AUTOMATED LOW-POWER ASIC DESIGN FOR SPEECH PROCESSING

by

Engling Yeo

Memorandum No. UCB/ERL M95/99

20 December 1995

AUTOMATED LOW-POWER ASIC DESIGN FOR SPEECH PROCESSING

by

Engling Yeo

Memorandum No. UCB/ERL M95/99

20 December 1995

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Automated Low-Power ASIC Design For Speech Processing

Engling Yeo

ABSTRACT

Recent developments in Hyper's design space exploration and high-level synthesis techniques have brought the realization of automated synthesis of memory-intensive, low-power implementations closer. This work describes the design path taken to synthesize the front end of a speech recognition chip. It starts from the Hyper high-level synthesis tool by looking at possible alternatives at the algorithmic level, as well as using suitable transformations on the control flow data graph (CDFG) representation within Hyper. The flowgraph is then mapped onto an architecture, and more power analysis provides feedback for possible power improvements. Finally, with the help of the LagerIV architectural-level tools, a silicon layout suitable for fabrication is generated

Table Of Contents

List of Figures

List of Tables

1.0 Introduction.

The objective of this work is to design the front end of a speech recognition system to handle the acoustic signal processing. The algorithm adopts a proposal by Hermansky et. al. [6], RASTA-PLP, which is a refinement over the standard linear prediction (LPC) model.

Speech-recognition systems comprise a wide collection of disciplines, including statistical pattern recognition, communication theory, signal processing, combinatorial mathematics, and linguistics, amongst others. Typically speech recognition starts with the digital sampling of speech. The next stage is the acoustic signal processing, which converts the speech waveform to some type of parametric representation. Most techniques include spectral analysis; e.g. LPC analysis, MFCC, cochlea modeling and many, many more. The next stage is recognition of phonemes, groups of phonemes and words. This stage can be achieved by many processes such as DTW (Dynamic Time Warping) [2], HMM (hidden Markov modelling)[2], NNs (Neural Networks)[2], expert systems [2] and combinations of techniques. HMM-based systems are currently the most commonly used and most successful approach.

Following the current wave of interest in low-power portable applications such as Berkeley's personal communications terminal, InfoPad, the power issues will be emphasized. A speech recognition system is an excellent substitute for the keyboard on portable devices that require human input.

Hyper's [1] hardware library of low-power cells are used in the design. The entire behavioral input description is done in Silage[4]. Employing different flowgraph transformations, various area, speed, and power optimization techniques were explored. The estimates provided by Hyper gave a quick comparison of the relative effectiveness between various sets of transformations.

After the scheduling and hardware mapping steps, architectural power analysis was done using SPA. It identifies the bottlenecks of power consumption. The information is used to suggest further improvements to the architecture.

As a final step, the architectural description is converted to silicon layout using the LagerIV CAD tools. Running IRSIM on the extracted layouts provides some power figures that can be compared with the SPA results.

1.1 Prediction Model Fundamentals

Some details of the algorithm are left out here, and described in more rigorous fashion in Chapter 2. This section is intended to give an overall view of the RASTA-PLP algorithm, leading to the block diagram in Figure 1.4. Chapter 2 will discuss the algorithm used to implement each of the blocks, as well as the design space exploration.

1.1.1 Linear Prediction (LPC) Model

The greatest common denominator of all recognition systems is the acoustic signal processing. A common front-end is the linear predictive coding (LPC) analysis [2]. The analysis assumes that the speech synthesis consists of a digital filter, H(z), whose excitation source is chosen from either an impulse train (voiced speech) or a white noise generator (unvoiced sounds) depending on a voiced/unvoiced switch. The switch is controlled by the voiced/unvoiced character of the speech. The filter has an all-pole transfer function which is controlled by the vocal tract parameters characteristic of the speech being produced. The aim is therefore to predict the a_k coefficients of the filter H(z), also known as the LPC coefficients. Figure 1.1 shows a p^{th} order autoregressive (AR) transfer function, with p poles.

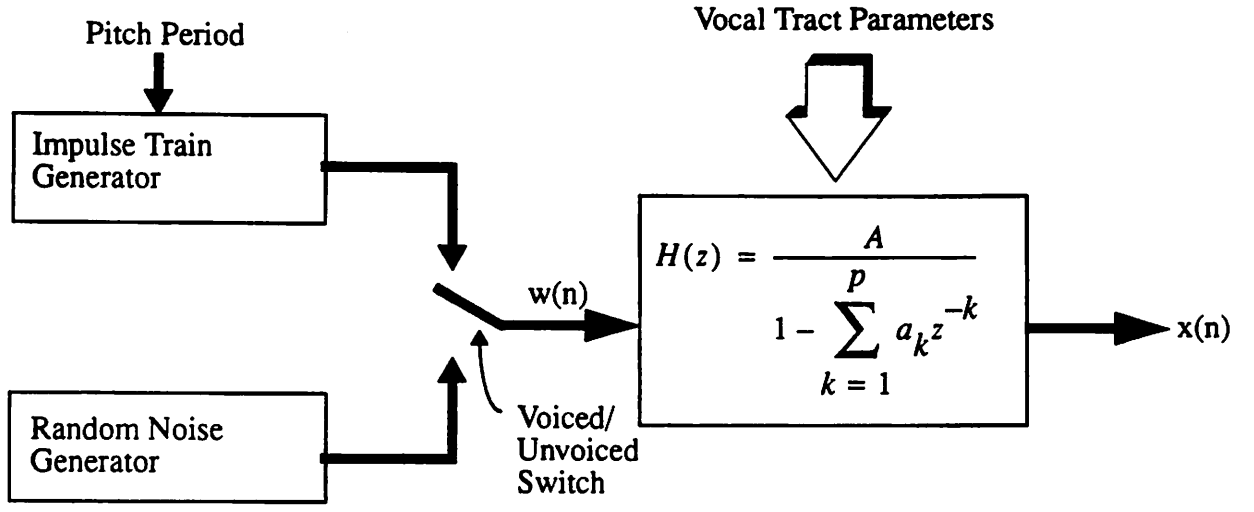


FIGURE 1.1 Speech synthesis model based on LPC Model

From the transfer function, a given speech sample at time n, x(n) can be represented as a linear combination of the past p speech samples,

$$x(n) = Aw(n) + a_1x(n-1) + a_2x(n-2) + \ldots + a_px(n-p)$$

(EQ 1.1)

Hence, the autocorrelation function can be related by

$$\begin{aligned} r_x(k) &= E[x(n)x^*(n-k)] \\ &= E\{[w(n) + a_1x(n-1) + a_2x(n-2) + \ldots + a_px(n-p)]x^*(n-k)\} \\ &= E\{[a_1x(n-1) + a_2x(n-2) + \ldots + a_px(n-p)]x^*(n-k)\} \\ &= a_1r_x(k-1) + a_2r_x(k-2) + \ldots + a_pr_x(k-p) \end{aligned}$$

x^* : complex conjugate of x.

(EQ 1.2)

A p^{th} order AR model has therefore, p unknowns, a_1 through a_p. EQ1.3 shows p sets of EQ1.2 with the value of k ranging from 1 through p arranged in a matrix form. Since the speech samples are real the autocorrelation function is symmetric, i.e. $r_x(k) = r_x(-k)$. If there is a way to estimate the first $p+1$ entries ($r_x(0)$ through $r_x(p)$) of the short term autocorrelation function, the LPC coefficients can then be predicted by solving EQ1.3 with Durbin's algorithm[3] for Yule-Walker equations, since the $p \times p$ matrix is Toeplitz symmetric:

$$\begin{bmatrix} r(0) & r(1) & r(2) & \dots & \dots & r(p-1) \\ r(1) & r(0) & r(1) & \dots & \dots & r(p-2) \\ r(2) & r(1) & r(0) & \dots & \dots & r(p-3) \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & r(0) \\ r(p-1) & r(p-2) & r(p-3) & \dots & \dots & r(0) \end{bmatrix} \bullet \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \dots \\ \dots \\ a_p \end{bmatrix} = \begin{bmatrix} r(1) \\ r(2) \\ r(3) \\ \dots \\ \dots \\ r(p) \end{bmatrix}$$

(EQ 1.3)

The autocorrelation function is obtained by an inverse Fourier transformation of the power spectrum. The short term (20ms~30ms window) spectrum is approximated by a periodogram estimate[4], defined as

$$R_p(e^{j\omega}) = \left| \sum_{n=0}^{N} x(n) e^{-j\omega n} \right|^2 \approx \sum_{n=0}^{N} r_x(n) e^{-j\omega n}$$

(EQ 1.4)

The Fourier transform of the speech samples is usually obtained with a 256-point or 512-point FFT on a speech input sampled between 8kHz to 16kHz. The transfer function of the vocal tract, H(z) is assumed to remain unchanged throughout the short time segment.

A very important LPC parameter set, which can be derived directly from the LPC coefficients, is the set of LPC cepstral coefficients, c_m. These are the coefficients of the Fourier transform representation of the log magnitude spectrum. Bogert et al. [17] observed that the logarithm of the power spectrum of a signal containing an echo hasa n additive periodic component due to

the echo, and thus its Fourier transform, the cepstrum, should exhibit a peak at the echo delay. The cepstral coefficients are associated with homomorphic analysis, which have been shown to be a more robust, reliable feature set for speech recognition than LPC [18].

The recursion used is given by the following equations, with A and a_m variables defined as in EQ1.1 and EQ1.3.

$$c_0 = \ln A^2$$

$$c_m = a_m + \sum_{k=1}^{m-1} \left(\frac{k}{m}\right) c_k a_{m-k} \qquad 1 \le m \le p$$

$$c_m = \sum_{k=1}^{m-1} \left(\frac{k}{m}\right) c_k a_{m-k} \qquad m > p$$

(EQ 1.5)

1.1.2 Bank of Filter Analysis and Perceptual Linear Prediction (PLP)

LPC models all frequencies equally well. Unfortunately, this is not consistent with the human auditory devices. It was shown by Hermansky [5] that recognizers equipped to utilize the perceptually based spectra deliver recognition accuracy comparable to that of LPC, but with smaller order autoregressive models, i.e. fewer LPC coefficients, thus gaining the advantage of a reduced computational load.

The frequency analysis step in human speech processing is performed by the inner ear. It is accomplished by waves travelling along the basilar membrane in the inner ear (cochlea). Higher frequencies resonate near the input while lower frequencies resonate progressively nearer the apex of the cochlea. The natural frequency scale of the ear, which corresponds to the total length (~35mm) of the basilar membrane, has a resolution of about 1.5mm, giving rise to about 24 "critical bands" which span the audible range of approximately 20Hz~20000Hz.

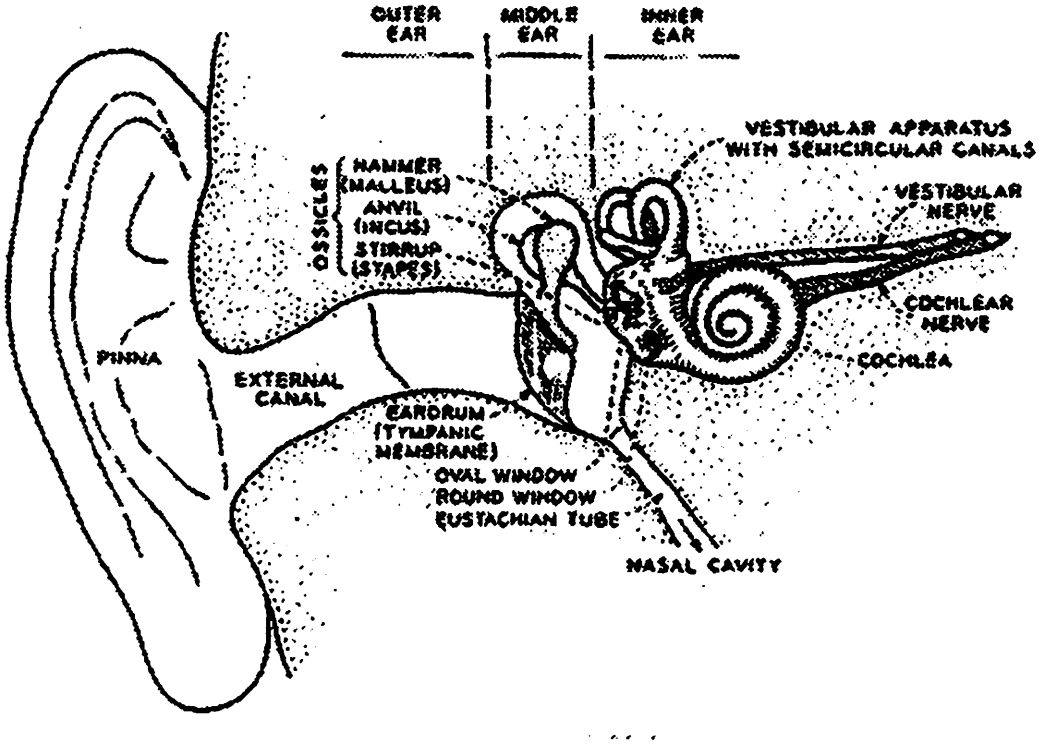


FIGURE 1.2 Physiological model of the human ear

The PLP model suggests that the human perception of frequencies can be mimicked by passing the sampled speech through a bank of Q bandpass filters, giving the signals:

$$s_i(n) = s(n)*h_i(n) \qquad \text{for } 1 \le i \le Q$$
$$= \sum_{m=0}^{M-1} h_i(m)s(n-m)$$

(EQ 1.6)

However, instead of using EQ1.6, the critical band filtering is done in the frequency domain. The same effect could be achieved by weighting the FFT coefficients according to the magnitude frequency response of a filter bank. The center frequencies of the filters are spaced equally on a Bark scale from 0Hz through 8Hz. Each filter spans a common bandwidth on a Bark

scale. The Bark scale is a non-linear transformation of the usual frequency scale, given below in EQ 1.7. From Figure 1.3, it is clear that for the same bandwidth in Bark scale, band A, which is centered at a higher frequency, spans a wider range of actual frequencies then band B. This mimics the progressively coarser resolution of human perception at increasing frequencies.

Frequency: ω rad s^{-1}
Bark Frequency: Ω

$$\Omega(\omega) = 6\ln\left\{\frac{\omega}{1200\pi} + \left[\left(\frac{\omega}{1200\pi}\right)^2 + 1\right]^{0.5}\right\}$$

(EQ 1.7)

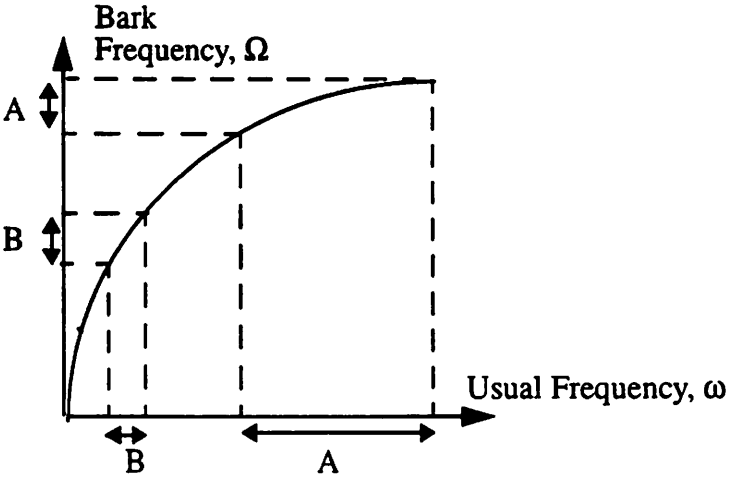


FIGURE 1.3 Non-Linear Transformation to Bark frequency.

