.

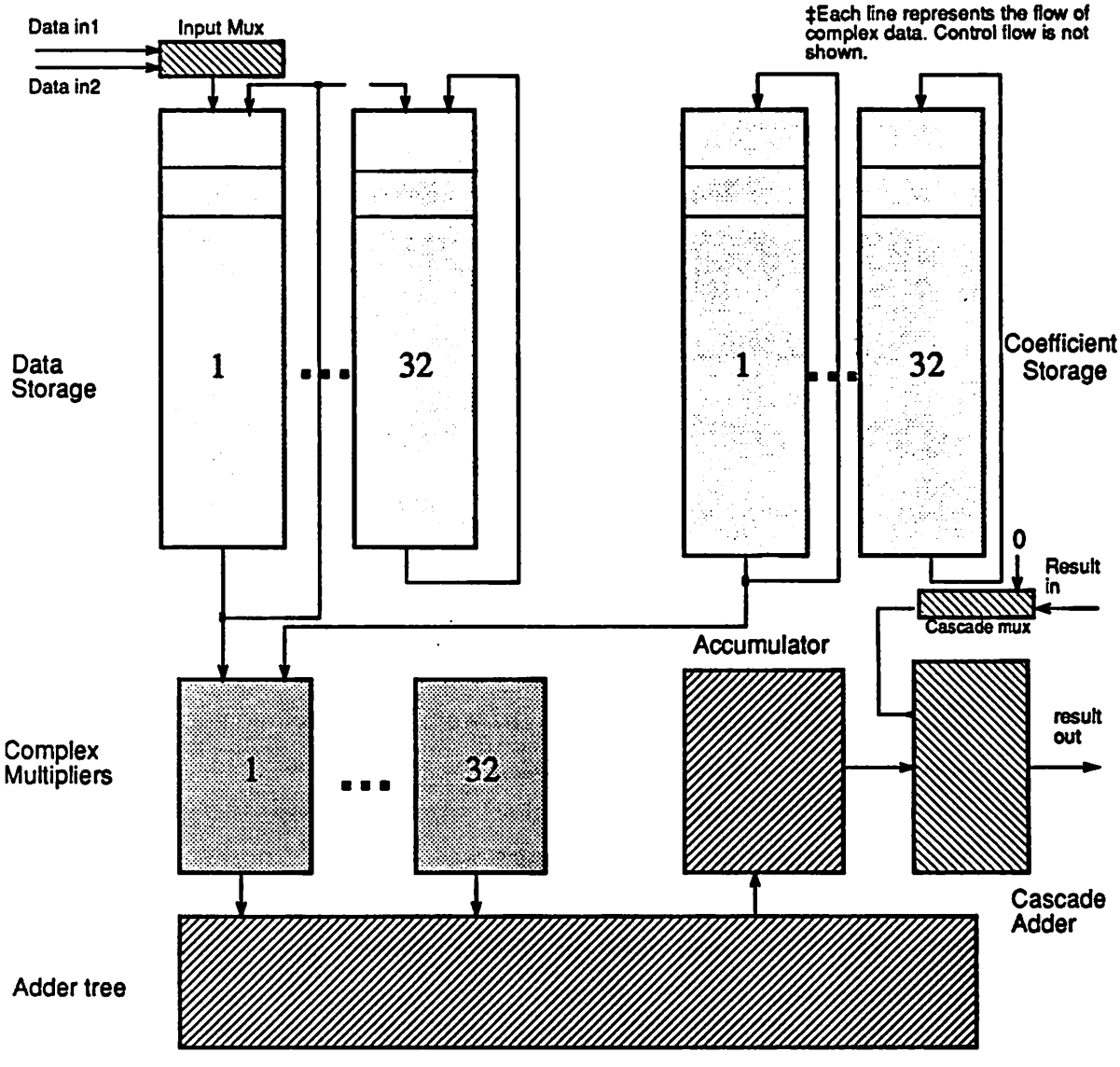


Figure 11: FIR Filter Chip Architecture

Implementation of Macro-Functions

7

7.1 Memory

The chip requires a storage of 1024 coefficients and data. This translates to a memory requirement of 10 Kbits. Various implementations shown in table 1 were tried for this block.

TABLE 1. Memory Block

Block	Advantages	Disadvantages
DRAM	Library part	Speed, routing and address generation
SRAM	Area	Routing and address generation
FIFO	Library part	Area, data management, and speed
Shift Register	Simplicity	Power and clock lines

The DRAM had the advantage of being a library part, and hence having a zero design time, but had the fatal drawback that it could not function at the speed required. The SRAM had the advantage of being very compact, but the disadvantage that all the area saved due to a compact block size would be lost in routing and address generator block. Another disadvantage of using an SRAM was that considerable circuit design was required. Another structure considered for the memory block was the FIFO. This block was fairly unsuitable, and the only advantage was that it was a library part. Finally a shift register implementation was considered. The advantage here was simplicity. However, there were a number of disadvantages from a power and clock distribution point of view since every shift register would be active during each clock cycle. Eventually, the shift register approach was chosen because of simple design and

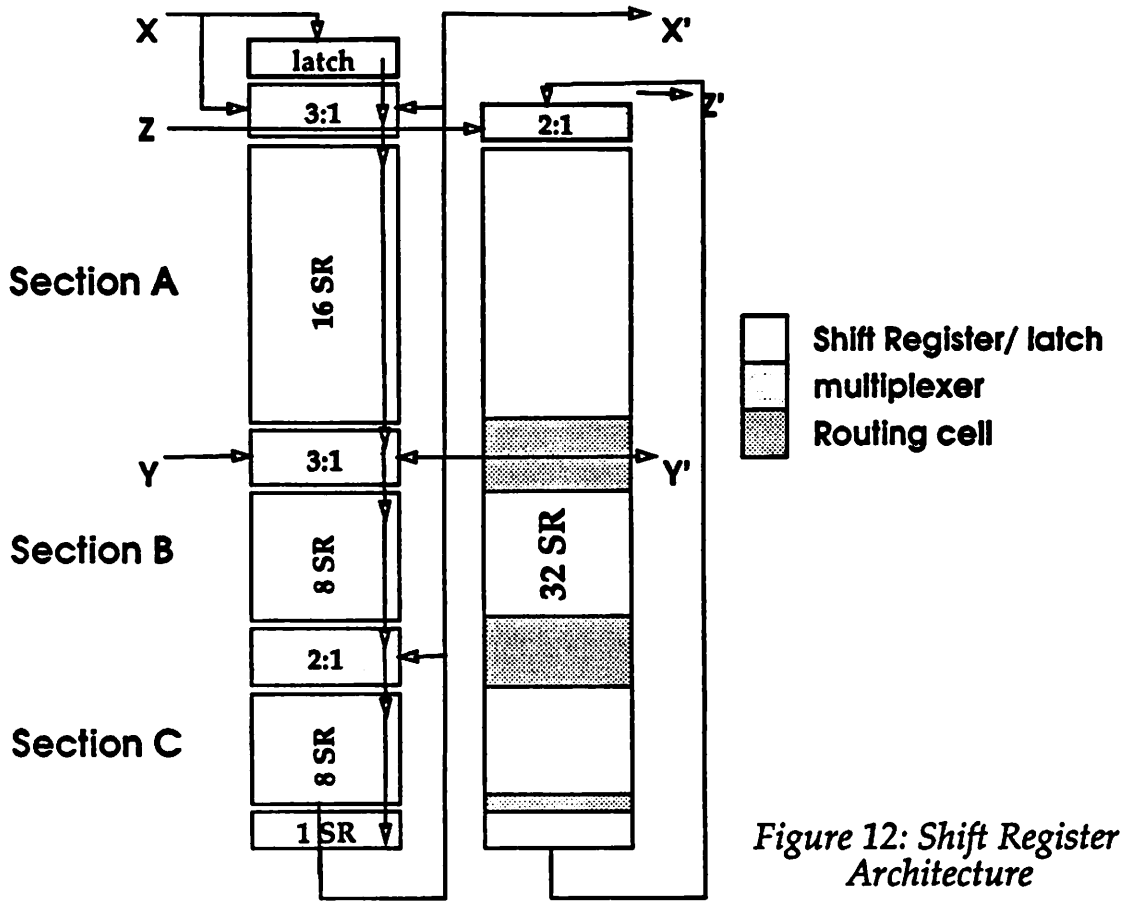


Figure 12: Shift Register Architecture

relatively compact area. Just to give an idea of the problems this block created, the block is expected to consume about 0.4W and the four clock lines have a capacitance of about 177 pF each, driving which in turn consumes about 0.9W.

A detailed architecture using this approach is shown in figure 12. For the 1024 tap mode, all three sections A, B, and C are used for circulating the data. The uppermost 3:1 multiplexer is used to switch between circulating data and data from the latch, which is the actually the delayed data from the previous block. The latch stores the data for 32 cycles before injecting it into the stream again. In the 512 tap mode, only sections B and C are used for circulation. The second 3:1 multiplexer decides between the circulating data and the data coming from section A, which is effectively delayed data from the previous block originating from source X. Finally, in the 256 tap mode, only section C is used for circulation of the data and the 2:1 multiplexer selects between the circulating data and the delayed version of data from the previous block from source Y through section B. C programs which mimic this architecture are included in the appendix.

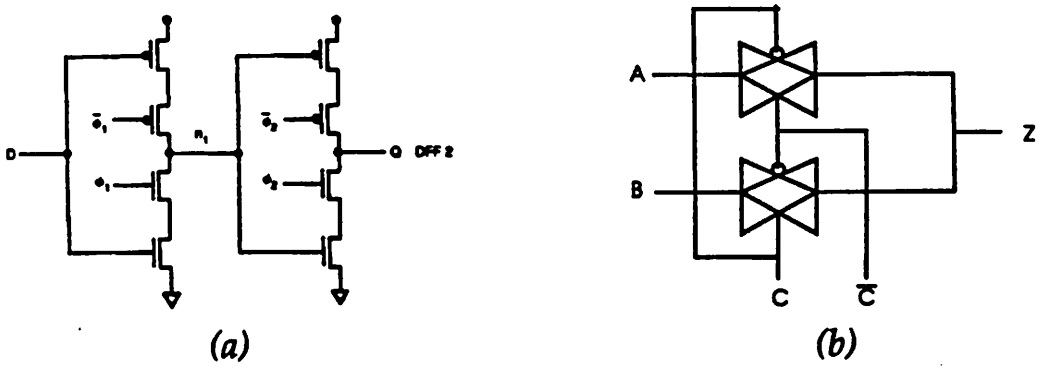


Figure 13: Circuits used (a) Shift Register cell (b) Multiplexer cell

The circuits used to implement the shift registers were of the C^2MOS family. These shift registers are compact and relatively insensitive to skews. Also, there is no static power in this logic family. The price paid is the high clock capacitance and dynamic storage of charge. See figure 13a for the circuit. The multiplexers circuit used was just pass gates with outputs connected. See figure 13b for the circuit.

Once these basic cells were laid out in *MAGIC* and simulated using *SPICE*, they were assembled into sub-blocks using *MAGIC*. These sub-blocks were incorporated into the *LAGER* frame-work and the next level of hierarchy was formed using automatic tools. These memory blocks were simulated using *irsim*.

7.2 Multiplier Block

The requirement here is to multiply a complex data by a complex coefficient. The block should supply this product at the prescribed 32 MHz speed. A number of implementations were tried here too as shown in table 2.

TABLE 2. Multiplier Block

Block	Advantages	Disadvantages
ROM	Library part	Speed
PLA	Library part and small	Speed
Data-path	Library parts and speed	Area
Standard Cell	Library parts, smallest, and fastest	-

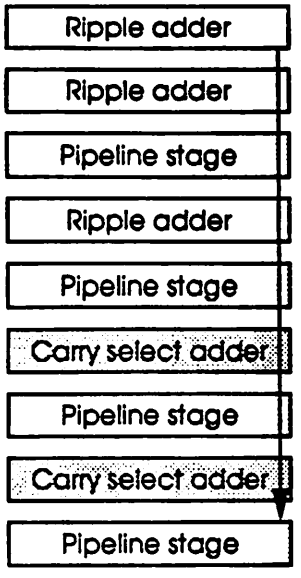


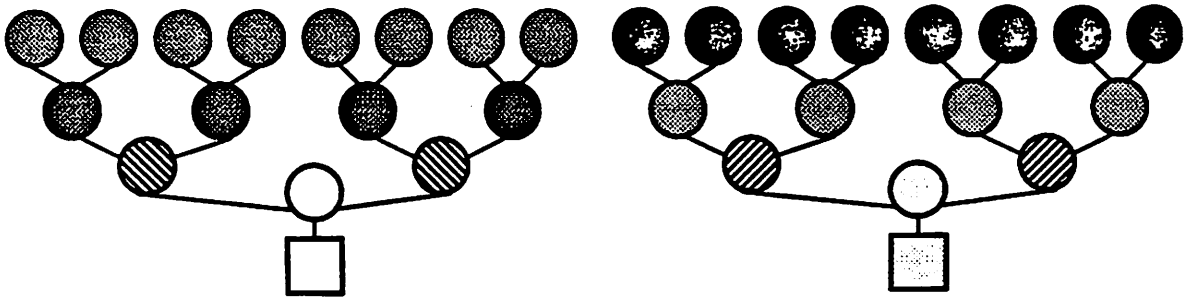
Figure 14: Pipelining in the Adder Tree

The two clearly superior implementations were the data-path and the standard cell implementations. Among the two, the standard cell implementation was significantly smaller. As opposed to a single real multiplier of size 307×535 λ generated by the data-path compiler, the standard cell layout was only 251×328 λ. See the appendix for a comparison of layouts.

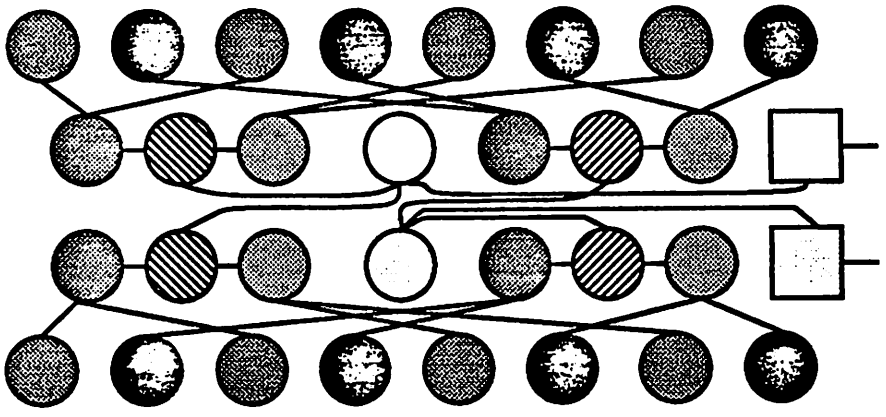
7.3 Adder Tree

The main challenges while designing the adder tree were that it had to be fast and compact. To meet the speed problem, the adder tree was heavily pipelined, see figure 14, with four levels of pipeline registers. Fast carry select adders were used where ripple adders became too slow. To achieve a compact layout, very careful planning was done, and judicious scaling of the bits was done. With regards to scaling of bits, the product from the multiplier is 6 bits wide, and the adder tree grows in bit width from 6 to 11. Note that there is no saturation or truncation in this tree.

The careful planning of the adder tree is shown in figure 15. Figure 15a shows two half size adder trees. The one on the left is the real adder tree and the one on the right is the imaginary one. As one can notice, the number of adders in the top level is almost identical to the number of adders in the subsequent levels. It is therefore logical to try and place the entire adder tree in two layers. That is exactly what is done. In addition to this placement, the two adder trees - real and imaginary are also interleaved. Since a two layer layout turned out to be too thin and wide, it was folded over into four levels, figure 15b, to achieve a very compact layout. See the layout in appendix to compare with figure 15b.



(a) Inefficient Placement

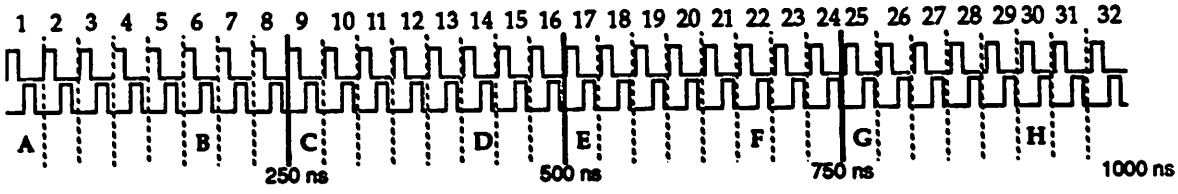


(b) Transformed Placement

Figure 15: Layout Strategy for the Adder Tree

7.4 Controller

The controller is used to control the various modes, and to load up the coefficients. The timing diagram of one of the modes, the 256 tap mode is shown in figure 16. During states A, C, E, and G, the multiplexer lines for the data storage are changed so that the circulation mode turns into pass-along for one cycle. In states B, D, F, and H, the output of the accumulator is valid, and has to be cleared for the next cycle. The implementation options here were to use a PLA or to use a standard cell implementation. Due to the slow speed of a PLA, the standard cell implementation was used.



A - Load data, shift cycle for all multiplexers, latch new data, generate slow-clock

B - Clear Accumulator, Latch accumulator output

C - Load data, shift cycle for all multiplexers, latch new data

D - Clear Accumulator, Latch accumulator output

E - Load data, shift cycle for all multiplexers, latch new data

F - Clear Accumulator, Latch accumulator output

G - Load data, shift cycle for all multiplexers, latch new data

H - Clear Accumulator, Latch accumulator output

Figure 16: State/Timing Diagram for the Controller in the 256 Tap Mode

7.5 Output Circuitry

The output circuitry is shown in figure 17. This was implemented using the data-path library. This circuit is used to provide cascade ability for the FIR filter chip.

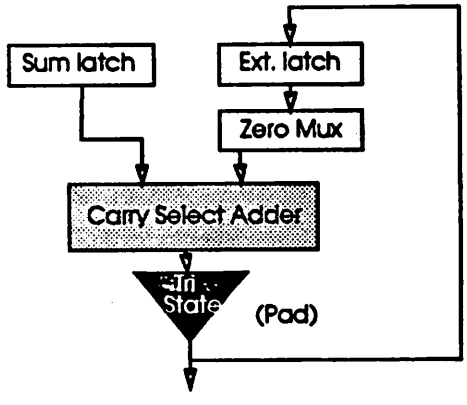


Figure 17: Output Circuitry (simplified)

7.6 Overall Layout

Once the structure of the adder tree was fixed, it was clear that multipliers would have to feed it from both top and bottom sides. Hence two banks of multipliers were laid out, and feeding the two multiplier blocks, two memory blocks were laid out. Since one multiplier was being fed by one

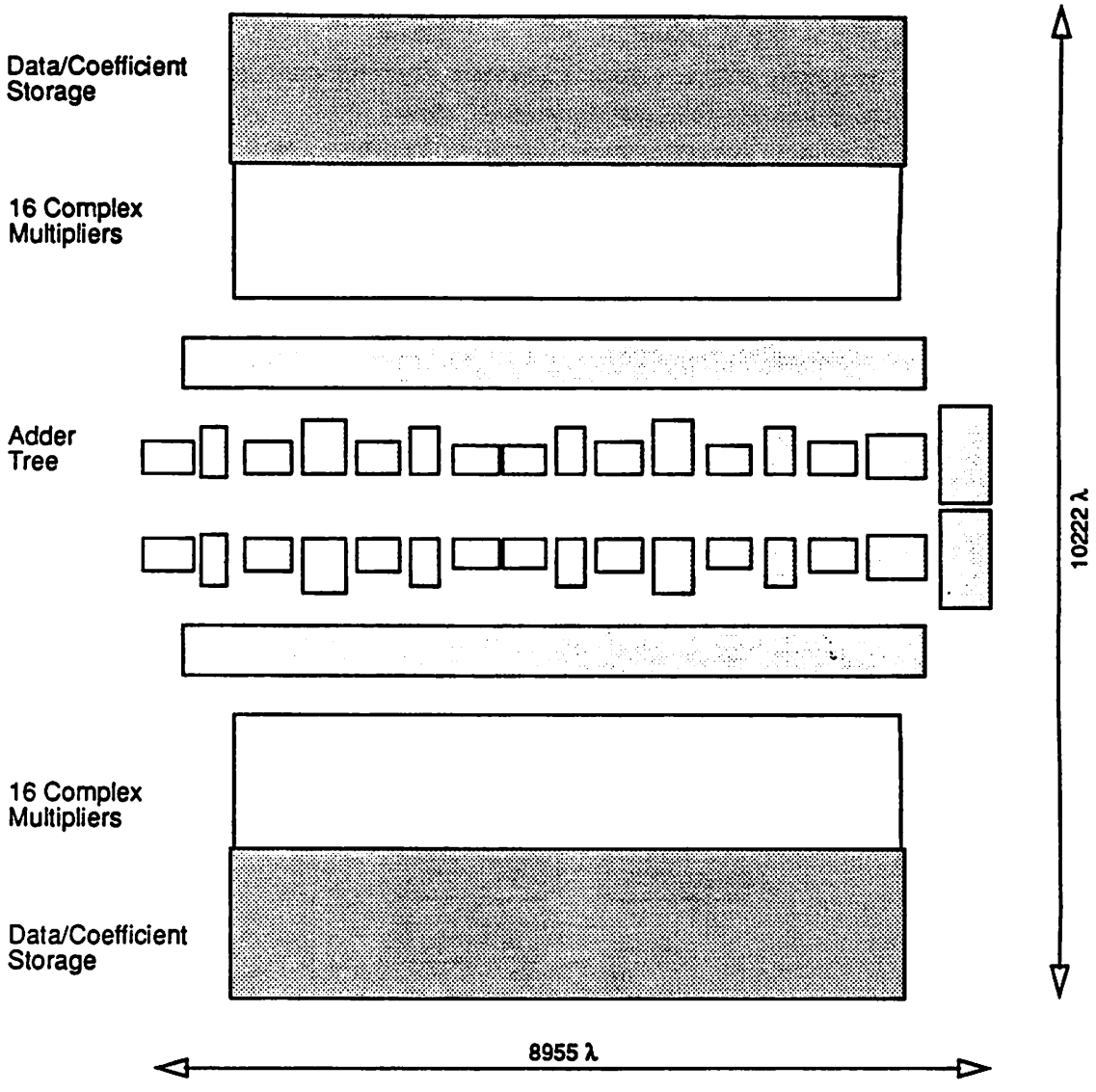


Figure 18: Overall Layout of the Chip (Main Components only)

coefficient shift -register and one data-register, it was obvious that pitch-matching these two blocks would be highly area efficient. This is exactly what was implemented. Figure 18, shows the layout of the overall chip (only the main components, see appendix for the chip photograph).