1.1.3 Relative Spectral, Perceptual Linear Predictive model (RASTA-PLP)

The PLP model, however still suffers from linear distortions in the communications channel. A simple muffling of speech could change the LPC coefficients sufficiently to fail a recognizer trained on clean data. Hermansky et al. [6] proposed a refinement to LPC to make it more tolerant to channel modulation effects (convolutional noise). The following were added in their technique called RASTA-PLP (Relative Spectral, Perceptual Linear Predictive):

- Bandpass filtering estimates the temporal derivative of the Bark-scale-transformed spectrum by drawing a regression line through spectral values obtained in the last 5 consecutive windows.

• Add the equal loudness curve and raise the coefficients by power of 0.33 to simulate the power law of hearing. The raise power function is simplified by using a natural log, followed by a multiplication of factor 0.33, and an inverse logarithm (exponential function). i.e. $c^{0.33} = \exp(0.33 * \log(c))$.

The complete block diagram is shown in Figure 1.4. The non shaded blocks make up the most basic LPC analysis model. The critical band filtering block (medium shading) implements the PLP improvement, while the most heavily shaded blocks complete the RASTA-PLP algorithm.

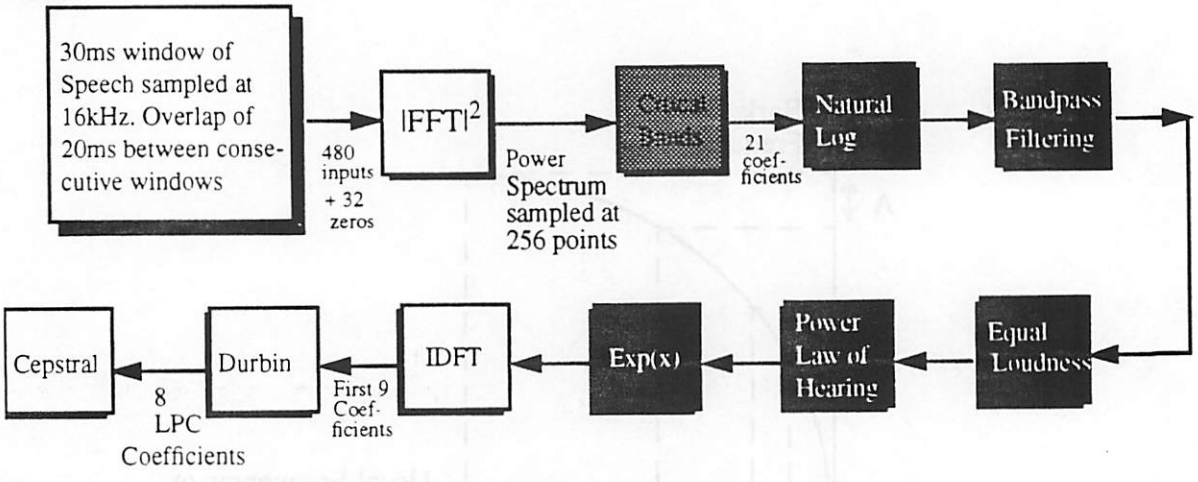


FIGURE 1.4 Flowgraph of RASTA-PLP

1.2 The Hyper synthesis Environment

The Hyper system is an integrated synthesis environment for real time applications. It uses the Silage[7] language for behavioral description input of the application. Silage is a signal-flow language developed especially for DSP specifications. It is an applicative language whose fundamental operation is function application. The term Static Single Assignment (SSA) is used to describe the computation method of Silage. SSA avoids any false dependencies between intermediate data, thus helping the exploitation of parallelism. Other advantages include built-in stream and temporal operators for handling of continuous streams of discrete-time inputs, and built-in data types that allow user specification of data bit widths.

Hyper parses the Silage description and compiles a Control/Data flow graph (CDFG). The CDFG represents the algorithm as a flow graph consisting of nodes, data edges, and control edges.The nodes represent primitive data operators such as adders, comparators and memory read/writes. The data edges represent data precedences between these nodes. Control edges are introduced to enforce extra precedence rules between nodes (e.g. the execution time of operation X has to trail the execution of operation Y by at least N clock cycles). Aside from the standard arithmetic operations, the CDFG allows a number of macro control flow operations such as loops and if-the-else blocks. The introduction of those control statements results in a hierarchical graph. The body of a loop or a conditional is represented by a sub-graph, which is contracted into a single node at the next higher hierarchy level. The hierarchical representation has the advantages of compactness and descriptiveness. It also preserves any structural hints from the designers.

The CDFG serves as a central database to which all other Hyper tools (Figure 1.5), such as min-bound estimations, flow graph transformations, and assignment/scheduling, refer when executed. The results of these operations are back annotated onto the database. The following is a brief description of each class of operations

- .**Memory Management:** Performs memory moduleselection, memory merge and address generation. Memory module selection assigns an appropriate memory module from the hardware library for each array in the application. Memory merge allows 2 or more different arrays to be merged onto a single memory module. Address generation reads the memory address annotated on the array read/ write nodes in the CDFG and creates nodes that will be executed as any other data operation to generate the address.

- **Module Selection:** Selects an appropriate hardware library module for each flow graph operation.

- **Estimation:** Derives the minimum and maximum bounds on the required hardware resources. This information will serve as an initial solution and will help select the next synthesis operation to perform.

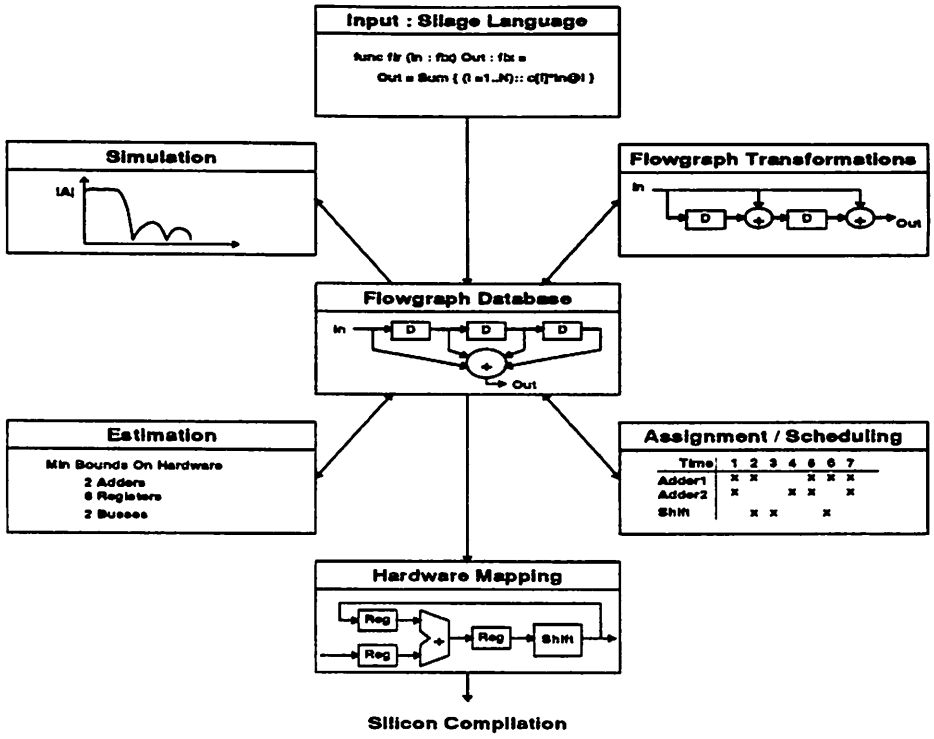


FIGURE 1.5 The Hyper synthesis environment

- **Transformation:** Manipulates the signal flow graph of the algorithm to improve the final implementation, without changing the input-output relation. Typical transformations are retiming, loop unrolling and common-subexpression elimination and pipelining.

- **Allocation, Assignment and Scheduling:** Selects the amount of hardware resources (execution units, registers and interconnect), needed for the execution of the algorithm. Bind each flow graph operation to a particular hardware unit and time slot.

- **Hardware Mapping:** Maps the allocated, assigned and scheduled flow graph (called the decorated flow graph) onto the available hardware blocks. The result of this process is a structural description of the processor architecture in the SDL[8] language, which serves as the input to the LagerIV silicon assembly tool suite.

Furthermore, the Hyper simulator can be invoked after any of the synthesis steps (though Hardware Mapping does not alter the CDFG) to verify the functionality of algorithm and the correctness of the transformations performed. Also, when run in the Bit-True mode, which simulates with a fixed point representation of the data with bit-widths as specified by the user, the simulator can optimize and check the value of a number of performance parameters, such as the signal to noise ratio, or the effects of truncation on the transfer function.

Hyper allows the designer to play the area-time-power trade-off game. The effect of the various built-in transformations are easily reflected by the Estimator results. Also, Hyper provides comparisons between different algorithms pertaining to the same behavior.

1.3 SPA - Stochastic Power Analysis

SPA is a tool that allows a designer to obtain predictions of area and power consumption at the architectural level. The input to SPA is a register-transfer level description of the architecture to be analyzed and a set of input vectors. Both textual and graphical results are available, with area and power estimates broken down by hardware class and type of component.

The primary RTL input describes the architecture under consideration. Conceptually, this consists of the information displayed in Figure 1.6. In particular, it contains a (possibly hierarchical) description of datapath blocks used in the chip, as well as the interconnect network joining these blocks. SPA uses a textual architectural description language (ADL) for this purpose. The information about the interaction between the blocks is embodied in the control path of the chip. Using a control description language (CDL), the control flow of the design can be described as a set of state tables. These tables specify how the next state and outputs of the control module relate to the present state and inputs. In order to maintain a relatively high level of abstraction, CDL allows the user to specify control signals and states as enumerated (symbolic) types rather than bit vectors. For example, the instructions to a memory can take the form of "READ" or "WRITE" rather then obscure binary codes.

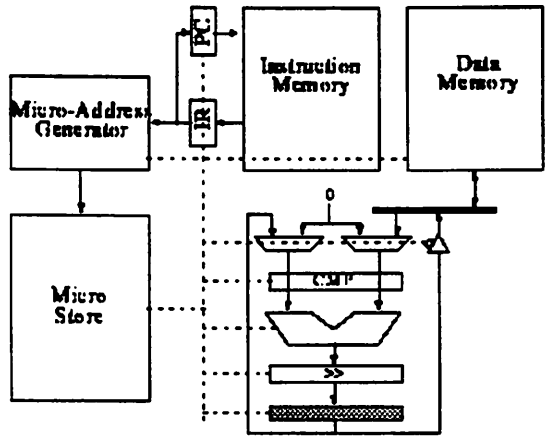


FIGURE 1.6 Architecture of micro coded instruction set processor

SPA makes use of power and area models that are stored in a hardware library. Unlike the high-level power estimator used in Hyper which uses a white noise input to the design, SPA accounts for the effect of hardware activity on power consumption. It collects activity statistics for each bus and module through a complete functional simulation of the architecture.

SPA performs three types of analysis, interconnect analysis, area analysis, and power analysis. Interconnect analysis estimates the physical capacitance associated with wires in the design. The result of this is essential for analysis of both area and power. Power estimation is performed using the Dual Bit Type (DBT) [13] model for datapath and memory elements and the Activity-Based Control (ABC) [13] model for control path elements.

1.4 LagerIV

LagerIV is a CAD tool suite to enable rapid prototyping of Integrated Circuits. Designs can be described in high-level languages / descriptions and converted to layout for fabrication. The conversion is enabled by the use of structure processors and layout generators.A structure processor performs a description/netlist transformation on an Oct Structural Master View (SMV), and generates an Oct Structural Instance View (SIV). Structural processors used include **dpp** for datapath modules and **bdsyn** for control cells. A layout generator performs the physical place-

ment and signal routing given an SIV netlist description and generates an Oct physical view and **Magic** output. Layout generators used include **Flint** and **Stdcell.**

2.0 High Level Analysis

The design path begins with a description of the algorithm for the various blocks in Silage. For each of the major blocks, several algorithms performing the same functionality are investigated. The Hyper simulator provides a functionality verification while the Hyper estimator gives an initial profile of each algorithm by estimating power consumption, memory accesses, area, etc.

Initial estimates of all the blocks (Figure 2.1) in the RASTA-PLP algorithm revealed the power consumption and critical path of the FFT block to be well above any other blocks. From a power prospective, that is where most of the design effort will be directed towards. In addition, other then the FFT and the Critical Band Filtering blocks, the remaining processes, such as the Exponential, Equal Loudness and Power Law blocks, have fairly straightforward and short computations, leaving less room for design space exploration. The main effort of the high level analysis is therefore pointed at the FFT and Critical Band Filtering blocks, while similar work on the remaining blocks will see less detailed. The emphasis on power continues down to the architectural design level. The selected algorithms for the FFT and the Critical Band Filtering blocks will be mapped onto a Register Transfer Level (RTL), and have their layouts generated eventually. Further architectural analysis of the FFT block will be provided by a power and area breakdown using SPA.

The use of Silage description has inherent advantages compared to more commonly known procedural languages such as C. However Silage requires the use of manifest loops and manifest array indices which, as will be seen, is an obstacle to the description of less structured, and less regular algorithms. In such cases, the appropriateness of a C++ input description approach will be explored.

The sampling rate of the speech input was chosen to be 16kHz. A 30ms window is applied on the sampled speech to obtain 480 data points. This is zero-filled to make up the 512 data points needed for a 512-point FFT. Consecutive 30ms-windows have an overlap of 20ms. The throughput of the RASTA front end must therefore be less than 10ms. In order to meet the time criteria, a

2-staged pipe-lining is used. The 512-point FFT takes up most of the first stage while the rest of the front-end is computed in the second stage.

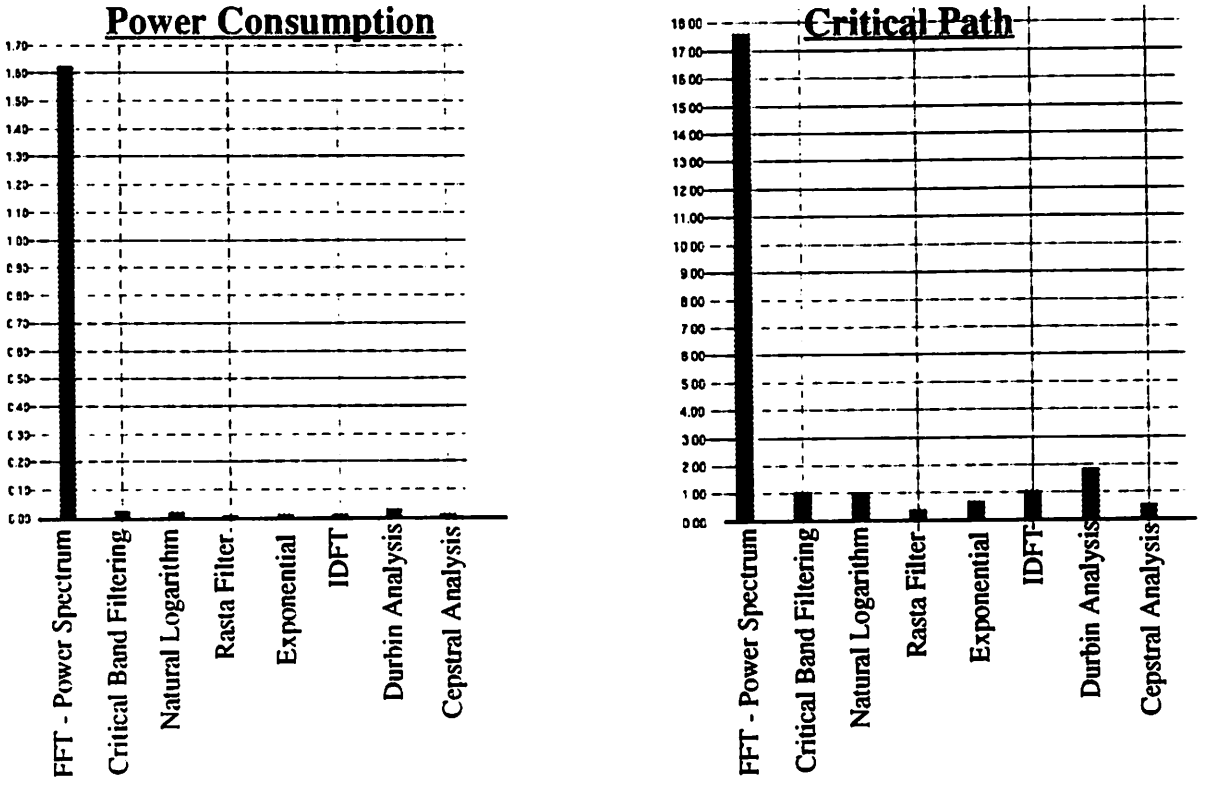


FIGURE 2.1 Initial Estimates of Blocks in RASTA-PLP block showing a disproportionately large amount of power consumed in the FFT block.

2.1 $|FFT|^2$, Power Spectrum

The periodogram estimate of the power spectrum of a 30ms segment of speech consisting of M samples is defined as

$$R_p(e^{jw}) = \frac{1}{M}\left|X(e^{j\omega n})\right|^2$$

where

(EQ 2.1)

$$X(e^{j\omega}) = \sum_{n=0}^{M-1} x(n)e^{-j\omega n}$$

(EQ 2.2)

The power spectrum can therefore be estimated by the magnitude square of the FFT of the input samples. The constant factor $^1/_M$ is ignored in the computations as doing so will simply multiply every autocorrelation coefficient by a factor of M. Equation 1.3 remains unchanged, and therefore, does not affect the final cepstral values.

The 480 speech samples contained in the 30ms window are added with 32 zeros in order to make a Radix-2 FFT (512-point FFT) feasible. Since the inputs are all real, the power spectrum is symmetrical about w=π, or about the 256th point of the FFT. Thus only the first 256 output-coefficients are required, and they represent the power spectrum from 0 through 8kHz.

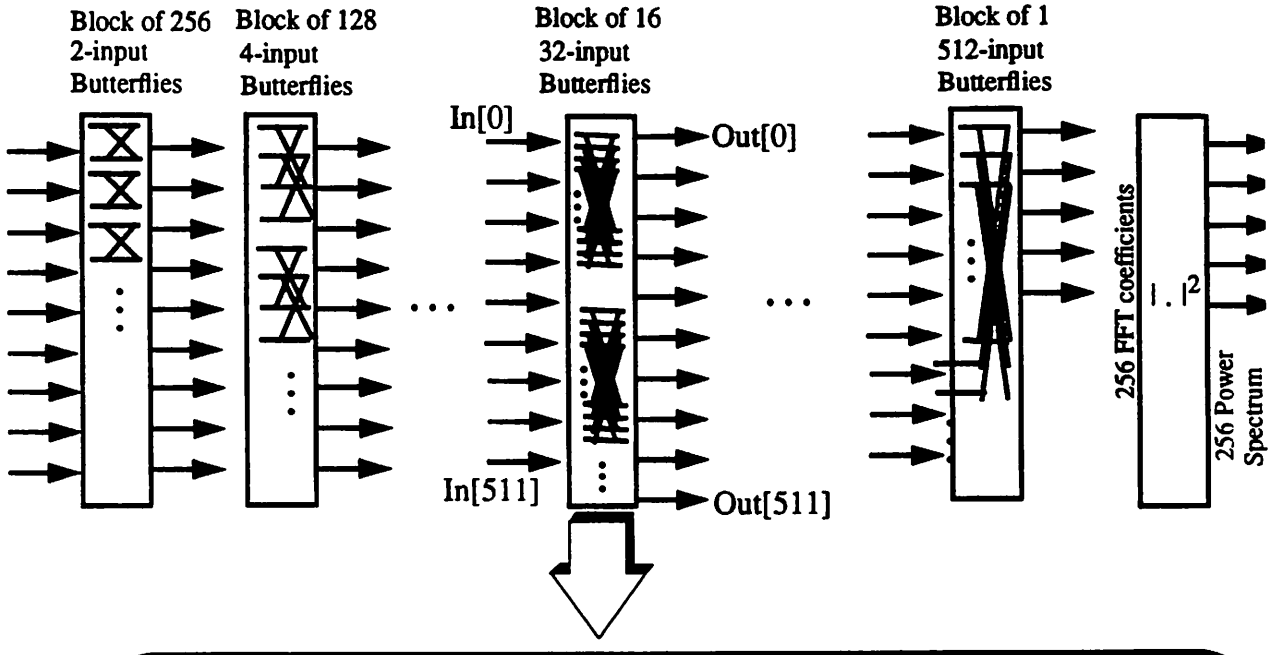
A number of FFT algorithms exists. It is found that algorithms that reduce the computation complexity in terms of number of additions and multiplications, will on the other hand, have less regularity in their structures. Memory address generation and number of memory accesses also proves to be a big issue since the memory size is massive (512 for each array). In particular, the following choices were examined:

- Radix 2 algorithm
- Radix2-Winograd Algorithm
- Split Radix (2,4) Algorithm

2.1.1 Radix 2 algorithm

The 2^9=512 point radix-2 FFT can be realized as a cascade of 9 blocks. Each block consists of repetitions of either a 2-input, 4-input,..., 256-input or 512-input butterfly respectively. There are 2 x 256 complex twiddle factors accounting for distinct values of $\cos(^{k\pi}/_{256})$ and $\sin(^{k\pi}/_{256})$, for 0<=k<256. These are stored in two 256-word ROMs.

In Silage, each block is realized as a nested loop of depth 2. The code within Figure 2.2 shows that of a typical block. The last four statements in the deepest nest implements the cross additions (complex additions) in a butterfly. Later blocks have fewer butterflies with more inputs, so the *j* variable iterates over fewer values, while the k variable iterates over more values.




```
(j : 0 .. 15)::
  begin
   (k : 0 .. 15)::
    begin
     real_low = In_real[32*j + k];
     imag_low = In_imag[32*j + k];
     cosw = Wr[16*k];
     sinw = Wi[16*k];
     temp5r = word(cosw * real_low) - word(sinw * imag_low);
     temp5i = word(sinw * real_low) + word(cosw * imag_low);
     Out_real[32 * j + k] =  In_real[32*j + k + 16] + temp5r;
     Out_imag[32 * j + k] = In_imag[32*j + k + 16] + temp5i;
     Out_real[32 * j + k + 16] = In_real[32*j + k + 16] - temp5r;
     Out_imag[32 * j + k + 16] = In_imag[32*j + k + 16] - temp5i;

    end;
  end;
```

Cross Additions of Butterfly

FIGURE 2.2 Radix-2 FFT. 9 stages of butterflies and a $|.|^2$ operation.

Note that the array address indices of inputs and outputs in the Silage statements differ only by either a constant addition or a constant multiple. Common subexpression elimination for the address generation of these array indices was therefore feasible. Instead of computing the value of, say *(32 *j + k + 16)* and *(32*j + k)* every time those address are needed, *(32*j + k)* is computed only once during each iteration of the innermost loop, and *16* is added to this value also only once.

TABLE 1. Comparison of counts before and after Common Subexpression Elimination

Module	Count Before CSE	Count After CSE
Add	38142	17150
Multiply	26624	12032
Twiddle Factor Memories	4096	4096
Estimated Energy Consumption	12.7μJ	8.3μJ

It is observed that each butterfly produces outputs which occupy exactly the same addresses as the inputs. The above example, for instance, obtains inputs from In[32*j + k] and In[32*j + k + 16], and outputs to Out[32*j + k] and Out[32*j + k + 16]. Thus, using only one pair of input memory hardware for the real and imaginary parts respectively, each iteration of the butterfly obtains a number of inputs from the memories and overwrites the inputs with its outputs. However, the description of such overwriting of array values will violate the static single assignment (SSA) checks in Silage. This problem is solved by using different array names in the Silage description for the outputs of each block, and then implementing in-place storage of these different arrays by manually altering the intermediate CDFG.

The simplicity of the radix-2 algorithm is attractive. However, at a clock period of 550 ns (Section 2.6 describes the choice of clock frequency), the critical path was 23807 cycles. This works out to ~10.3 ms which exceeds the required throughput of the RASTA front end. Although more pipe-lining might solve the throughput constraint, this will invariably increase the power consumption and area as additional hardware is required to handle the parallel computations of a pipelined algorithm.

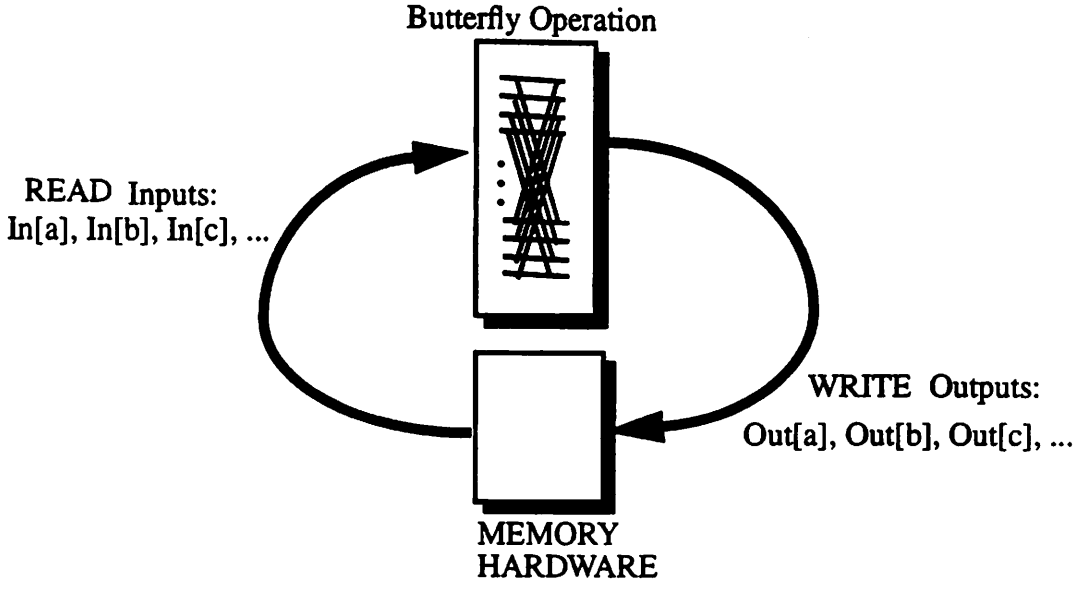


FIGURE 2.3 In-place storage of Intermediate results

2.1.2 Radix-2-Winograd Algorithm

In the Radix-2-Winograd Algorithm, the first 3 blocks (2-input, 4-input, and 8-input) of the Radix-2 algorithm are replaced with a single block consisting of 32 Winograd 8-point FFTs [11]. The approach takes advantage of the fact that for the first 3 blocks of the Radix-2 algorithm, more then half the number of complex twiddle factors are either ± 1 or $\pm j$, while the rest of the them correspond to $\exp(^{j\pi}/_{4})$ and $\exp(^{-j\pi}/_{4})$. These last 2 terms, which have equal real and imaginary parts, constitute to the only non-trivial multiplication, a factor of 0.707106 ($^{1}/_{Sqrt(2)}$). Each 8-point Winograd FFT, uses only 2 multiplications, compared to an 8-point Radix-2 algorithm which requires 8 multiplications. Also, instead of reading the twiddle-factors from memory, the only non-trivial twiddle-factor, $^{1}/_{Sqrt(2)}$ is stored in a constant register. This eliminates the need for address generation of the twiddle factors as well as expensive memory accesses. Appendix B shows the Silage description of the Radix2-Winograd algorithm. The Winograd FFT, is described in Silage as the function *fft8*. It is clean, has no iteration loops, and consists primarily of cheap additions and subtractions.

As a result, the critical path is reduced to 17607, a reduction of 26%. This proves to be critical, as it is possible to achieve a throughput below the 10ms criteria, without any further pipe-

lining. At the same time, the reduction in computational complexity also leads to a 20% reduction in power consumption.

TABLE 2. Comparison of EXU counts for a 512-point FFT using Radix-2 and Radix2-Winograd

Module	Radix-2	Radix2-Winograd
Add	14590	12230
Subtract	5632	4928
Multiplie	12032	9472
Log Comparator	2558	1670
Transfer	9990	7756
Register Accesses	109064	87233
Memory Accesses	20224	16128
TOTAL EXU COUNT	174090	139417
CRITICAL PATH	23807	17607
ESTIMATED ENERGY CONSUMPTION	8.3µJ	6.6µJ

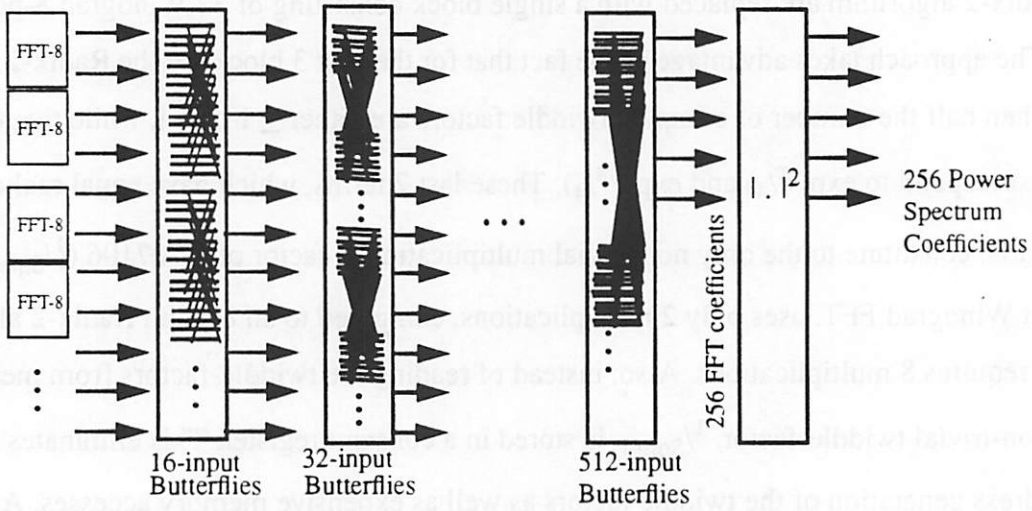


FIGURE 2.4 Radix-2-Winograd Algorithm. The FFT-8 is handled by a Winograd small-N FFT.