7.7 Clock generation and distribution

A two phase non overlapping clock is used. A cross coupled NOR-gate, figure 19, clock generator is used. The chip also accepts external two phase clocks. The two phases are routed globally while their inverses are generated locally. The overall capacitance on *each* clock line, ϕ_1, ϕ_2, $\overline{\phi}_1$, $\overline{\phi}_2$ was about 187 pF. To drive this huge load and to ensure tolerable clock skew (<1ns), careful layout of the clock lines was done. The buffer scheme is shown in figure 20, the layout on chip is shown in figure 21, and the *spice* results are shown in figure 22.

7.8 Power distribution

Since the chip has a total of about 152,000 transistors, it was necessary to generate a well-planned power distribution tree. Care was taken to limit IR drops and to avoid any possibilities of electro-migration. The power network tree can be found in figure 23.

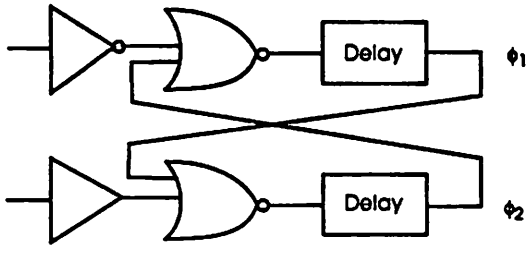


Figure 19: Clock Generation

7.9 Pads

Under normal circumstances, only the output buffers contribute to any ground bounce. However, in this

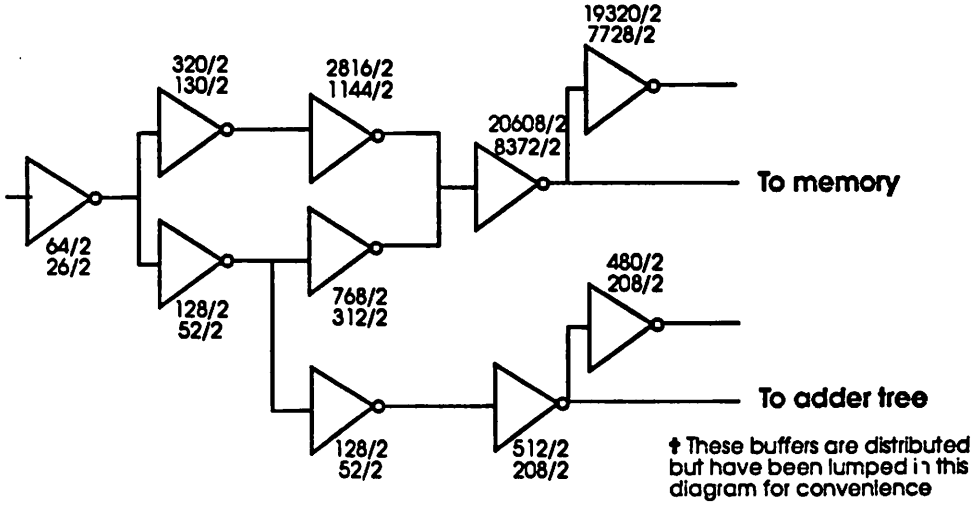


Figure 20: Clock Buffer Scheme

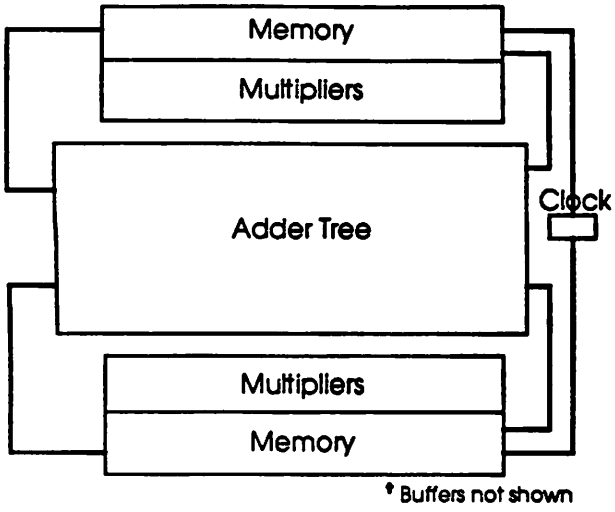


Figure 21: Clock Distribution

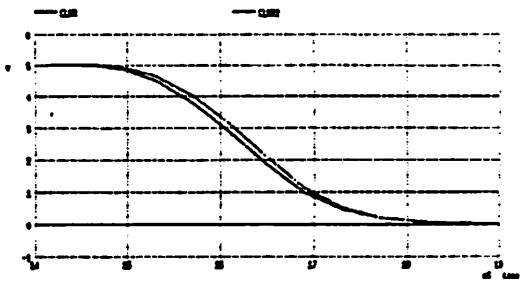


Figure 22: Clock Skew

case, the clock drivers were contributing significantly to the ground bounce problem. Figure 24 shows the *spice* result for the ground bounce assuming that the inductance from power line to chip is 0.5nH for the clock drivers. To avoid this problem, power and ground pads have been liberally distributed. For every 4 signal pads, there is one ground pad and one power pad. This number is well below the suggested conservative ratio of 6 [13].

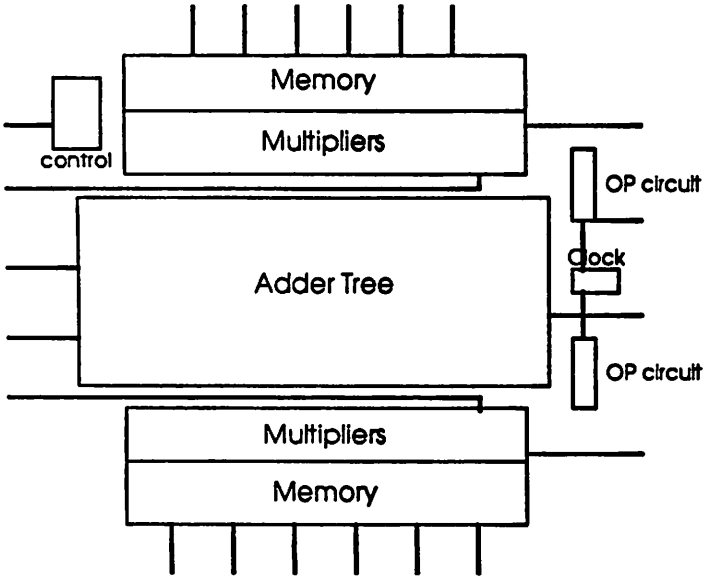


Figure 23: Power Network

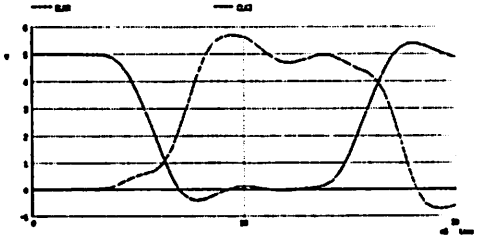


Figure 24: Clock Bounce (0.5 nH)
(as a result of ground bounce)

8 Verification

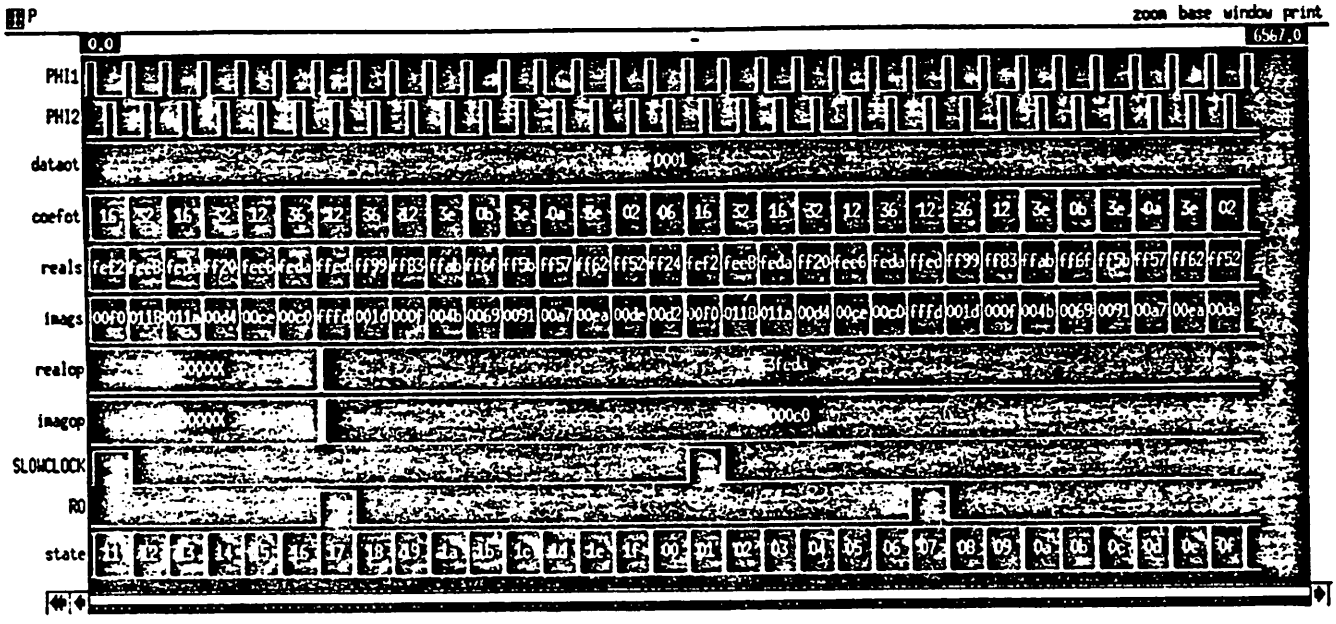
8.1 Simulations

It is futile to do a design without any simulations. For this chip, simulation was done at various levels of abstractions. The highest level of simulation was done in *SILAGE* flowgraph language and *C* to verify the functionality of the quantization scheme and the architecture. The code for the *SILAGE* program is included in the appendix. The next level of simulation was done using *irsim*. The entire chip and all the sub-blocks were simulated at this level. *Irsim* was used for verifying functionality at the chip level, and for verifying both functionality and timing at sub-block level. Some sections of the chip were simulated using *THOR* when the design was still at a fairly high level, and some portions were simulated using *SPICE*, when the design was at cell level. Figure 25 shows some *IRSIM* results.

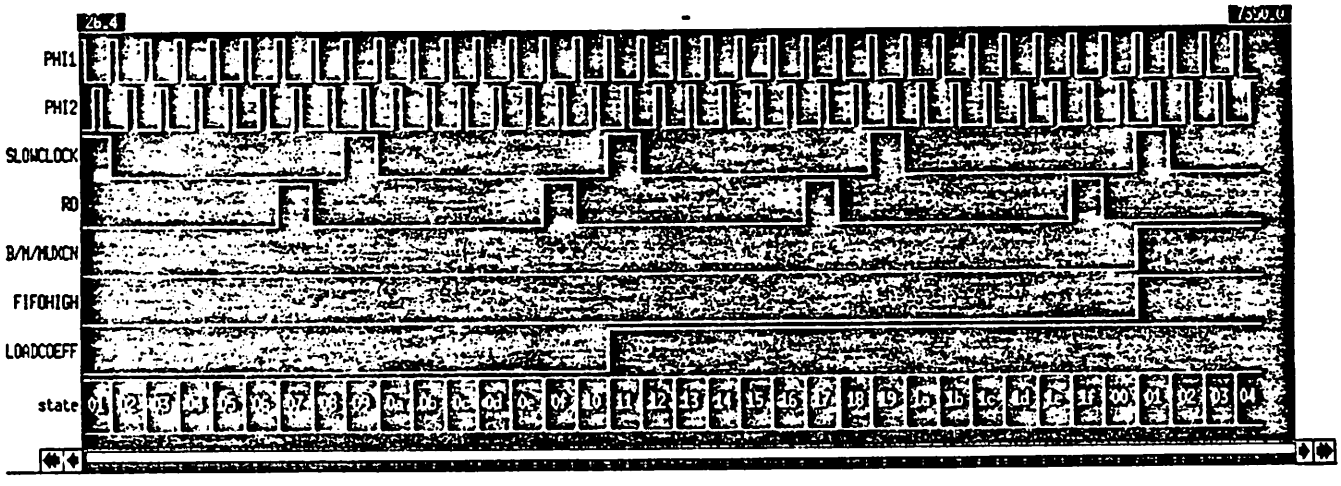
To further ensure that the custom cells were functional, a tiny-chip was fabricated containing the custom cells, (see appendix for a plot of the chip). The C^2MOS shift registers and the multiplexers worked at a 25MHz clock speed.