2.1.3 Split-Radix Algorithm

The split-radix (radix-2 and radix-4) algorithm combines the relatively small number of addition terms in a radix-2 algorithm, and the smaller number of multiplications in a radix-4 algo-

rithm. At any stage, a N-DFT is decomposed into a $^{N}/_{2}$-DFT using radix-2, and two $^{N}/_{4}$ DFT using radix-4.

$$X_{2k} = \sum_{n=0}^{\frac{N}{2}-1} W_{\frac{N}{2}}^{nk}\left(x_n + x_{\frac{N}{2}+n}\right)$$

(EQ 2.3)

$$X_{4k+1} = \sum_{n=0}^{\frac{N}{4}-1} W_{\frac{N}{4}}^{nk} W_N^{n}\left[\left(x_n - x_{\frac{N}{2}+n}\right) + j\left(x_{n+\frac{N}{4}} - x_{n+\frac{3}{4}N}\right)\right]$$

(EQ 2.4)

$$X_{4k+3} = \sum_{n=0}^{\frac{N}{4}-1} W_{\frac{N}{4}}^{nk} W_N^{3n}\left[\left(x_n + x_{\frac{N}{2}+n}\right) - j\left(x_{n+\frac{N}{4}} - x_{n+\frac{3}{4}N}\right)\right]$$

(EQ 2.5)

Due to asymmetry in the decomposition, the structure of the algorithm is more involved than either of the previous two algorithms. Appendix C shows the Silage description of the Split-Radix Algorithm. It is highly unstructured, and difficult to understand. Unlike the previous methods where each stage can be described as a recursive decomposition and therefore, as manifest loops, the split-radix algorithm distinguishes between the decomposition of X_{2k} and X_{4k+1}, X_{4k+3}. A Fortran description written by Sorenson[12] implements the split radix in loops that have non-manifest, loop counters that have variable lower and upper bounds. A similar realization in Hyper is difficult with Silage as the input description language because the current Silage parser accepts only manifest loop counters. Therefore, a complete loop unroll of the Fortran code is necessary, resulting in a long and irregular series of loops.

On the other hand, the problems associated with the Split Radix algorithm may be solved if a C++ input description language to Hyper were available. Typical C++ syntax would allow both non-manifest loops and multiple assignments of variables. This would also solve the problem of having excessive arrays, which was the side-effect of the loop unrolling, which in turn, was needed to satisfy the manifest loops requirement in Silage.

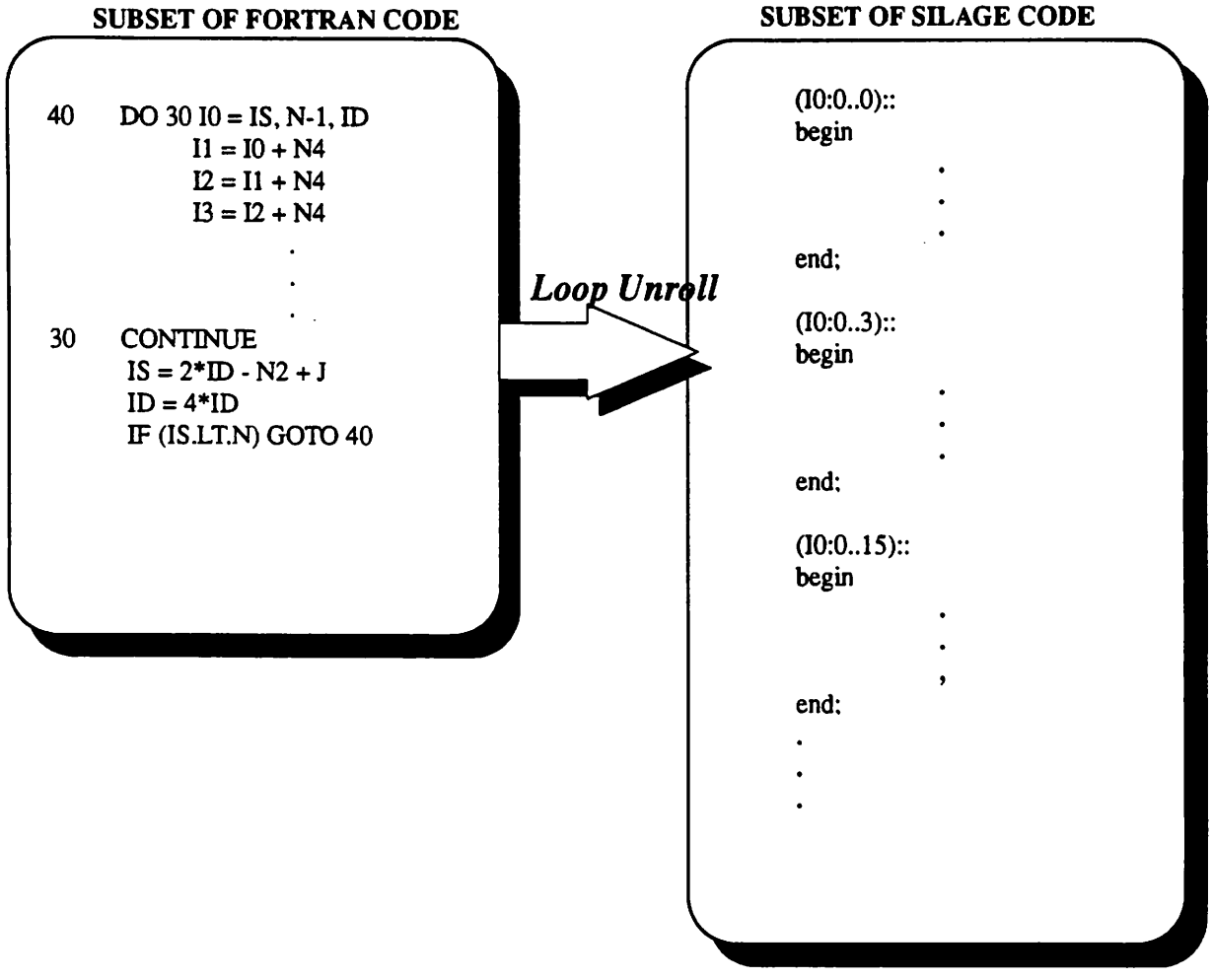


FIGURE 2.5 Loop Unrolling during translation from Fortran code to Silage code.

It was discovered that the split-radix algorithm enjoys more then a twofold reduction in critical path to 8307. The power consumption saw an impressive improvement of about 30%. Table 3 gives a comparison between Radix2-Winograd algorithm and the Split-Radix algorithm. It shows significant improvement in terms of number of additions and multiplications.Similar

Common Subexpression Elimination transformations as that described earlier in Section 2.1.1 were also employed to achieve the following figures.

TABLE 3. Comparison of EXU counts for a 512-point FFT using Radix2-Winograd and Split-Radix Algorithms

Module	Radix2-Winograd	Split-Radix
Add	12230	8818
Subtract	4928	4224
Multiplies	9472	6016
Log Comparator	1670	946
Transfer	7756	5277
Register Accesses	87233	62570
Memory Accesses	16128	12800
TOTAL EXU COUNT	139417	100651
CRITICAL PATH	17607	8307
ESTIMATED ENERGY CONSUMPTION	6.6μJ	4.6μJ

2.1.4 Choice of Algorithm for FFT

The Radix2-Winograd algorithm is chosen as the algorithm to implement the FFT due to its ability to meet the target throughput, as well as it overall regularity which simplifies the implementation.and lowers amount of overhead.

The Radix2-Winograd algorithm is a swift and obvious improvement over the Radix-2 algorithm. With a little modification of the Silage description, a respectable amount of power savings is achieved. The extent of irregularity introduced by the Winograd-8-point FFT is also minimal. Recall that the 8-point Winograd FFT is implemented as a efficient code with mostly additions and subtractions which have a low cost of power consumption.

It is worth noting, however, that the the initial Radix-2 algorithm is extremely regular. The Silage code basically implements each of the 9 blocks in the same way, differing only in the iteration bounds and constants in the array indices. In terms of architecture, the Radix-2 can be implemented easily with only one structural block that is repeated 9 times with the appropriate modification of the iteration bounds and other constant values.

The comparison between the Radix2-Winograd and the Split Radix algorithm, however, is less clear cut. While the Split Radix algorithm described in the last section has obvious gains in terms of both execution counts as well as power estimates, the description in Silage has been difficult, and even looks awkward when it finally works. Moreover, the less regular structure also makes it less desirable then the radix-2-Winograd algorithm. Section 3.3, will describe how regularity in the algorithm can translate to a more compact architecture with less interconnect overhead, leading consequently to reduced power consumption. Regular algorithms also typically require less control. Though there has been some effort to detect and quantify such useful properties of algorithms [15] and [16], transformations for enhancing these properties are largely unexplored. There is therefore reason to believe that the difference of power consumption between the Radix2-Winograd algorithm and the Split-Radix algorithm may not be as pronounced as the numbers suggested in Table 3.

Currently, work is being done to allow C++ as the input behavioral language for Hyper. This will allow a more concise description of the split radix algorithm, including provision for non-manifest loop counters that will make the difference in both description and implementation.

Table 4: Summary of algorithms used for $|FFT|^2$ block

Algorithm	Power Reduction Technique	Incremental Power Improvement	Benefits/Disadvantages
Radix -2 (Appendix A)	Common Subexpression Elimination	36% Power Reduction	Regularity of Structure
Radix2-Winograd (Appendix B)	Partly Different Algorithm Twiddle factors no longer need to be stored in arrays ==> Reduced Memory Power	20% Power Reduction	Fewer Twiddle Factors Less Regularity

Table 4: Summary of algorithms used for $|FFT|^2$ block

Algorithm	Power Reduction Technique	Incremental Power Improvement	Benefits/Disadvantages
Split Radix (Appendix C)	Wholly Different Algorithm Sizable reduction in additions and multiplications. Also reduced number of other EXU counts, e.g. memory accesses, comparators, etc.	30% Power Reduction over Radix2-Winograd	Asymmetry of decomposition leads to difficult behavioral description in Silage. Much less regularity

2.2 Critical Band Filters

Zwicker [10] proposed that the bandwidths of the critical bands are approximately constant on the Bark scale, Ω, whose relation with the usual frequency scale, $\omega = 2\pi f$, was given in EQ 1.7. Schroeder's [9] proposal of the asymmetric critical band filtering function $F(\Omega)$ (frequency response) in EQ 2.3 for a filter centered at Ω_0, is approximated by piecewise linear function given in EQ2.4.

$$10\log F(\Omega - \Omega_0) = 15.8 + 7.5(\Omega - \Omega_0 + 0.5) - 17.5\sqrt{1 + (\Omega - \Omega_0 + 0.5)^2}$$

(EQ 2.6)

Critical band filtering is done in the frequency domain since both the frequency response of the critical band filters as well as the FFT of the signal are known. The power spectrum consists of 256 distinct samples that span 0 through 8kHz. The frequency response of the bank of filters is sampled at a resolution of $^{8kHz}/_{256} = 31.25Hz$. The power spectrum coefficients are then

weighted correspondingly by the magnitude of these samples. The 19 bandpass filters, are centered at equal Bark intervals, **B**, and span the frequency interval 0 through 8kHz.

$$\Psi_i(\Omega) = \begin{cases} 0 & \text{for } \Omega-\Omega_i < -1.3 \\ 10^{2.5(\Omega-\Omega_i+0.5)} & \text{for } -1.3 < \Omega-\Omega_i <= -0.5 \\ 1 & \text{for } -0.5 < \Omega-\Omega_i < 0.5 \\ 10^{-(\Omega-\Omega_i-0.5)} & \text{for } 0.5 <= \Omega-\Omega_i < 2.5 \\ 0 & \text{for } \Omega-\Omega_i > 2.5 \end{cases} \qquad \Omega_i = i \times B, \quad i \in [1, 19]$$

(EQ 2.7)

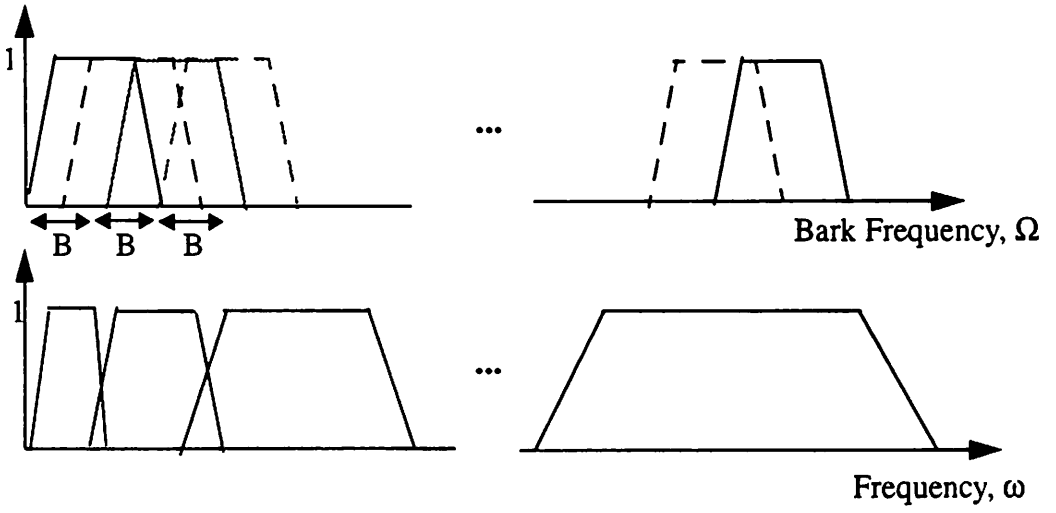


FIGURE 2.6 Critical Band Filters. Filters are centered at equal Bark intervals, **B**.

In accord with the RASTA-PLP, the first sample (d.c.) is copied from sample at Ω=B, and last samples (centered at 8kHz) is copied from sample at Ω=19B to make a total of 21 samples spanning 0 to 8kHz.

2.2.1 Direct Computation

The initial algorithm for computing the critical-band outputs consists of 19 loops. Each loop essentially performs the critical-band masking. It uses an iterative multiply-add to sum the multiplication of the power spectrum coefficients with the height of the masking curve. A typical loop is shown in the inset of Figure 2.7.