8.2 Testing

The chip was tested using the Tektronics 920IT Digital Analysis System. The chip was fully functional at 20MHz, but worked at 25MHz only with a supply voltage higher than 5 volts. The cause for its not working at above 25 MHz has been identified.



(a) 512 tap mode



(b) Coefficient Loading

Figure 25: IRSIM Results

The accumulator at the end of the adder-tree is the critical path which fails above 25 MHz. Unfortunately this error was not detected during the design phase. A detailed description of the critical path is shown in figure 26.

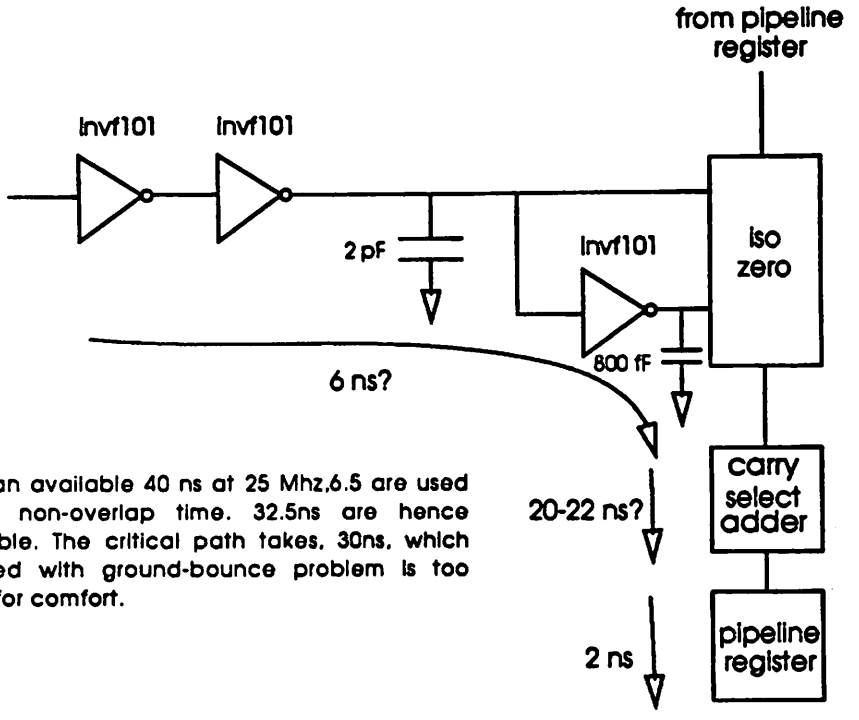


Figure 26: Critical Path

This problem can be solved in two ways. First is by adding another level of pipeline between the zero multiplexer and the adder. The other option is to use a 1.2 mm technology in which the problem will get solved due to a higher circuit speed.

9 Summary

The chip summary is as follows:

TABLE 3. Chip Summary

Chip Area	13.2 mm x 12.2 mm
Chip Area	160.5 sq mm
Speed	25 MHz
Technology	2.0 micron pwell
Pads	132
Signal Pads	80
Transistors	152,504
Power	1-1.5 watts at 25 MHz

Bibliography

[1] Backer, D. C., and S. R. Kulkarni, A New Class of Pulsars, *Physics Today*, 26-35, March, 1990.

[2] Kulkarni, S. R., Project Description and Scientific Justification for a wide band pulsar processor, *California Institute of Technology Radio Astronomy Laboratory internal report*, 1988.

[3] Backer, D. C., Coherent Dispersion Processor, *University of California, Berkeley Radio Astronomy Department internal report*, 1989.

[4] Hankins, T. H., and B. J. Rickett, Pulsar Signal Processing, *Methods in Computational Physics*, Vol. 14, 55, 1975.

[5] Hankins, T. H., and J. M. Rajkowski, A Wide Bandwidth Signal Processor for Removing Dispersion Distortion from Pulsar Radio Signals, *Rev. Sci. Instr.*, 58, 674 - 681, 1987.

[6] Golla, Carla et al., A 30M Samples/s Programmable Filter Processor, *1990 IEEE International Solid-State Circuits Conference*, pp. 116-117.

[7] LSI logic Data Book

[8] Lee, E. A., and Messerschmitt D. G., Digital Communication, Kluwer Academic Publishers, Boston, 1988.

[9] Yeung, A. K. W., VLSI Implementation of a Configurable Multi-Processor System for DSP Behavioral Simulation, *University of California, Berkeley Electronics Research Laboratory memorandum No. UCB/ERL M90/15*, 1990, p. 23.

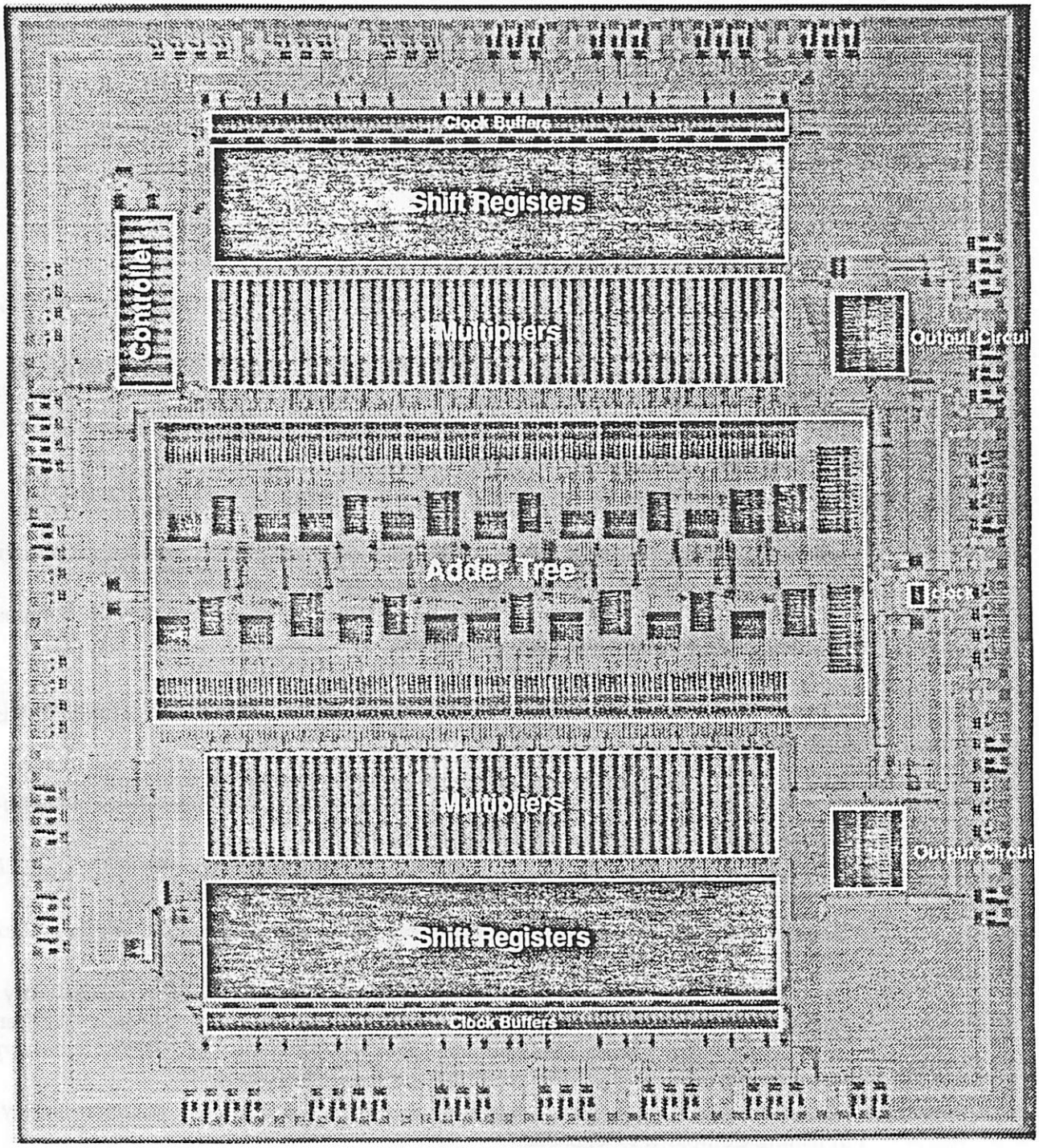
[10] Shung, C. B., et. al., An Integrated CAD System for Algorithm-Specific IC Design, *IEEE Transactions of Computer-Aided Design*, Vol. 10, No. 4, April 1991.

[11] Thompson A. R., J. M. Moran, and G. W. Swenson, Inferometry and Synthesis in Radio Astronomy, John Wiley and Sons, 1986, pp. 210-246.

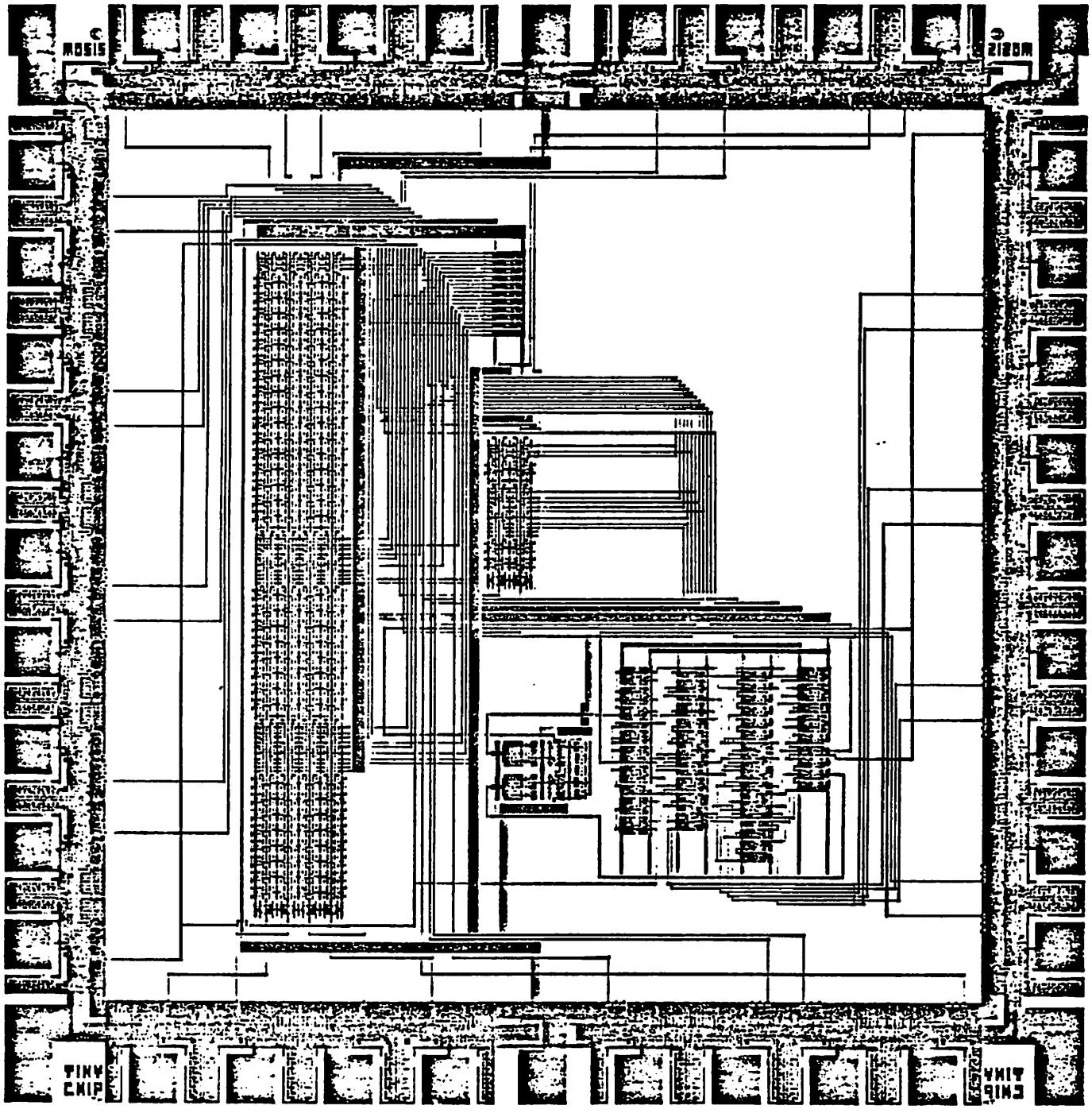
[12] Black, P., I. Linscott, J. Burr, and A. Peterson, Initial Investigation into the Feasibility of using VLSI Technology for the Implementation of a High Performance Matched Filter Processor for Pulsar Timing Measurements, *Stanford University Radio Science Laboratory internal report*, 1989.

[13] Bakoglu, H. B., Circuits, Interconnections, and Packaging for VLSI, Addison-Wesley Publishing Company, 1990, p. 322.

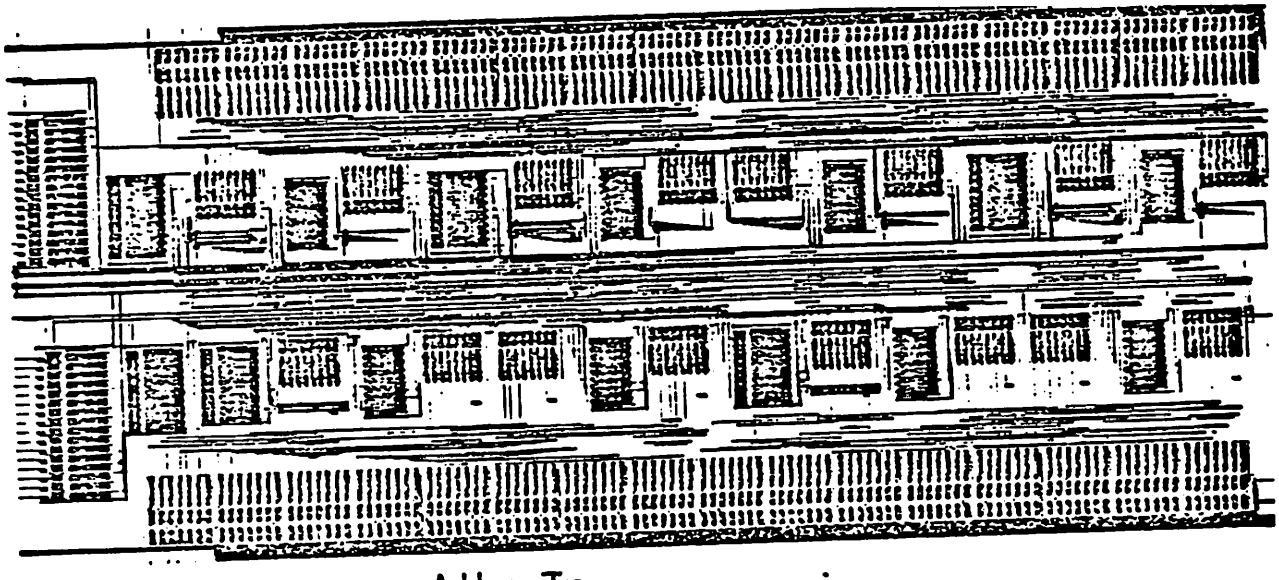
A. Chip Photograph



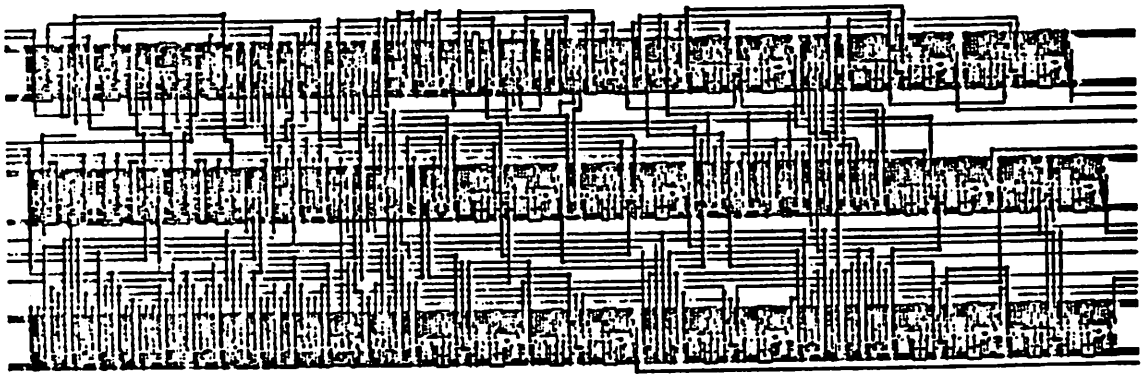
B. Tinychip Plot



C. Other circuits

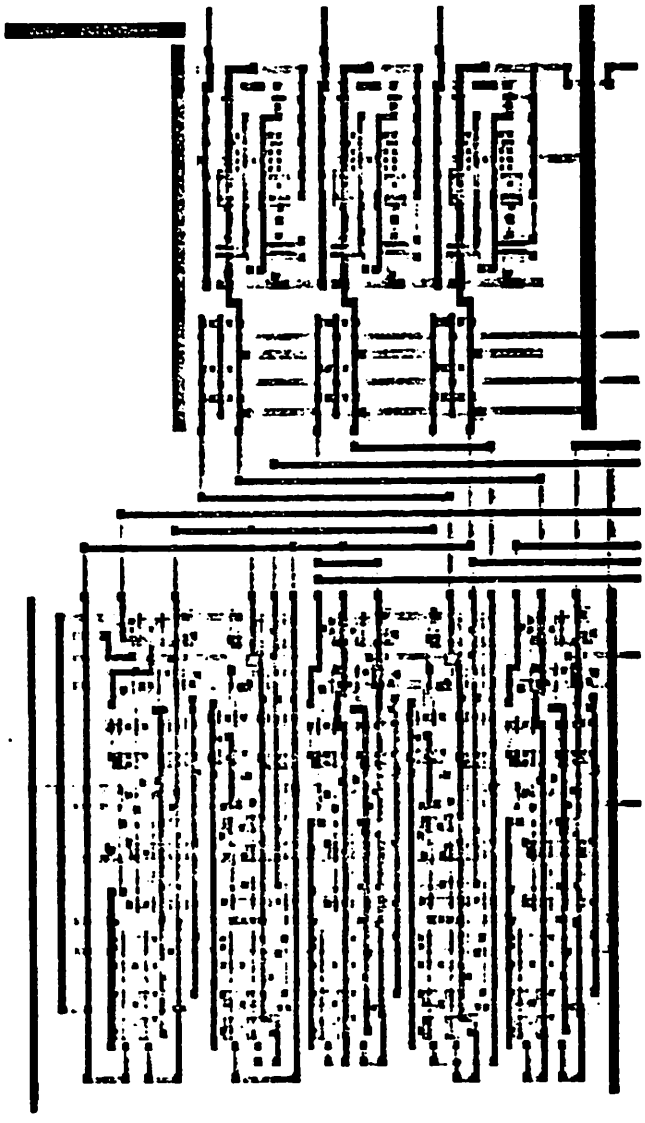


Adder Tree

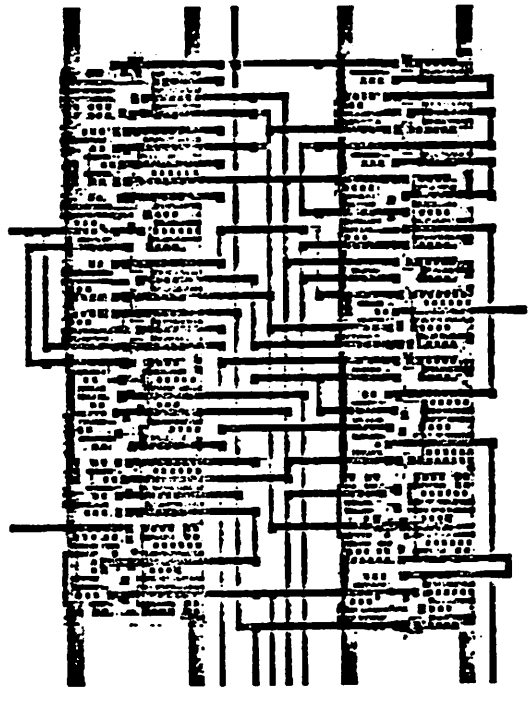


Multiplier - Complex

Shift Register cell →



datapath compiler
multiplier



standard cell
multiplier

fir.sil

```
/*****************************************************************/
/* dwd is the number of bits of data, cwd is the bits in the     */
/* coefficients, and owd is the number of output bits            */
/* taps.h contains the number of Taps required in the FIR        */
/* quant.h contains the precision of the data,coeffs, etc.       */
/* The In file contains the input data and the Out file the      */
/* Output data. The Rcoef and Icoef files contain the coeffs.    */
/*****************************************************************/
#include "taps.h"
#include "quant.h"
func main(In: dwd[2]; Rcoef, Icoef: cwd[Taps]) Result: owd =
begin
    Rdat = In[0];
    Idat = In[1];
    Rtem[0] = owd(Rdat * Rcoef[0] - Idat * Icoef[0]);
    Item[0] = owd(Rdat * Icoef[0] + Idat * Rcoef[0]);
    (i: 1 .. Taps_1)::
    begin
        Rtem[i] = owd(Rtem[i-1]+owd(Rdat@i * Rcoef[i]) - owd(Idat@i * Icoef[i]));
        Item[i] = owd(Item[i-1]+owd(Rdat@i * Icoef[i]) + owd(Idat@i * Rcoef[i]));
    end;
    Result = owd(owd(Rtem[Taps_1] * Rtem[Taps_1]) + owd(Item[Taps_1] *Item[Taps_1]));
end;
```

fir.com

```
INPUT FILE In pulsar.data;
INPUT FILE CONSTANT Rcoef iism.rc;
INPUT FILE CONSTANT Icoef iism.ic;
DISPLAY main Result fir.out;
STEP 352100;
GO;
```

quant.h

```
#define owd fix<8,0>
#define cwd fix<2,0>
#define dwd fix<2,0>
#define cwb 2
#define dwb 2
```