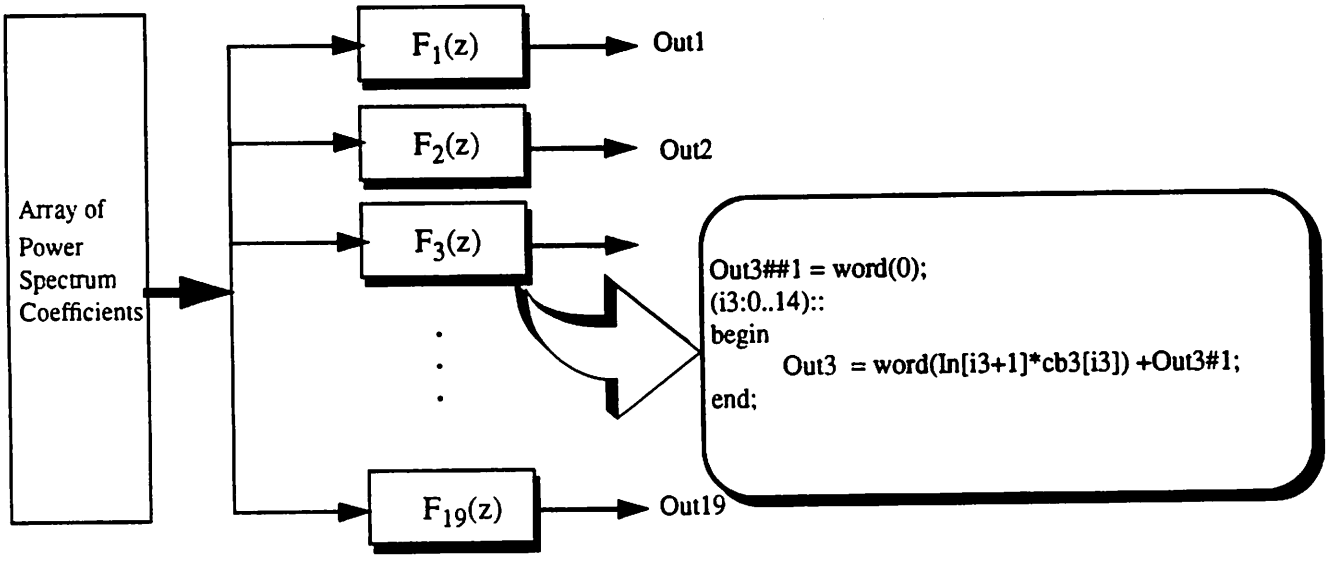


FIGURE 2.7 Implementation of the Critical Band Filtering in Frequency Domain. Inset shows description of a typical filter. *cb3[i]* is the height of the i^{th} sample of the window centered at Ω=3B (Bark frequency scale). The corresponding power spectrum coefficient, *In[i]*, is weighted by the constant stored in *cb3[i]*, and the product is updated to the iterating sum *Out3*.

In this algorithm, each filter in the bank of filters works independently of one another, and therefore appears to have a high degree of parallelism. In order to exploit any form of this concurrency property, there must be a way to simultaneously read more then one filter input, from the array of $|FFT|^2$ coefficients. The hardware used in Hyper does not allow multiple simultaneous read/writes from the same memory. Hyper does however permit partitioning of an array such that different ranges of elements of the array may reside in different memory hardware. Multiple simultaneous read/writes from the same array is thus possible as the data access will be made from different memories. Nevertheless, due to the overlap of frequencies between consecutive filter bands, some of the array elements of the power spectrum coefficients will have to be repeated in more then one partitioned memory. The total size of the partitioned memories will be necessar-

ily greater then 256, which is the number of distinct values in the array. In this case, increased concurrency is traded off for additional memory size, which can lead to higher power cost of memory accesses. The high level of concurrency is usually an important property, but here, where the 10ms limit is ample even when the filter outputs are computed serially (not concurrently), it would be a waste of power to operate the filter functions in parallel.

Besides requiring additional hardware, running the filters in parallel also required an excessive 832 reads (per set of sample inputs) from the 256 power spectrum coefficients. This is the result of the filters operating independently, and the extensive overlap between consecutive bands. Each power spectrum coefficient usually falls within the bandwidth of 3 or 4 neighboring filters. This undue number of reads is not power efficient as energy is required for additional address generation and memory accesses.

2.2.2 Improved Algorithm

To overcome the problem with excessive read operations, the 19 loops are taken apart, and replaced with 30 loops which represents the run-length of the overlapped power spectrum coefficients. For instance, if In[62] through In[72] fall within the windows centered at Ω=11B, 12B, 13B, and 14B, the following loop covers part of the 11^{th}, 12^{th}, 13^{th} and 14^{th} critical bands:

```
/*
* In: 62..72::
*/
Filt11_6##1 = Filt11_4;
Filt12_5##1 = Filt12_3;
Filt13_3##1 = Filt13_1;
Filt14_1##1 = word(0);
(i17:62..72)::
begin
   a17 = In[i17];
   Filt116 = word(a17 * cb11[i17-36]) + Filt11_6#1;(1)
   Filt125 = word(a17 * cb12[i17-43]) + Filt12_5#1;(2)
   Filt133 = word(a17 * cb13[i17-52]) + Filt13_3#1;(3)
   Filt141 = word(a17 * cb14[i17-62]) + Filt14_1#1;(4)
end;
```

where Filt11_X, Filt12_X, Filt13_X, and Filt14_X are the iterating sums for the 11^{th}, 12^{th}, 13^{th}, and 14^{th} critical bands respectively. This maneuver resulted in exactly 256 reads for the 256 power spectrum coefficients. Despite the increase from 19 loops to 30 loops, which would also require more control power due to increased control activity, the reduced EXU counts, primarily

due to reduced memory accesses, as well as address generation computations, overcomes this negative effect.

TABLE 5. Comparisons of EXU counts between algorithm for Critical Band Filtering

Module	Original Direct Computation	Improved Algorithm
Add	2461	1903
Multiply	832	837
Log Comparator	832	255
Transfer	5	22
Register Accesses	10011	7584
Memory Accesses	1683	856
TOTAL EXU COUNT	15824	11715
CRITICAL PATH	3328 (Filter outputs are computed serially, not concurrently)	1042
ESTIMATED ENERGY CONSUMPTION	327nJ	238nJ

The improved algorithm reads a single input, and computes the partial sum of several band coefficients. In the example above, each input read into *a17* is used to compute the four instructions (1), (2), (3), and (4). The partial sums can be done simultaneously. The assignments have no data-dependencies, lending to a certain amount of parallelism in the algorithm. In order to exploit this advantage, the number of EXU's will also need to be increased. This increases the switching capacitance of global signals such as CLK, and control signals, which are also routed to other parts of the circuit such as the global controller, and the power spectrum module (FFT block). Nevertheless, if the global signals are gated and turned off between the immediate completion of the critical-band filtering, and the next sample period (when the critical band would be in waiting mode), the reduced critical path would require fewer clock switches, and therefore a net power consumption of 27%!

As mentioned in Section 2.0, the remaining blocks will be analysed in a less detailed fashion since they are computationally far less complexed then the first 2 blocks that have been analyzed thus far. The remaining sections of the Chapter describes the Silage input for each of the blocks, concluding by a short discussion on selection of clock period and a table of estimates of important parameters such as critical path and power numbers, obtained at the chosen clock frequency.

2.3 Modeling of Critical Band Filter outputs

In order to improve the recognizer's tolerance to a changing communication channel, RASTA-PLP estimates the temporal derivative of the Bark-scale-transformed spectrum by drawing a regression line through the spectral values obtained in the last 5 consecutive input sets. Equal Loudness Preemphasis [19] and Intensity-loudness Power Law [21] both further simulate human hearing conditions.

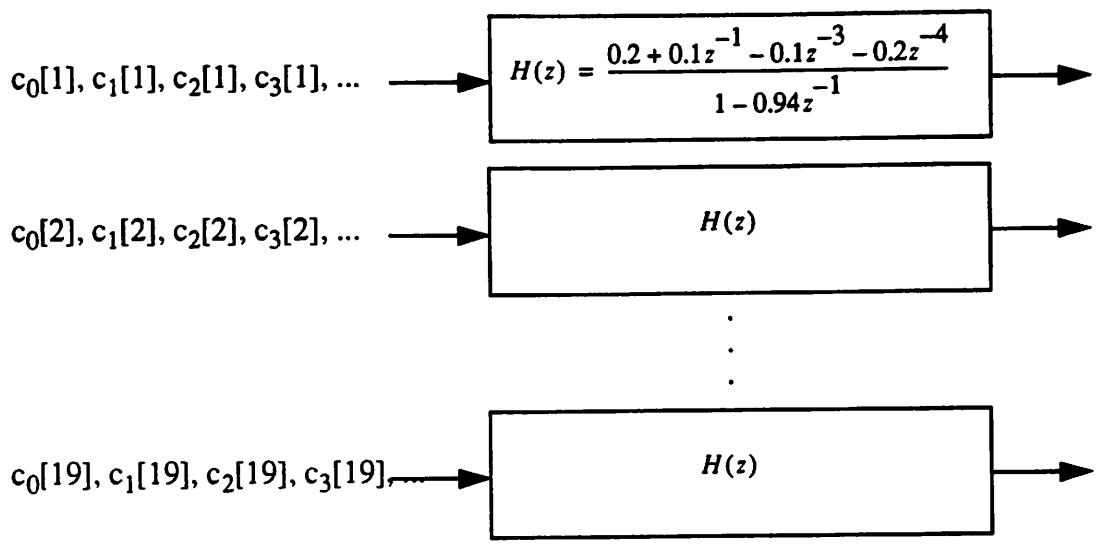
Due to the simplicity of computations involved during Estimation of Temporal Derivative, Equal Loudness Preemphasis and Intensity-loudness Power Law, the Silage code for these 3 blocks are combined into one description given in Appendix F. The Natural Logarithm and Exponential algorithms assists in the implementation of the Power Law block, and allow the Equal Loudness Preemphasis filtering to be done simultaneously. Sections 2.3.2 and 2.3.3 show why. Appendix G and H list the Silage description for the Natural Logarithm and Exponential blocks respectively.

2.3.1 Estimation of temporal derivative

Each of the 19 critical band coefficients is streamed into an averaging filter as shown in Figure 2.8. Since consecutive speech windows have a $^2/_3$ overlap in time (20ms overlap of the 30ms window), there is a high correlation between the current and previous critical band outputs. The averaging removes some error due to possible changes in the channel. Silage description of the filter is achieved by one simple sentence for each filter:

```
Out = ((Out@1 * 0.94) + (0.2 * In) + (0.1 * In@1) - (0.1 * In@3)  - (0.2) * In@4));
```

(EQ 2.8)



$c_i[n]$ is the n^{th} critical band output at time i.

FIGURE 2.8 Filtering that implements Estimation of the temporal derivative.

2.3.2 Equal Loudness preemphasis

The RASTA-PLP algorithm preempahsizes the sampled power spectrum with the 40dB simulated equal loudness curve. The transfer function of the filter [20] is given below in the normal frequency scale.

$$E(\omega) = \frac{\omega^4(\omega^2 + 56.8 \times 10^6)}{(\omega^2 + 6.3 \times 10^6)^2(\omega^2 + 0.38 \times 10^9)}$$

(EQ 2.9)

The function $E(\omega)$ is an approximation to the nonequal sensitivity of human hearing at different freqeuencies [19] and simulates the sensitivity of hearing near the 40-db level.

The $E(\omega)$ spectrum is similarly sampled at 19 equal Bark intervals, Ω=B, corresponding to the middle frequency of each critical band. The Silage description for the equal loudness pre-

emphasis is therefore, nothing more then weighting each critical-band coefficient with a constant, $E(\omega_i)$, for $\Omega_i(\omega_i)=i*B$ in accord to the Bark scale transformation given in EQ1.7.

2.3.3 Intensity-loudness Power Law

A cubic-root amplitude compression approximates the power law of hearing [21] and simulates the nonlinear relation between the intensity of sound and its perceived loudness. Together with the psychopathic equal-loudness preemphasis, this reduces the spectral-amplitude variation of the critical-band spectrum so that the following all-pole modeling can be done by a relatively low model order.

In order to simulate the power law of hearing, the critical band coefficients are raised to the power of 0.33. The equal loudness preemphasis and intensity-loudness Power Law are realized together by taking the natural log of the critical band coefficients, so that the constant multiplication in equal loudness preemphasis becomes a constant addition, which is cheaper computationally, while the raise power operation is equivalently realized by a constant multiplication. An exponential operation completes the manipulation, shown below:-

$$E(\Omega_i)\, X^{0.33}(\Omega_i) \quad \text{Exp}\{\, [0.33 * \ln[X(\Omega_i)]] + \ln[E(\Omega_i)]\}$$

(EQ 2.10)

2.3.4 Natural Log

The following describes how the natural log of a number, x is computed.

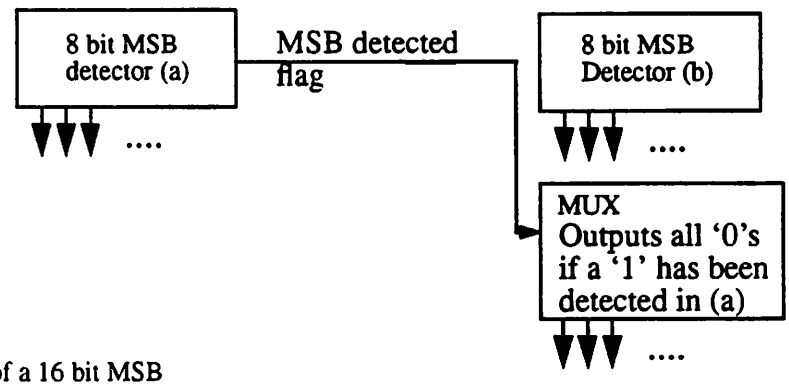
$$\begin{aligned} \ln(x) &= \ln(2^p * y) \\ &= p*\ln(2) + \ln(y) \\ &= p*\ln(2) + a1*y + a2*y^2 + a3*y^3 + a4*y^4 + a5y^5 \end{aligned}$$

$$\begin{aligned} a1 &= 0.99949556 \\ a2 &= -0.49190896 \\ a3 &= 0.28947478 \\ a4 &= -0.13606275 \\ a5 &= 0.03215845 \end{aligned}$$

where p = Most Significant Bit ==> $1 \leq y \leq 2$. This reduces the computation to a multiplication, p* ln(2), while ln(y) is evaluated by polynomial expansion. The value of p is obtained by a

MSB detector, while y ($= {}^{x}/_{(2^{P})}$) is obtained by shifting the binary of x to the right p times. Two implementations of the MSB detector were examined. A log detector and a linear detector.

The log detector, with order O(log n), may speed up the computation for worst case inputs (Speed of MSB $\alpha\ {}^{1}/_{p}$). However, from the view of power consumption, the log comparator is less efficient, as it will be comparing bits whose comparisons might have been omitted by the linear comparator. For example, the figure below shows a 16 bit MSB log detector used to detect the MSB of the bit string, 0000-0101-1011-0011. The bypass registers a '1' if a '1' bit has been detected in (a). The 8-bit MSB detector (b) will always perform its computations regardless of the output of detector (a). While gaining speed, this set-up does not consider the redundancies between (a) and (b).



e.g of a 16 bit MSB
log detector.

FIGURE 2.9 16-bit Log detector for MSB

A linear MSB detector, on the other hand, omits all further operations after the first '1' has been detected, thereby conserving energy. An improved linear MSB detector combines the MSB detection and the division by 2^{P} operation in a loop with n iterations (n= Bit length of x = 33) by taking advantage of the fact that a division by 2^{P} is equivalent to P binary right shifts. The algo-

rithm starts with P1=n and maintains 2 copies of binary stream X1 and Y1, which are both set initially equal to x,

```
for i:1..n,
    if leading bit (MSB) of X1 is a '0',
        X1 << 1;
        P1 = P1 - 1;    /* MSB of x not found yet. Continue shifting X1 to
                           the left by one step, and decrement P1 by 1 */
    else
        Y1 >> 1;        /* MSB of x detected. For each of remaining P1
                           iterations, shift Y1 to the right by one step */
```

Eventually, at the end of n iterations, P1=p, and Y1 would have shifted left by p times such that Y1=y.