taps.h

```
#define Taps 835
#define Taps_1 834
```

storage_16.c

```
/*This program mimics the storage elements for the FIR chip */
/*Instead of 32 shift registers in 16:8:8 configuration,     */
/*this program simulates 8:4:4, but that should not be a     */
/*problem. There is also an extra register corresponging to */
/*the 33rd extra register in the actual chip. One slow clock*/
/*in this case is therefore 16 as opposed to 32.             */

# include <stdio.h>
```

main

```
main()
{
int data[3][17],coeff[3][16];
int i,j,clock,eval,num,dtmp1,dtmp2,ctmp1,ctmp2;

 /*right now just try 16 register mode                      */
 /*eval = clocks evaluated already                          */
 /*num = number to be entered as new data*/
 /*initialize the coefficient columns                       */
 /*initialize the data columns to 0                         */
 /*and finally print both columns                           */

eval = 0;
num = 100;
for(j=0;j<16;coeff[1][j]=16-j,coeff[2][j]=coeff[1][j]+16,j++);
coeff[1][0]=0;coeff[2][0]=16;
for(j=0;j<17;data[1][j]=data[2][j++]=0);
printf("\t%d\n\n",data[2][16]);
for(j=0;j<16;j++) printf("%d\t%d\t\t%d\t%d\n",data[1][15-j],data[2][15-j]
        ,coeff[1][15-j],coeff[2][15-j]);
printf("\n");

 /* enter into the normal operation                          */
 /* take as input the number of clock      */
 /* to be executed (fast clocks)                             */

while (scanf("%d",&clock) != EOF){
        for(i=clock;i>0;i--) {

 /*case 1 is where the 'special' 0,16,32 etc clock cycle happens*/
 /*the rest are normal cycles. The normal cycle just circulates */
 /*while the special cycle interchanges data, but circulates coe*/

                if((eval /16)*1000==(int)(((float) eval) /0.016)) {
                        for(j=0;j<15;j++) data[2][j]=data[2][j+1];
                        data[2][15] = data[2][16];
                        data[2][16] = data[1][1];
                        for(j=0;j<15;j++) data[1][j]=data[1][j+1];
                        data[1][15] = data[1][16];
                        data[1][16] = num--;
                        ctmp1 = coeff[1][0];
                        ctmp2 = coeff[2][0];
                        for(j=0;j<15;j++) coeff[1][j]=coeff[1][j+1];
                        for(j=0;j<15;j++) coeff[2][j]=coeff[2][j+1];
                        coeff[1][15] = ctmp1;
                        coeff[2][15] = ctmp2;
                        }
                else {
                        dtmp1 = data[1][1];
                        dtmp2 = data[2][1];
                        for(j=0;j<15;j++) data[1][j]=data[1][j+1];
                        for(j=0;j<15;j++) data[2][j]=data[2][j+1];
                        data[1][15] = dtmp1;
                        data[2][15] = dtmp2;
```

storage_16.c

...main

```
                        ctmp1 = coeff[1][0];
                        ctmp2 = coeff[2][0];
                        for(j=0;j<15;j++) coeff[1][j]=coeff[1][j+1];
                        for(j=0;j<15;j++) coeff[2][j]=coeff[2][j+1];
                        coeff[1][15] = ctmp1;
                        coeff[2][15] = ctmp2;
                        }
                eval++;
                }
        printf("\t%d\n\n",data[2][16]);
        for(j=0;j<16;j++) printf("%d\t%d\t\t%d\t%d\n",data[1][15-j],data[2][15-
                j],coeff[1][15-j],coeff[2][15-j]);
        printf("\n");
        }
}
```

```
/*This program mimics the storage elements for the FIR chip */
/*Instead of 32 shift registers in 16:8:8 configuration,    */
/*this program simulates 8:4:4, but that should not be a    */
/*problem. There is also an extra register corresponging to */
/*the 33rd extra register in the actual chip. One slow clock*/
/*in this case is therefore 16 as opposed to 32.            */
# include <stdio.h>
main()                                                          main
{
int data[3][17],coeff[3][16];
int i,j,clock,eval,num,dtmp1,dtmp2,ctmp1,ctmp2;


/*right now just try 8 register mode                          */
/*eval = clocks evaluated already                             */
/*num = number to be entered as new data*/
/*initialize the coefficient columns                          */
/*initialize the data columns to 0                            */
/*and finally print both columns                              */

eval = 0;
num = 100;
for(j=0;j<8;coeff[1][j]=coeff[1][j+8]=8-j,coeff[2][j]=coeff[2][j+8]=coeff[1][j]
        +8,j++);
coeff[1][0]=coeff[1][8]=0;coeff[2][0]=coeff[2][8]=8;
for(j=0;j<17;data[1][j]=data[2][j++]=0);
printf("\t%d\n\n",data[2][16]);
for(j=0;j<16;j++) printf("%d\t%d\t\t%d\t%d\n",data[1][15-j],data[2][15-j]
        ,coeff[1][15-j],coeff[2][15-j]);
printf("\n");

/* enter into the normal operation                            */
/* take as input the number of clock      */
/* to be executed (fast clocks)                               */

while (scanf("%d",&clock) != EOF){
        for(i=clock;i>0;i--) {

/*case 1 is where the 'special' 0,8,16 etc clock cycle happens*/
/*the rest are normal cycles. The normal cycle just circulates */
/*while the special cycle interchanges data, but circulates coe*/

                if((eval /8)*1000==(int)(((float) eval) /0.008)) {
                        for(j=0;j<15;j++) data[2][j]=data[2][j+1];
                        data[2][15] = data[1][1];
                        for(j=0;j<15;j++) data[1][j]=data[1][j+1];
                        data[1][15] = num--;
                        ctmp1 = coeff[1][0];
                        ctmp2 = coeff[2][0];
                        for(j=0;j<15;j++) coeff[1][j]=coeff[1][j+1];
                        for(j=0;j<15;j++) coeff[2][j]=coeff[2][j+1];
                        coeff[1][15] = ctmp1;
                        coeff[2][15] = ctmp2;
                        }
                else {
                        dtmp1 = data[1][1];
                        dtmp2 = data[2][1];
                        for(j=0;j<15;j++) data[1][j]=data[1][j+1];
                        for(j=0;j<15;j++) data[2][j]=data[2][j+1];
                        data[1][7] = dtmp1;
                        data[2][7] = dtmp2;
                        ctmp1 = coeff[1][0];
                        ctmp2 = coeff[2][0];
                        for(j=0;j<15;j++) coeff[1][j]=coeff[1][j+1];
```

storage_8.c

...main

```
                        for(j=0;j<15;j++) coeff[2][j]=coeff[2][j+1];
                        coeff[1][15] = ctmp1;
                        coeff[2][15] = ctmp2;
                        }
                eval++;
                }
        printf("\t%d\n\n",data[2][16]);
        for(j=0;j<16;j++) printf("%d\t%d\t\t%d\t%d\n",data[1][15-j],data[2][15-
                j],coeff[1][15-j],coeff[2][15-j]);
        printf("\n");
        }
}
```

```
/*This program mimics the storage elements for the FIR chip */
/*Instead of 32 shift registers in 16:8:8 configuration,     */
/*this program simulates 8:4:4, but that should not be a     */
/*problem. There is also an extra register corresponging to */
/*the 33rd extra register in the actual chip. One slow clock*/
/*in this case is therefore 16 as opposed to 32.             */
# include <stdio.h>
main()                                                          main
{
int data[3][17],coeff[3][16];
int i,j,clock,eval,num,dtmp1,dtmp2,ctmp1,ctmp2;


/*right now just try 4 register mode                          */
/*eval = clocks evaluated already                             */
/*num = number to be entered as new data*/
/*initialize the coefficient columns                          */
/*initialize the data columns to 0                            */
/*and finally print both columns                              */

eval = 0;
num = 100;
for(j=0;j<4;coeff[1][j]=coeff[1][j+4]=coeff[1][j+8]=coeff[1][j+12]=4-j,
        coeff[2][j]=coeff[2][j+4]=coeff[2][j+8]=coeff[2][j+12]=coeff[1][j]
        +4,j++);
coeff[1][0]=coeff[1][4]=coeff[1][8]=coeff[1][12]=0;
coeff[2][0]=coeff[2][4]=coeff[2][8]=coeff[2][12]=4;
for(j=0;j<17;data[1][j]=data[2][j++]=0);
printf("\t%d\n\n",data[2][16]);
for(j=0;j<16;j++) printf("%d\t%d\t\t%d\t%d\n",data[1][15-j],data[2][15-j]
        ,coeff[1][15-j],coeff[2][15-j]);
printf("\n");

/* enter into the normal operation                            */
/* take as input the number of clock      */
/* to be executed (fast clocks)                               */

while (scanf("%d",&clock) != EOF){
        for(i=clock;i>0;i--) {

/*case 1 is where the 'special' 0,4,8 etc clock cycle happens*/
/*the rest are normal cycles. The normal cycle just circulates */
/*while the special cycle interchanges data, but circulates coe*/

                if((eval /4)*1000==(int)(((float) eval) /0.004)) {
                        for(j=0;j<15;j++) data[2][j]=data[2][j+1];
                        data[2][7] = data[1][1];
                        for(j=0;j<15;j++) data[1][j]=data[1][j+1];
                        data[1][7] = num--;
                        ctmp1 = coeff[1][0];
                        ctmp2 = coeff[2][0];
                        for(j=0;j<15;j++) coeff[1][j]=coeff[1][j+1];
                        for(j=0;j<15;j++) coeff[2][j]=coeff[2][j+1];
                        coeff[1][15] = ctmp1;
                        coeff[2][15] = ctmp2;
                        }
                else {
                        dtmp1 = data[1][1];
                        dtmp2 = data[2][1];
                        for(j=0;j<15;j++) data[1][j]=data[1][j+1];
                        for(j=0;j<15;j++) data[2][j]=data[2][j+1];
                        data[1][3] = dtmp1;
                        data[2][3] = dtmp2;
                        ctmp1 = coeff[1][0];
```

...main

```
                                ctmp2 = coeff[2][0];
                                for(j=0;j<15;j++) coeff[1][j]=coeff[1][j+1];
                                for(j=0;j<15;j++) coeff[2][j]=coeff[2][j+1];
                                coeff[1][15] = ctmp1;
                                coeff[2][15] = ctmp2;
                                }
                        eval++;
                        }
                printf("\t%d\n\n",data[2][16]);
                for(j=0;j<16;j++) printf("%d\t%d\t\t%d\t%d\n",data[1][15-j],data[2][15-
                        j],coeff[1][15-j],coeff[2][15-j]);
                printf("\n");
                }
}
```

UCB

1024 Tap FIR Filter

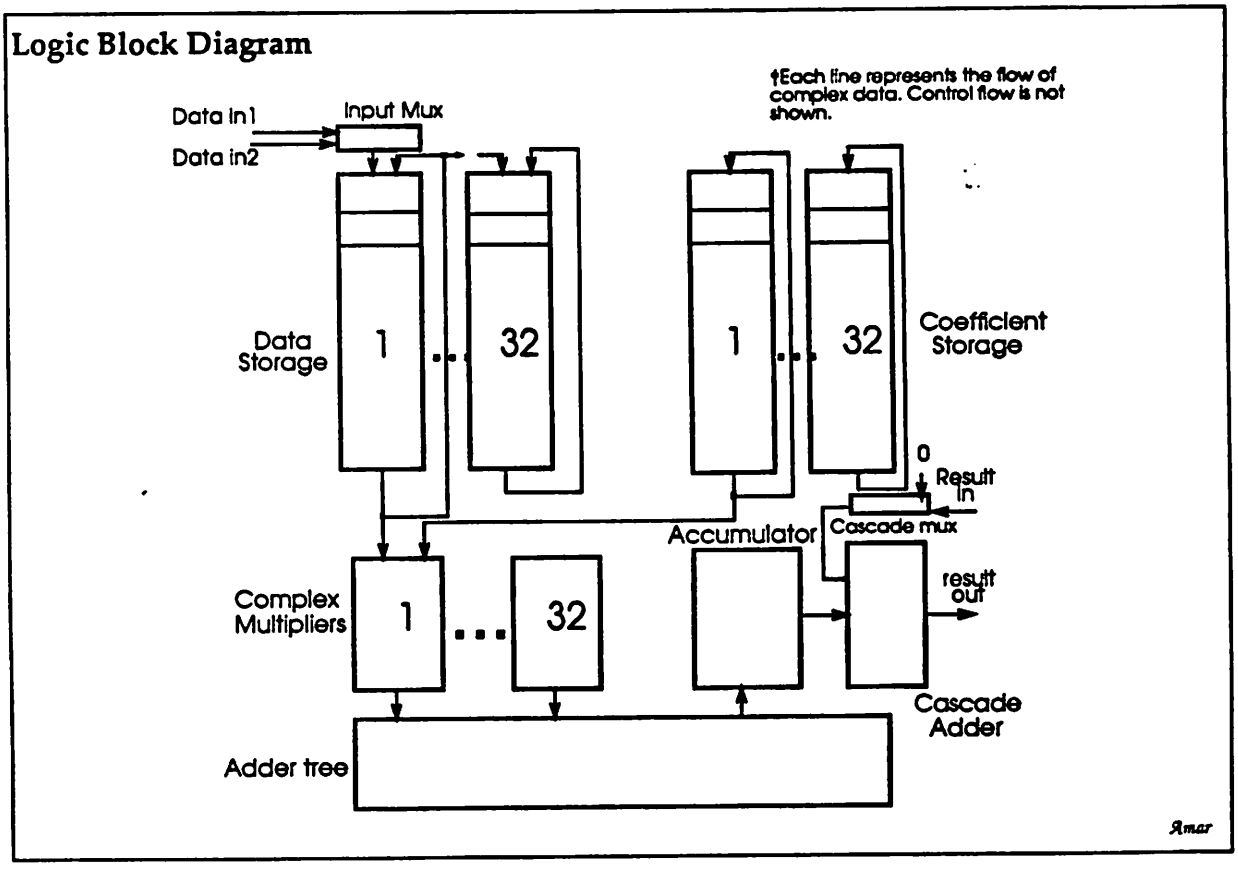
Features

- 1024 taps 1 kHz - 1 MHz
- 512 taps 2 kHz - 2 MHz
- 256 taps 4 kHz - 4 MHz
- Programmable coefficients
- Cascadable
- 2 µm pwell CMOS process

Functional Description

The PULFIR1 is a 1024 tap FIR filter. It can also function in 512 and 256 tap modes. The maximum data rate processed is 4 MHz, but the product of taps × frequency cannot exceed 1024 MHz-taps. The coefficients are programmable but not adaptive. The data and coefficients are *complex* numbers. Data is (2,2) bits and coefficients are (3,3) bits. The data is encoded as ±3, ±1 and the coefficients are encoded as ±3, ±2, ±1, and 0. It is also possible to cascade more than one PULFIR1 chips.

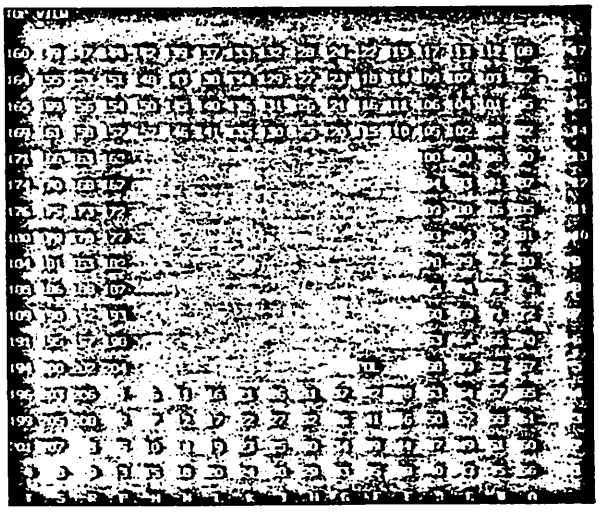
Logic Block Diagram



Pin Configuration

The package used is a 208 pin Kyrocera PGA package. The pin out is as follows:

1	unused	53	unused	105	unused	157	unused
2	unused	54	unused	106	unused	158	unused
3	unused	55	unused	107	unused	159	unused
4	unused	56	unused	108	unused	160	unused
5	unused	57	unused	109	unused	161	unused
6	unused	58	unused	110	unused	162	unused
7	unused	59	unused	111	unused	163	unused
8	unused	60	unused	112	unused	164	unused
9	unused	61	unused	113	unused	165	unused
10	unused	62	unused	114	COEFOUT[0]	166	RRES[1]
11	substrate=Vdd	63	substrate=Vdd	115	substrate=Vdd	167	substrate=Vdd
12	IRES[11]	64	DATA1[3]	116	COEFOUT[1]	168	RRES[0]
13	IRES[10]	65	DATA1[2]	117	COEFOUT[5]	169	GND
14	IRES[9]	66	DATA1[1]	118	COEFOUT[4]	170	Vdd
15	GND	67	DATA1[0]	119	GND	171	LRIN
16	Vdd	68	GND	120	Vdd	172	REALOE
17	IRES[8]	69	Vdd	121	COEFOUT[3]	173	CKSEL
18	IRES[7]	70	STATE[4]	122	COEFOUT[2]	174	PHI2
19	IRES[6]	71	STATE[3]	123	RRES[17]	175	PHI1
20	GND	72	STATE[2]	124	RRES[16]	176	PHI2IN
21	Vdd	73	STATE[1]	125	GND	177	GND
22	IRES[5]	74	GND	126	Vdd	178	Vdd
23	IRES[4]	75	Vdd	127	RRES[15]	179	PHI1IN
24	IRES[3]	76	STATE[0]	128	RRES[14]	180	CLK
25	GND	77	FIFOHIGH	129	RRES[13]	181	SHIFTINV
26	Vdd	78	INIT	130	GND	182	SHIFT
27	IRES[2]	79	CASCADE	131	Vdd	183	GND
28	IRES[1]	80	GND	132	RRES[12]	184	Vdd
29	IRES[0]	81	Vdd	133	RRES[11]	185	SCANOUT
30	GND	82	IMODE[1]	134	RRES[10]	186	SCANIN
31	Vdd	83	IMODE[0]	135	GND	187	LIIN
32	COEFIN[0]	84	SOURCESEL	136	Vdd	188	IMAGOE
33	COEFIN[1]	85	LOADCOEFF	137	RRES[9]	189	GND
34	COEFIN[2]	86	GND	138	RRES[8]	190	Vdd
35	GND	87	Vdd	139	RRES[7]	191	IRES[17]
36	Vdd	88	RO	140	GND	192	IRES[16]
37	COEFIN[3]	89	SLOWCLOCK	141	Vdd	193	IRES[15]
38	COEFIN[4]	90	DATAOUT[3]	142	RRES[6]	194	GND
39	COEFIN[5]	91	GND	143	RRES[5]	195	Vdd
40	GND	92	Vdd	144	RRES[4]	196	IRES[14]
41	Vdd	93	DATAOUT[2]	145	GND	197	IRES[13]
42	DATA2[2]	94	DATAOUT[1]	146	Vdd	198	IRES[12]
43	DATA2[0]	95	DATAOUT[0]	147	RRES[3]	199	unused
44	DATA2[3]	96	unused	148	RRES[2]	200	unused
45	DATA2[1]	97	unused	149	unused	201	unused
46	unused	98	unused	150	unused	202	unused
47	unused	99	unused	151	unused	203	unused
48	unused	100	unused	152	unused	204	unused
49	unused	101	unused	153	unused	205	unused
50	unused	102	unused	154	unused	206	unused
51	unused	103	unused	155	unused	207	unused
52	unused	104	unused	156	unused	208	unused



The package dimensions are:

1.69350 $^{\pm 0.020}$ inches on each side.

Special static protection for the user required for this chip.

Pin Description

COEFIN[5-0] [I]: This is the bus for loading in coefficients. As a convention, the higher significant bits are for imaginary, and lower for real.

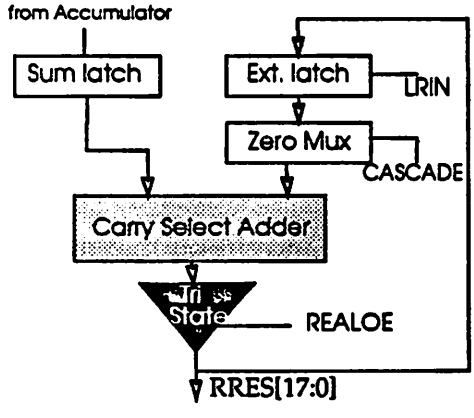
DATA1[3:0] [I]: This is one of the two busses to input the data.

DATA2[3:0] [I]: This is the second bus to input data. The SOURCESEL pin controls which of the two sources gets selected.

DATAOUT[3:0] [O]: This bus is the data output from the memory section. It is useful for cascading. The output data is latched and is stable for one entire slow-clock. Notice that this also introduces a 'delay' of one clock cycle between chips which must be compensated for.

COEFOUT[5:0] [O]: This bus is the coefficient output from the memory section. It is not latched and will constantly be changing.

RRES[17:0] [I/O]: This bus is the 18 bit real bus from the chip. When the REALOE pin is asserted, the chip outputs the 18 bit result, while when REALOE is not asserted, this bus is in input state. See the output circuitry figure shown below for a clear understanding of this bus. This bus does not have input static protection, and these pins must not be exposed to static.



Output Circuitry (for real result)

IRES[17:0] [I/O]: This bus is the imaginary complement of the RRES bus.

LOADCOEFF [I]: This control signal requests the FIR chip to load coefficients. The request signal must be asserted *high* and must be held till an acknowledgment is received. Also, this signal must be pulled *low* before coefficient loading is over, else the chip will re-enter the loading state.

FIFOHIGH [O]: This control signal an acknowledgment in response to the LOADCOEFF signal. This signal will be asserted *high* as an acknowledgment. The FIFO or memory providing the coefficients must use this signal and the clock to strobe out the coefficients.

RO [O]: This control signal is pulsed *high* every time a "new" output is available. The value is stable internally by the rising edge of RO, but may be available outside the chip after some delay.

SLOWCLOCK [O]: This control signal is pulsed high when a "new" data is read from the data bus. In other words, the external data must be stable during the presence of this pulse. This pulse, or alternatively RO may also be used to synchronize more than one chip. A clever design may use RO for the same purpose as SLOWCLOCK, and hence obviate this signal.

CASCADE [I]: This input control signal is "static" in that it must be held to a constant value during the entire operation period of the chip. This signal should be *low* for stand-alone and *high* for cascade operation.

LRIN [I]: This input control signal (see output circuitry) for loading in a value on to the RRES bus for cascade operation. This signal should be pulsed *high* for loading. Unfortunately, the latch associated with the loading operation is synchronous with the internal clock, which adds the constraint that the LRIN must be asserted for at least $1\frac{1}{2}$ clock cycles, which for 32 MHz translates to about 47ns.

LIIN [I]: This input control signal is the imaginary complement of LRIN.