The combined MSB detector and division by 2^P, therefore has a data-independent delay. This simplifies the design as the worst case input does not produce a delay that is any longer than a best case input. The critical path to compute the log for 19 critical band coefficients turned out to be 1027, which when run at a clock rate of 537ns, is only 0.551ms. This is not of much consequence considering the 10ms throughput time limit of the RASTA-PLP front end. It is chosen for its design simplicity and power advantages.

2.3.5 Exponential

The following describes how the exponential of a number, x is computed.

Let $e^x = 2^p$

$= 2^i * 2^f$ where i is an integer and $0 \leq f \leq 1$

Shift

Polynomial Approximation: $1 + b_1 f + b_2 f^2 + b_3 f^3 + b_4 f^4$

$b_1 = 0.693147$
$b_2 = 0.240227$
$b_3 = 0.055504$
$b_4 = 0.009618$

And P is found by:

$\ln(e^x) = \ln(2^p)$

$x = p * \ln(2)$

$p = x * {}^1/_{\ln(2)}$; constant multiplication

f is the decimal portion of P, with 2^f computed by polynomial approximation. i is therefore the integer portion of P, and 2^i is realized by binary shifts. If x is negative, i will be negative, and

2^i corresponds to i left shifts. Otherwise it is i right shifts. The critical path for computing 19 exponentials is 685, which, at a clock period of 550ns, is only 0.377ms. Again, well under the 10ms limit.

TABLE 6. Estimates obtained with Vdd=1.2V; Clock Period 550ns

Computation Block	Critical Path	Delay (ms)	Power (μW)
Log	1027	0.55	21.2
Rasta Filtering, Equal Loudness & Power Law	389	0.21	7.85
Exp	685	0.37	10.7
Total	2401	1.13	39.8

2.4 Inverse Discrete Fourier Transform, Autocorrelation Coefficients

This far, the algorithm has only involved obtaining the power spectrum and reshaping it in such a way to mimic the human perceptual hearing. From the modeled power spectrum, the autocorrelation function is obtained through an inverse discrete Fourier Transform. Since the 21 critical band coefficients cover the frequencies $0 \leq \omega \leq \pi$, the IDFT is given by,

$$r(n) = \sum_{k=0}^{40} R_p(k) e^{j\frac{2\pi}{41}kn} \qquad R(k) = R(40-k) \text{ for k} > 20$$

The power spectrum is real and even, which implies that the autocorrelation function is real. Ignoring all imaginary product terms,

$$r(n) = R_p(0) - R_p(20) + \sum_{k=1}^{19} R_p(k)\left(\cos\left(\frac{2\pi}{41}kn\right) + \cos\left(\frac{2\pi}{41}(40-k)n\right)\right)$$

$$= R_p(0) - R_p(20) + \sum_{k=1}^{19} R(k) w_{n,k}$$

$$w_{n,k} = \cos\left(\frac{2\pi}{41}kn\right) + \cos\left(\frac{2\pi}{41}(40-k)n\right)$$

However, only the first nine autocorrelation coefficients are needed for a 8^{th} order LPC analysis. Therefore, instead of attempting to use a general n^{th} order FFT (in this case, n=21)

which would compute all the 21 autocorrelation coefficients. It was decided that evaluating the first 9 coefficients by direct summation would simplify the design. The algorithm, as described in Silage, consists of a nested loop of depth 2 with the outer loop iterating from n=0 to n=8, and the inner loop iterating from k=1 to k=19. The $w_{n,k}$ values are stored in a 2-dimensional 9 by 19 array. The result is a partial IDFT which uses 11.2μW, and takes 0.6ms to complete. The figures are insignificant when compared with the overall power consumption or throughput of the entire front-end. Even if the power consumption of the IDFT block could be improved by 30%, the net effect in the global picture would still be negligible. The Silage description for this block can be found in Appendix I.

2.5 Durbin Analysis[3]

Equation 1.3 is solved here to obtain 8 cepstral values. The autocorrelation coefficients are first normalized by $^1/_{r(0)}$. There is no loss of generality by doing so. The cepstral values are solved iteratively by the Durbin algorithm. This solves the equation in $O(n^2)$ as compared to $O(n^3)$ if a more general Gaussian elimination approach is taken. The normalization of the coefficients requires an inverse operation (x^{-1}). In addition, each iteration of the algorithm also requires the inverse of a variable β. The front-end was simulated over a large pool of speech samples and it was observed that the values of r(0) are always in excess of 15, while the values of β are always between 0.7 to 1.0. The inversions are done by non-restoring division. In order to simplify the design of the divider, the operands of the inverse are adjusted so that the quotient $^x/_y$ observes the rule $x \leq y$. In this way, the absolute value of the output of the divisor is strictly less than or equal to 1. Therefore, there is no integer bit in the divider.

As $r(0) > 1$, $^1/_{r(0)}$ automatically satisfies the $x \leq y$ requirement. However, $^1/_\beta$ is processed in the following manner:

Since $(1.0 \le |\beta| < 0.7)$

$$\left|\frac{1}{\beta}\right| = \frac{1}{|\beta|} = \left(\frac{1}{|\beta|} - 1\right) + 1$$

$$= \frac{1 - |\beta|}{|\beta|} + 1$$

where $\frac{1 - |\beta|}{|\beta|} < 1$ is evaluated by the divider

Silage code for the Durbin's algorithm is given in Appendix J.

2.6 Cepstral Conversion

In accord with EQ 1.5, the Silage description in Appendix K uses a loop to iterate the summation. The gain is assumed, for simplicity, to be 1. So the first cepstral coefficient $c_0 = \ln(1) = 0$, a trivial solution. The $^1/_m$ factor needed for computation of all other coefficients is obtained from the array *minverse*, which can be implemented on an srom.

2.7 Clock Period

The single most delay comes from a 33-bit by 25-bit multiplier in the Natural Log block. Operating at 1.2V, clock overhead is added and the minimum time estimated for completion of the multiply is 1033ns. This aside, all the other multipliers have fewer bit lengths and usually have a delay between 600ns to 750ns. The adders have even lower delays, with the largest 25-bit adder having a delay of only 294ns.

Using a multi-cycle system, the execution of each EXU is scheduled over sufficient number of clock cycles to achieve a stable output. For instance, the operation of a 25-bit adder may take 2 clock cycles, while that of a 32-bit multiplier may take 5 clock cycles. This allows better utilization of a shorter clock period, reducing the amount of redundant time where certain EXUs may idle while waiting for the next clock input, and the overall delay of the system. Also, a shorter clock period results in more available cycles within the 10ms limit that the design

throughput must conform to. The additional cycles could allow more resource sharing, thereby reducing the overall area.

Nevertheless, the increased clock frequency of a multi-cycle system also results in higher clock power and control power due to the higher switching activity. Such trends are visible in an experiment where the properties of the FFT block with respect to different clock periods are investigated using Hyper's estimator. Result are shown in Table 7.

TABLE 7. Variation of properties with clock rate for FFT block

	Setup A	Setup B	Setup C	Trend of Property as Clock Frequency Increases
Clock Period	0.9μs	0.5μs	0.1μs	Decrease
Available Cycles within 10ms	11111	20000	100000	Increase
Critical Path	15815 (Exceeds 10ms)	18951	49485	Increase
Area	155.53 mm^2	137.76 mm^2	33.11 mm^2	Decrease
ENERGY (nJ)				
Registers	427	431	504	Increase
Control	112	134	707	Increase
Bus	925	870	428	Decrease
Clock	450	508	967	Increase
Sub-Total of Register, Control, Bus and Clock Power	1914	1943	2606	

Table 7 shows that using a faster clock, the control power as well as clock power are increased, as expected. Critical path also goes up as a result of EXU's which execute over multiple cycles, and also due to the use of fewer hardware units, which was made possible by the increased available cycles within the 10ms time margin. The decrease in bus power with a faster clock rate can be attributed to the reduced area, which meant shorter interconnects with less capacitance.

In view of the drive for low power, the clock period was set near the maximum value that would still meet the 10ms throughput criteria. The multi-cycle system was set up such that run-

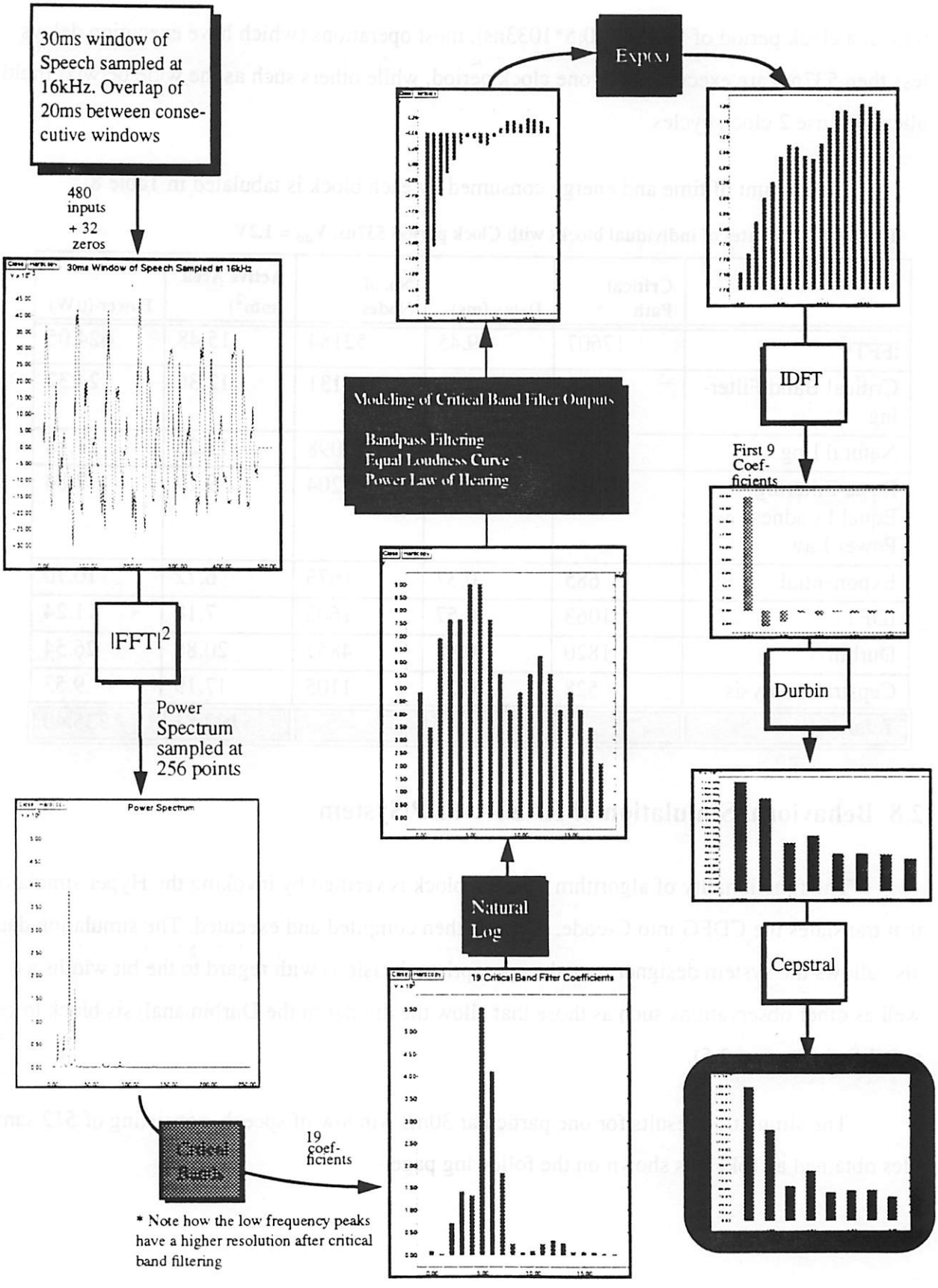


FIGURE 2.10 Behavioral Simulation of System

ning at a clock period of 537ns (~ 0.5*1033ns), most operations (which have execution delays less then 537ns) are executed over one clock period, while others such as the wide betwixt multiplications use 2 clock cycles.

The amount of time and energy consumed by each block is tabulated in Table 8.

TABLE 8. Estimates of individual blocks with Clock period 537ns, V_{dd} = 1.2V

	Critical Path	Delay (ms)	No. of Nodes	Active Area (mm^2)	Power (μW)
$\|FFT\|^2$	17607	9.45	52184	15.48	624.05
Critical Band Filtering	1043	0.56	4131	12.36	24.32
Natural Log	1027	0.55	3098	13.23	21.18
Rasta Filtering, Equal Loudness & Power Law	389	0.21	1204	9.79	7.84
Exponential	685	0.37	1673	6.72	10.70
IDFT	1063	0.57	1603	7.18	11.24
Durbin	1820	0.98	4851	20.86	26.54
Cepstral Analysis	528	0.28	1105	17.19	9.53
Total				102.81	735.40

2.8 Behavioral Simulation of RASTA-PLP System

The functionality of algorithm for each block is verified by invoking the Hyper simulator that translates the CDFG into C-code, which is then compiled and executed. The simulation data also allows the system designer to make appropriate decisions with regard to the bit widths as well as other observations such as those that allow the divider in the Durbin analysis block to be simplified (Section 2.5).

The simulation results for one particular 30ms window of speech, consisting of 512 samples obtained at 16kHz is shown on the following page:-

3.0 Architecture

For each of the two blocks, (FFT and critical band filters) the Hyper hardware mapper was used to create a register transfer level (RTL) netlist described in part by Structural Description Language (SDL), with the combinational logic of control signals described in Bdsyn. Mapping transforms the decorated flow graph into three types of structural cells; the datapath-structure cells, the controller state-machine cells, and the controller-interface cells. The datapath-structure (dp) cells describe the use of hardware instances referenced from the a low-power hardware library. The controller-interface cells work with the controller state-machine cells to determine the relation between global control signals (clock and current state) and the local datapath controller signals.

The architecture adopted makes extensive use of registers. Each EXU (e.g. adder, multiplier, shifter, etc.) obtains its operands by reading from some set of registers, and outputs its result to another set of registers.

*At the positive rising edge of the clock, both **RD** inputs of Reg_1 and Reg_2, and the **WR** input of Reg_3 are set on high.*

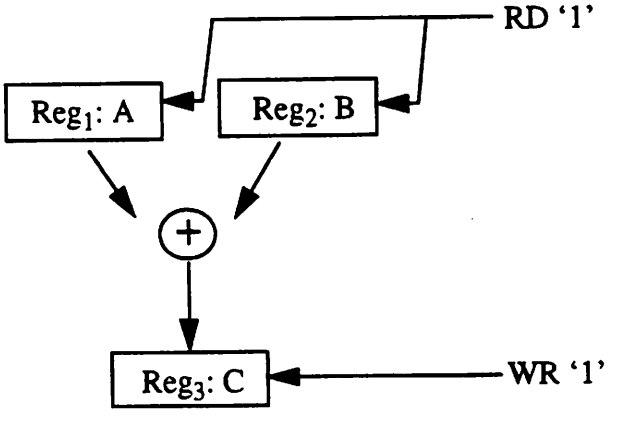


FIGURE 3.1 Example of an addition operation in one clock cycle: C = A+B

(The complete sdl netlist generated by the Hyper hardware mapper for both the FFT and Critical Band block can be retrieved via anonymous ftp from infopad.EECS.Berkeley.EDU/pub/yeo/sdl.)

Two essential steps in the hardware mapping are register-file recognition, and multiplexer reduction. They will be described in detail in the sections 3.3 and 3.4. Register-file recognition merges individual registers into register files. Since all registers in a register file share a common input and output bus, merging registers within a datapath cell reduces the number of data wires within the cell. Multiplexer reduction reduces the number of data bus multiplexers and number of global buses.

The hardware mapping step produces the SDL/Bdsyn description, which is input to the LagerIV silicon assembly environment. The LagerIV tool set is used to convert this architectural description to silicon layout in Magic format.

3.1 Clock Timing

The immediate advantage of using such an architecture is that it allows usage of the LagerIV hardware libraries without the need for investigating elaborate timing issues. The timing constraints are met by a multi-cycle system discussed in Section 2.7.

3.2 Merging of Hardware EXUs

The radix2-Winograd FFT algorithm uses 3 different fixed point representations for its intermediate data: fix<13,11>, fix<22,12> and fix<32,17>; where fix<α,β> means a total of α bits with β decimal bits. The majority of the EXU executions operate with inputs and outputs represented by fix<22,12>. It would be costly to allocate 3 different kinds of EXUs for each type of operation (add, multiply, etc.) in order to handle the 3 different data types. Instead, resource sharing is enhanced by an 'operation merge' option during module selection in Hyper. Each operation is replaced by one with the same functionality, but with the largest number of bits as required by the algorithm. For instance, if an algorithm contains both 8-bit as well as 16-bit additions, 'operation merge' replaces the 8-bit addition with a 16-bit operation. Thus, only one 16-bit adder is required to be allocated, as opposed to one 16-bit and another 8-bit adder required by the original algorithm.

Using 'operation merge' during module selection, operators with the same functionality share the same bit widths. In the FFT block, all adders are 32-bits wide, all subtractors 22 bits, and all multipliers, 22 bits.

3.3 Interconnect

By default, the original netlist has a flat structure. Each EXU, along with the necessary input registers, multiplexers, and output buffers, are mapped into a single datapath cell. Each block of memory, on the other hand, is mapped onto 3 separate datapath cells: input data selection datapath, memory instance, and output buffer. This easily results in about twenty separate datapath cells, and an equal number of Ctl cells, each controlling the operation of hardware units within a datapath cell. Layout placement of these cells is difficult with such a large number. Since there is extensive amount of resource sharing, almost every EXU has an output which is connected to the input of every other EXU. Global routing becomes a significant task in the layout generation.

The complexity of the interconnect, could be reduced by recognizing and taking full advantage of the regularity of the algorithm. The Critical Band Filter block serves as a good example to illustrate this.

In the critical band filters block, each set of the 19 nested-loops iterations has sentences that look like:

```
Filt105 = word(a15 * cb10[i15-30]) + Filt105#1;
Filt114 = word(a15 * cb11[i15-36]) + Filt114#1;
Filt123 = word(a15 * cb12[i15-43]) + Filt123#1;
```

additionB
multiplicationA additionC

FIGURE 3.2 Typical Sentences within Nested Loops found in Silage description of the Critical Band Filters

with not more then 4 sentences per iteration. The address generation (e.g. *i15 - 36*, for address of data in the *cb11* array to be read) is handed by an addition to a constant negative number (e.g. *-36*). Thus, each of the above sentences requires 1 multiplication (operation ***multiplicationA***) and 2 additions (operations ***additionB*** and ***additionC***).

A total of 3 multipliers(M1, M2, and M3), and 3 adders (A1, A2, and A3) were allocated. Note that the multipliers and adders could have been assigned consistently and systematically such that the output of A1 (handling ***additionB***) always goes to input of M1 (handling ***multiplicationA***), whose output, in turn goes back to A1 (now handling ***additionC***). Similarly, output of A2 goes to input of M2, whose output goes back to A2, etc. By allowing the output of one adder to be consistently connected to only 1 multiplier, the interconnect between the adders and the multipliers can be reduced. This however, puts some restriction on the scheduling. Currently, the automatic assignment generated by Hyper is inconsistent between different sets of nested loops, and therefore requires connections between inputs/outputs of all 3 multipliers with all 3 adders. Any scheduling of the sort just mentioned would therefore have to be done manually by editing the CDFG.

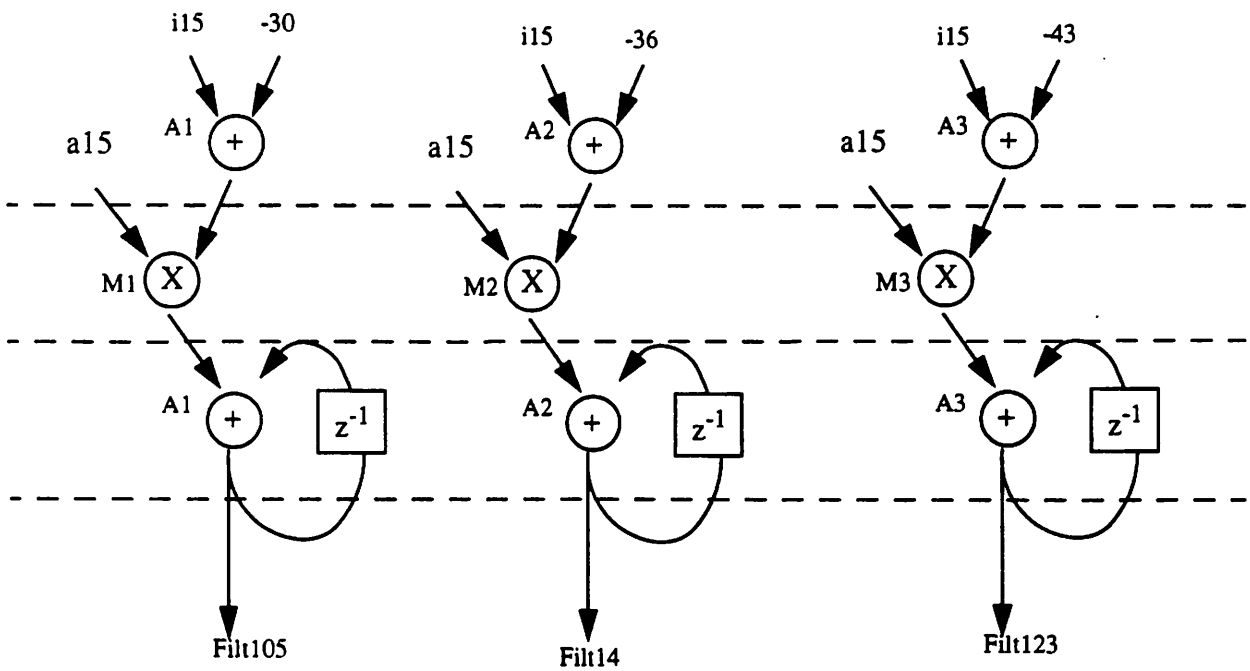


FIGURE 3.3 Improved Scheduling of regular Silage code.

On the other hand, the Hyper hardware mapper does allow the user to partition the interconnect problem by merging separate datapath cells. Ideally, merging is done for datapath cells with a strong data correlation. such that the hardware of cells A and B are located near one another on the layout. It reduces both the total number of input/outputs seen at the top hierarchy (global), as well as the length of interconnect between A and B. However, it was observed that the SDL syntax was a limiting factor in the amount of merging that could be done.

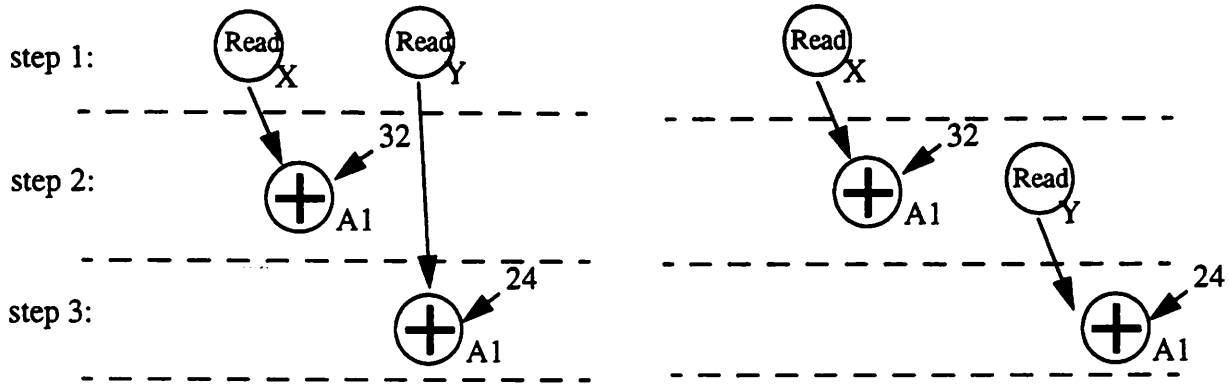
The SDL syntax requires that merged datapath cells have modules that share a common bit-width because one-dimensional placement strategy is employed in the datapath cells. Each module consists of a number of bit slices stacked vertically, such that the datapath cell has a uniform height dependent upon the bit-width of its modules. Section 3.6 discusses more of this tiling scheme.

Since hardware merging was done during module selection, the adders in the FFT block are all 32-bit wide, the subtractors 22-bit wide, and the multipliers, 43-bit wide. Likewise, the adders in the critical-band block are 13-bit wide, and the multipliers, 25-bit wide. As a result, merging can only be done, and is only done, to combine instances with the same functionality, i.e. 32-bit adders with 32-bit adders, or 25-bit multipliers with 25-bit multipliers). Nevertheless, the amount of global interconnect is reduced because some adder-to-adder connections are contracted into one lower hierarchy.

3.4 Registers

Register file recognition merges individual registers into register files.Within a datapath cell, the registers that are merged into a register file share common input and output terminals. Merging registers therefore reduces the number of local terminals. The registers are merged according to the functionality of their output node. For instance, the input registers of an adder are merged into 2 register files representing the left operand and the right operand of the addition respectively. The input registers of a write array node, on the other hand, are merged into 2 register files representing the address, and the data to be written respectively.

Each register file allows only one READ or one WRITE (a simultaneous READ and WRITE can be achieved by reading from a register and writing to a different register) over each clock cycle. Therefore, the scheduler must avoid the production of multiple inputs to the same register file at the same time step.

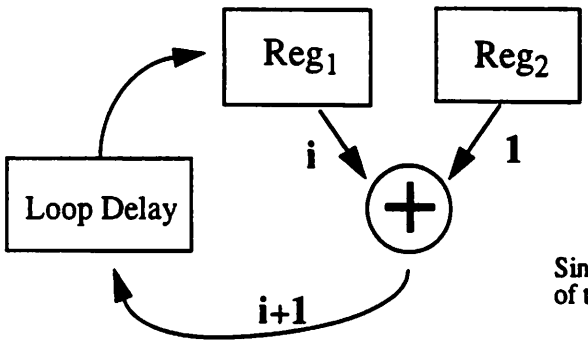


Scheduler Multiple write Error: Outputs of both Read Nodes, X&Y are left operands of the adder A1, so the outputs are stored in the same register file. Both outputs are produced concurrently in step 1. However the register file would not be able to simultaneously write both data at step 1.

Scheduler Simultaneous Read/Write Constraint: It is necessary during step 2, to read the value of X from the same register file as that in which the value of Y is written to. Therefore, the registers used for storing values of X & Ymust be different.

FIGURE 3.4 Read/Write conflicts and constraints of Register files. Note that there is only one instance of an adder, A1, available at any time step.

However, within an iteration loop, the scheduler can sometimes mistakenly schedule the production of a variable (e.g. the iteration counter) for the next iteration at the same time as the consumption of the variable during the current iteration. The situation causes simultaneous READ and WRITE to the *same* register, and a race condition ensues. An example in Figure 3.5 illustrates this:



Consider an iteration loop:
for i=1 to 10 do ...

Then there is an iteration counter adder that does:
i = i+1

Since the value of **i+1** is going to be used during the execution of the adder in the next iteration, it is stored back in $\mathbf{Reg_1}$.

FIGURE 3.5 Race Conditions during simultaneous READ and WRITE to the same register Reg_1 if Loop Delay is not implemented as an additional register.

Such a problem arises because the loop delay node in the CDFG is mapped onto a transfer EXU. In other words the hardware does not explicitly implement the delay. The solution calls for manual changes to store the value **i+1** temporarily in another register, $\mathbf{Reg_3}$ and then read the data from $\mathbf{Reg_3}$ into $\mathbf{Reg_1}$ at a later time step. This is equivalent to mapping the loop delay node onto a temporary storage. The additional transfer of data between $\mathbf{Reg_3}$ and $\mathbf{Reg_1}$ may increase the critical path by 1.

3.5 Control Signals

Control signals in the design are described by the controller cells, which is the combination of controller state machine cells, and an interface cell. Each datapath cell has a local controller cell which provides the output control signals in relation to the inputs, which are namely, the clock and current state. The use of local control signals reduces the complexity of layout/ placement by reducing the total number of global control signals. Current state is generated by a global finite state machine.

The relationship between the local control signals and global control signals is described textually as a truth table. From this, Bdsyn produces a collection of logic functions which implement the described function. The logic functions were implemented by a combination of standard cells[1] from the low power library with the aid of misII, as part of the LagerIV synthesis process.

The generation of the control blocks' layout from the state-machine description usually represent about 70% of the total LagerIV synthesis process CPU time.

Within the each datapath cell, the control signals are required to:-

- Control multiplexers which select the appropriate inputs to register files, from a number of input data buses
- Control the READ/WRITE operations of register files and memories, including srams and sroms.
- Control activity of tristate buffers which write data onto global buses. Several different buffers may have their outputs connected to the same bus, but there can be no more then one buffer, at any time, writing data onto the bus, while the others are tristated.

The extensive use of registers (Table 9) in the architecture requires not only an equally huge number of Read/Write control signals, but also average thrice as many multiplexer control signals which select the input to the register files. Generation of these control signals result in both additional area (control area), as well as power consumption (control power). It turns out from SPA results that control power is a major source of power consumption for this design. The result is not surprising, considering the high number of registers used, which is a good indication of the amount of control activity.

TABLE 9. Use of Registers in FFT and Critical Band Filters blocks

Property	FFT	Critical Band Filters
Number of Registers	116	250
Percentage of active area the registers occupy	20	40

3.6 Example of a typical Datapath cell: 16-bit Adder

This section looks at the components of a typical datapath cells which performs the functionality of a 16-bit adder.Besides the 16-bit adder itself, the datapath cell also describes the nec-

1. Standard cells here include a multitude of AND gates, AND-OR gates, NAND gates, buffers, flip-flops, multiplexers, etc.

essary registers that are merged into register files, additional mutiplexers which select from a multitude of 7 distinct inputs data buses, and 7 buffers which write the adder output onto as many global buses. The latter consists of both tristate buffers as well as conventional buffers that simply copy the inputs to the large output data buses.