CLK [I]: This is the input clock. It must be ha[illegible] a duty cycle of 50% and effects of cl[illegible]itter are not clear as of now.

IMODE[1:0] [I]: These two pins control the mode in which the chip will operate. It is recommended that these pins be used as "static control" lines as well. The various modes are

00: 256 tap mode
01: 512 tap mode
10: 1024 tap mode
11: For test purpose only

REALOE [I]: This input control signal sets the RRES bus in input or output mode. It is asserted *high* for configuring the bus in output mode, and *low* for configuring it in input mode. See the output circuit diagram for clarity.

IMAGOE [I]: This input control signal is the imaginary complement of REALOE.

SOURCESEL [I]: Another "static" input control line, this is to be kept *low* for selecting data on bus DATA1 and *high* for selecting bus DATA2.

INIT [I]: This signal is to pulsed high for resetting the controller on chip. Once the "static" control lines have been set up, this line should be pulsed.

STATE[4:0] [O]: This bus is purely for testing purpose and indicated the state of the controller.

PHI2, PHI1 [O]: These clock outputs are for testing purpose and are the internal two phase clocks.

CKSEL [I]: This controls the source of the clock, i.e. when it is *low*, it allows external PHI1, PHI2 inputs, when it is *high*, it uses the internal clock generator and the CLK input.

PHI1IN, PHI2IN [I]: These clock inputs are for testing purpose. In case there is a problem with the internal clock generator, an external two phase clock could be supplied.

SHIFT, SHIFTINV, SCANIN [I]: These clock inputs are purely for testing. Shift should be tied low, shiftinv low, and scanin low.

SCANOUT [O]: This is purely for testing.

Vdd [Supply]: These are 5 volt power lines.

GND [Ground]: These are ground lines.

Functional Description

Stand-alone mode

1: Set up the static inputs, the mode, cascade line, and the sourcesel line.

2: Toggle the init line.

3: Make a request for loading the coefficients through the loadcoeff line.

4: After the acknowledge through fifohigh, remove loadcoeff request.

5: Using fifohigh and the clock, strobe the FIFO/RAM which feeds coefficients to the chip.

6: Ramp up the chip clock to operational mode.

7: Once in normal operation, supply stable data during slowclock.

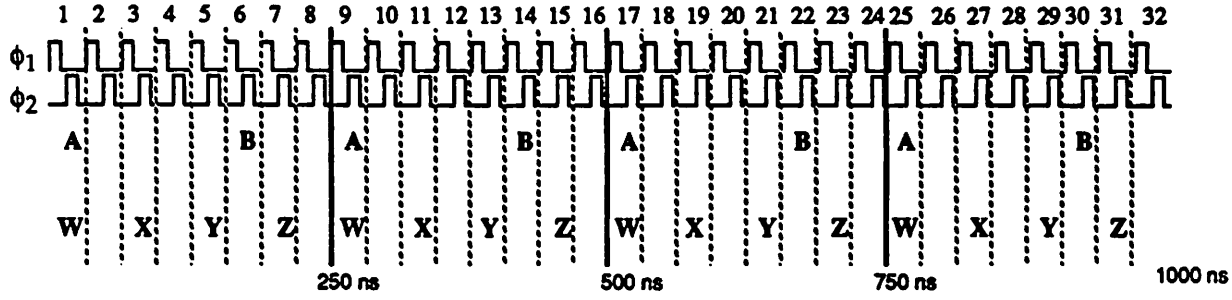
8: During normal operation, also read the output once RO has occurred.

Cascade Mode

1: It is possible to go through the process of loading the coefficients more than once, to load up more than one chip in cascade.

2: Since there is a delay of one slow clock cycle in the data transfer from

one chip to the next, this *must* be compensated in the summation of all the results. A proposed scheme could be:



A - SLOWCLOCK

B - RO, data in the sum latch (see output circuitry) is latched

W - If chip 4, latch in result. If chip3, output result

X - If chip 3, latch in result. If chip 2, output result

Y - If chip 2, latch in result. If chip 1, output result

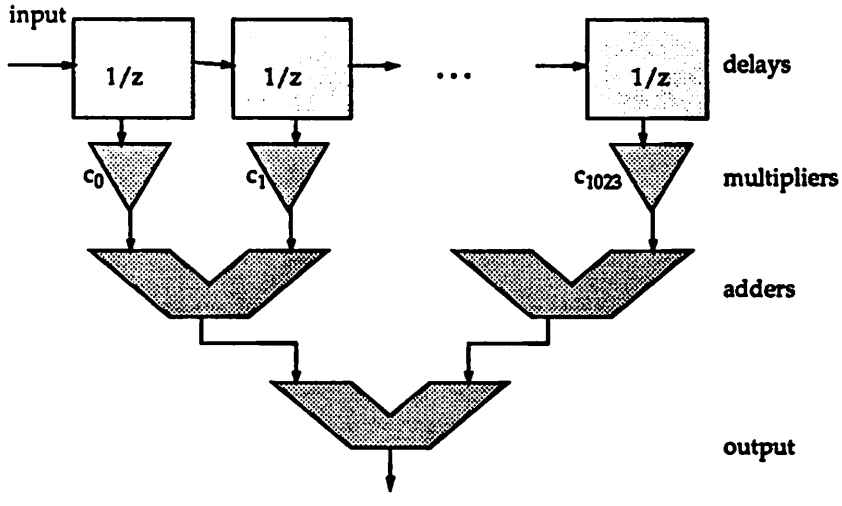
Z - If chip 4 output result

4 Chips in 256 Tap Mode

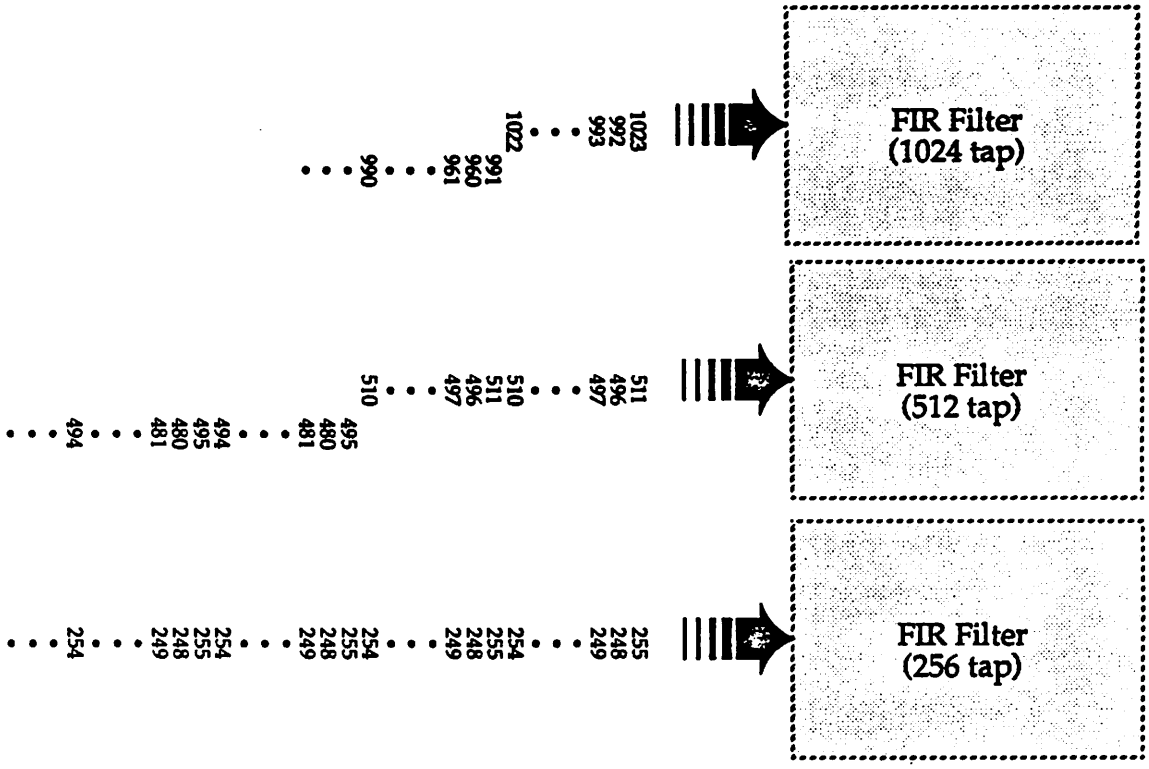
Coefficient Loading

The order in which coefficients are loaded is slightly unusual, and requires a closer look. Assume the 1024 tap FIR filter below:

The coefficients should be loaded up as shown in the diagrams below. The 1024 coefficients are loaded in 32 blocks of 32



coefficients with the last block entering first. Within each block the loading sequence is: N, N-31, N-30, ..., N-1. For the 512 and 256 tap FIR modes the coefficients are duplicated and quadruplicated, respectively, with the intra-block loading sequence being: N, N-15, N-14, ..., N-1, N, N-15, N-14, ..., N-1 for the 512 tap mode; and N, N-7, N-6, ..., N-1, N, N-7, N-6, ..., N-1, N, N-7, N-6, ..., N-1, N, N-7, N-6, ..., N-1.



Data and Coefficient Encoding

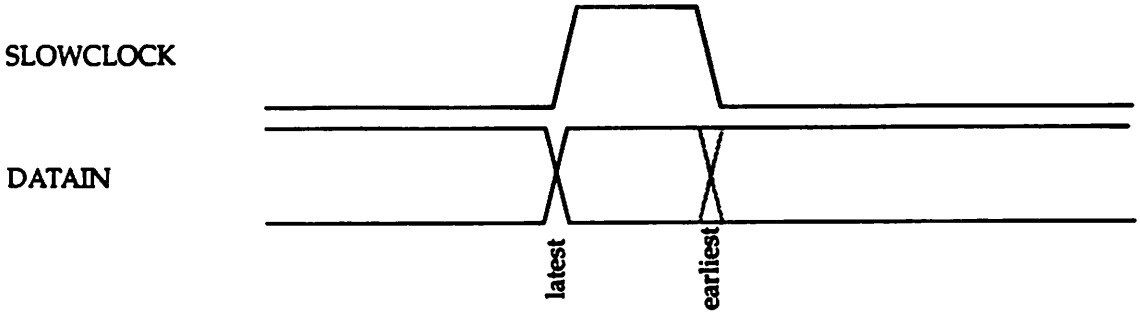
The encoding scheme is as follows: (Notice that the coefficients are encoded in plain two's complement)

TABLE 1. Coding

Data	Value	Coefficient	Value	Coefficient	Value
00	1	000	0	100	x
01	3	001	+1	101	-3
10	-1	010	+2	110	-2
11	-3	011	3	111	-1

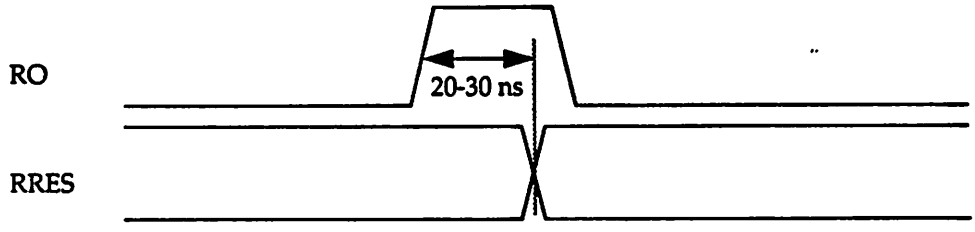
Switching Waveforms:

1. Write



2. Read

- Delay after RO



- Delay after REALOE: undetermined
- Delay after LRIN: undetermined

3. Generating strobe for coefficient FIFO