The adders in the FFT block are all 32-bit wide. For simplicity, all the adders, registers, etc. in this example have been reduced to 16-bit wide. The hardware declarations are shown in Figure 3.6.

```
(subcells
    (add
      ( add160)
      ((N 16)(CS_TYPE "z"))
    )
    (regfile
      ( add160_reg1)
      ((N 16)(R 16)(NC 6)(REGPLANE '("0000000000000011" " 0000000000000100"
      " 0000000000000101" " 0000000000000110" " 0000000000000111" " 0000000000001000"
      " 0000000000010000" " 0000000000100000" " 0000000001000000" " 0000000010000000")))
    )
    (regfile
      ( add160_reg0)
      ((N 16)(R 5)(NC 3)(REGPLANE '("0000000000000001" " 0000000100000000")))
    )
    (mux
      ( mux_15 mux_16 mux_19 mux_20 )
      ((N 16)(NUM_IN 4))
    )
    (mux
      ( mux_18 mux_22)
      ((N 16)(NUM_IN 3))
    )
    (buf
      ( buf_14 buf_18 buf_20)
      ((N 16)(SIZE "1"))
    )
    (tribuf
      ( buf_15 buf_16 buf_17 buf_19)
      ((N 16)(SIZE "1"))
    )
)
```

FIGURE 3.6 Declaration of Modules in a Typical datapath cell of a 16-bit adder.

Each register file selects its write input data from 1 of 8 input buses. This is done with (Figure 3.7) a combination of 2, 3, and 4-input multiplexers available from the hardware library. The permutation of these depends on the number of inputs.

Out of the 7 output buffers, 4 of them are tristate buffers which allow the output global buses to be multiplexed. Multiplexing of global buses allows reduction of total interconnect area, at the expense of additional area of tristate buffers. In this case, each tristate buffer is about 25% larger then the conventional buffers used to drive non-multiplexed buses.

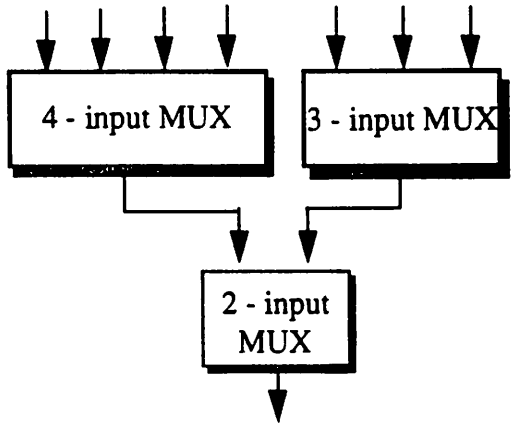


FIGURE 3.7 Selection of input from 7 input data buses.

Table 10 shows a breakdown of the area. Although the task of the datapath cell is to perform a 16-bit addition, the actual area used by the 16-bit adder is only 8%. The overhead of all the registers, multiplexers, etc. should therefore not be overlooked. More importantly, the interconnect represents an alarming 28% of the area.

TABLE 10. Hardware Allocation for 16-bit Adder and related Control Signals and Registers

Type of Hardware	Number of Units	Area (mm^2)	Percentage Total Area
16-bits Multiplexers	6	0.37	20
16-bit Registers in 2 Regfiles	17	0.62	33
16-bit Tristate Buffers	4	0.15	8
16-bit Buffers	2	0.06	3
16-bit Adder116	1	0.15	8
Interconnect		0.53	28
Total		1.88	100

Since data often flows through a datapath in a fairly linear fashion, one-dimensional placement strategy is employed in datapath cells. Each module consists of 16 bit slices stacked vertically as shown in Inset A of Figure 3.9. Figure 3.8 shows a bit-slice of a typical datapath block. As the number of interconnect wires that need to transverse vertically in the routing channel increases, the width of the channel has to be increased. It therefore extends the layout horizontally. Also all data inputs/outputs are located on the extreme right of the datapath cell (Figure 3.9). There is a necessity to be able to feedthrough data from the inputs on the far right to all the hardware modules. Arguably, the interconnect area could have been optimized by less rigid placement strategy of hardware units, and inputs/outputs connected to the nearest side of the datapath cell. However, this only complicates the design space with little improvement in terms of global results. It is also likely to make the global routing of buses more difficult. Instead of providing just one routing channel from a side of the datapath cell, it would then require channels on all 4 sides.

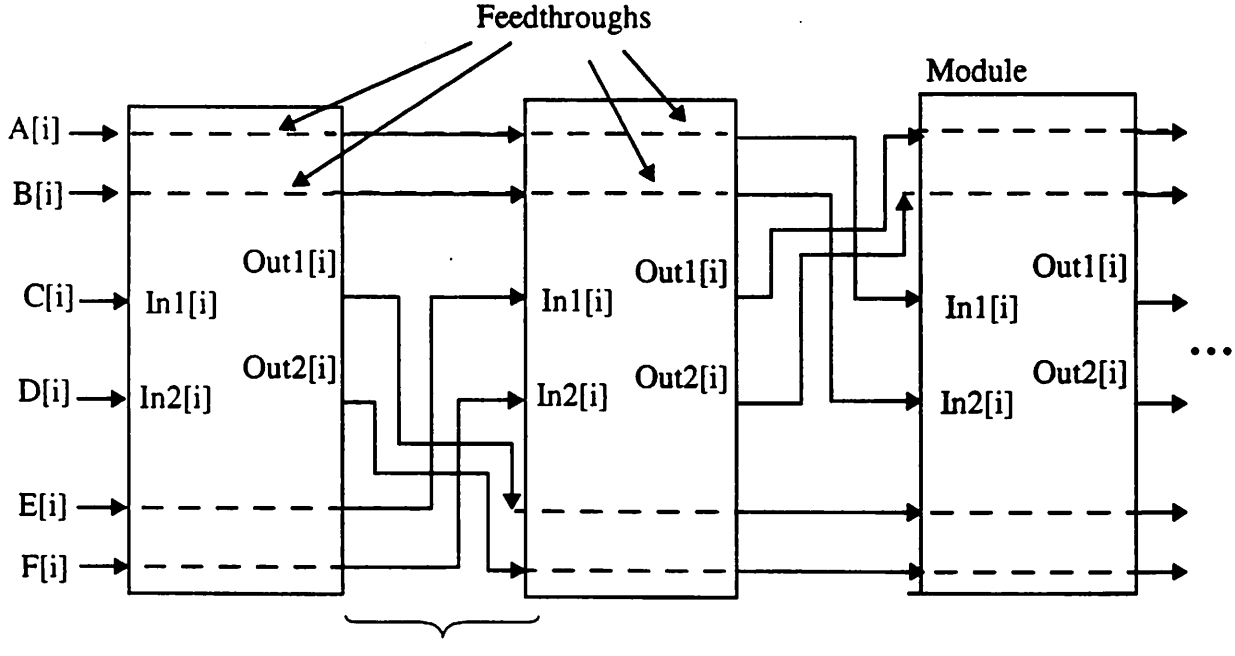


FIGURE 3.8 Placement strategy for a representative bit-slice of a datapath complex

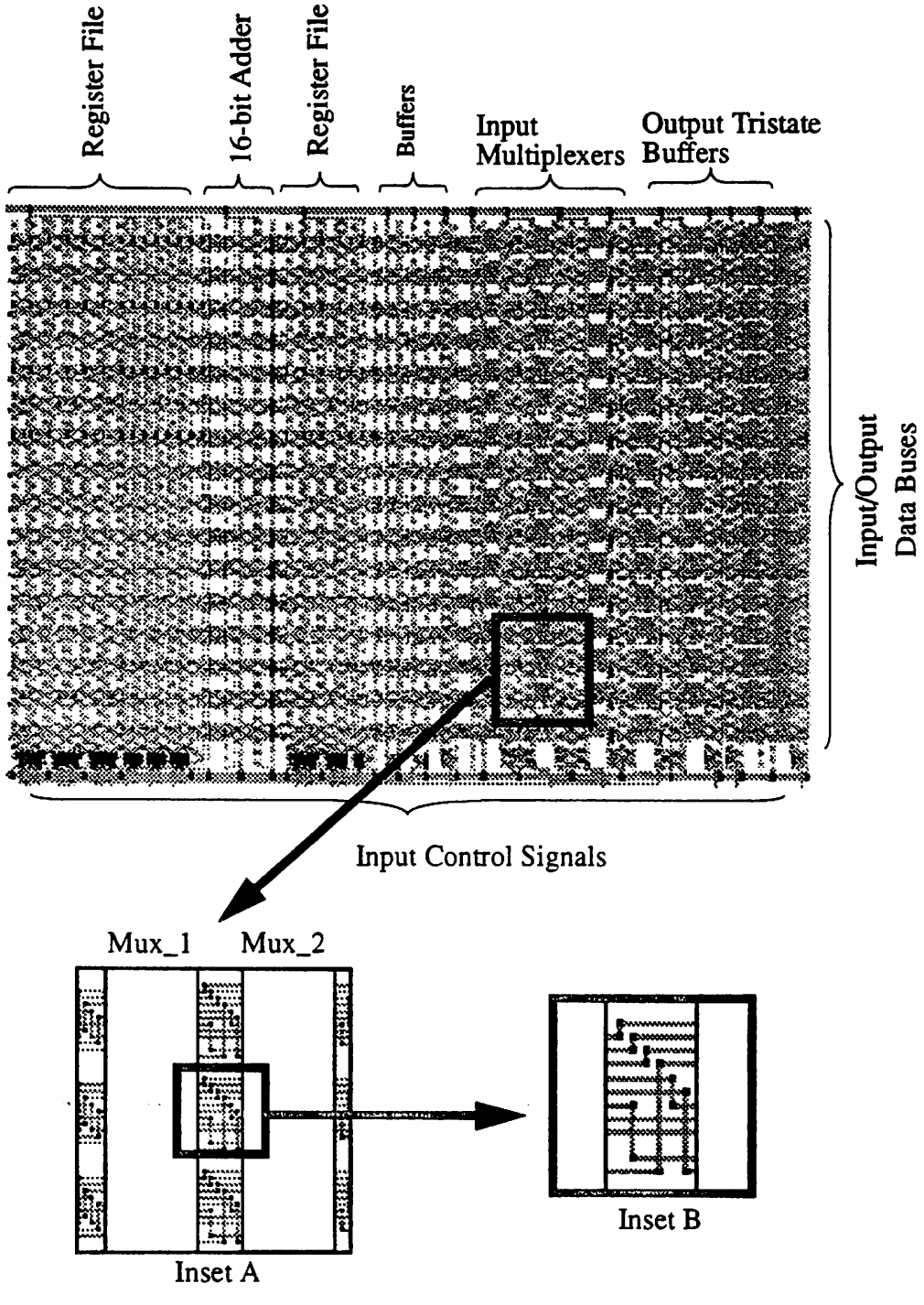


FIGURE 3.9 A 16-bit adder with input multiplexers, output buffers and register files. INSETs: Interconnect between adjacent multiplexers.

3.7 Layout

The designs of the FFT block and the Critical Band Filters were taken through to the layout level. Layout of each datapath cell was generated automatically by **dpp**, using also, the layout cells of hardware modules from the low_power library. Control cells are put together by **Bdsyn** using the standard cells as described in Section 3.5. The top hierarchy placement and channel definition was, however done manually, as the search space was much larger then that of individual cells, and the output of an automatic placement by Flint at the top level was generally unacceptable.

In order to handle the large number of interconnects, a central column was devoted for global interconnect routing. Local interface cells are placed next to the datapath cells that they control. Since the local interface cells are usually only a fraction the area of the respective datapath cells, they fit into the little spaces sandwiched between bigger, datapath cells.

Details of the layout are as follow:

TABLE 11. Hardware Allocation for $|FFT|^2$ block

Type of Hardware	Number
9-bit Log Comparator	1
22-bit Multipliers (22bit x 22bit)	4
22-bit Subtractor	2
32-bit Adders	2
Muxes	104
Registers	116

TABLE 12. Hardware Allocation for critical band filters block

Type of Hardware	Number
9-bit Log Comparator	2
25-bit Multipliers (13bit x 13bit)	3
13-bit Adders	3
Muxes	78
Registers	250

TABLE 13. Layout Dimensions using 1.2μm SCMOS Design Rules (λ = 0.6μm)

Design	Height (mm	Width (mm)	Area (mm^2)	No. of Transistors
$\|FFT\|^2$	17.9	12.2	218	400,000
Critical Band Filters	11.0	8.08	89.02	180,000

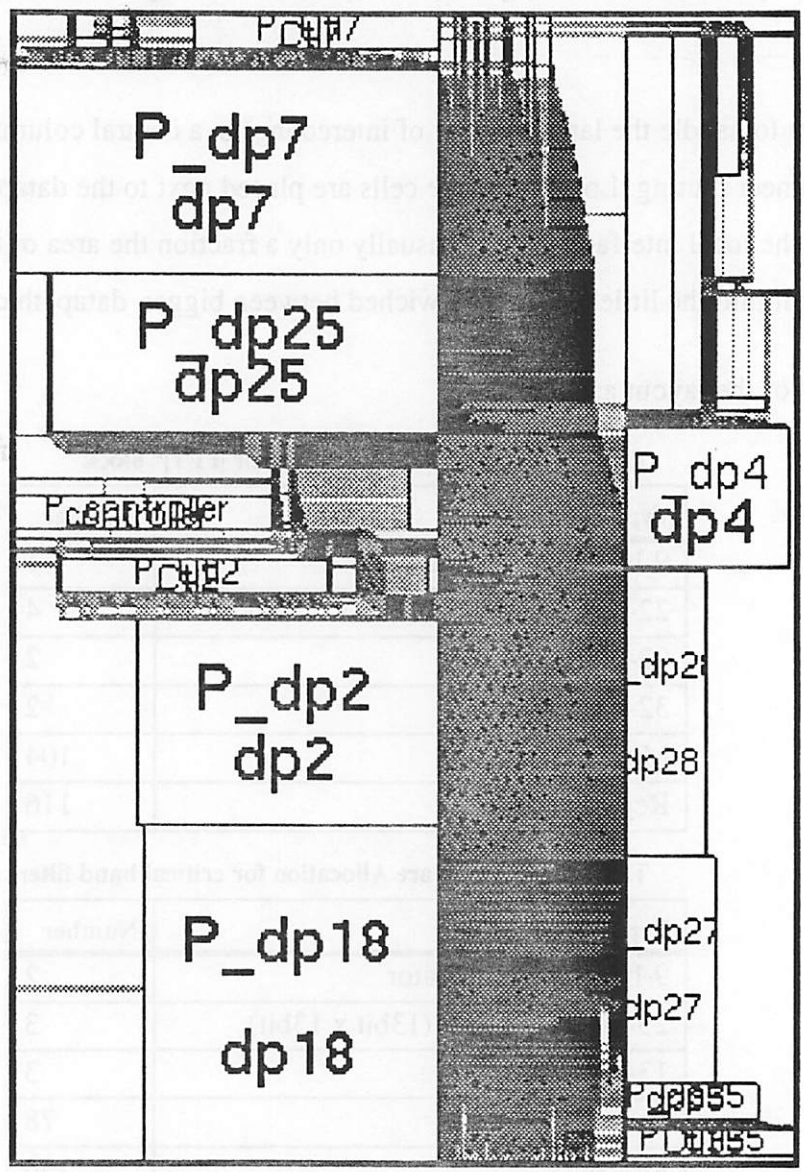


FIGURE 3.10 Layout of FFT

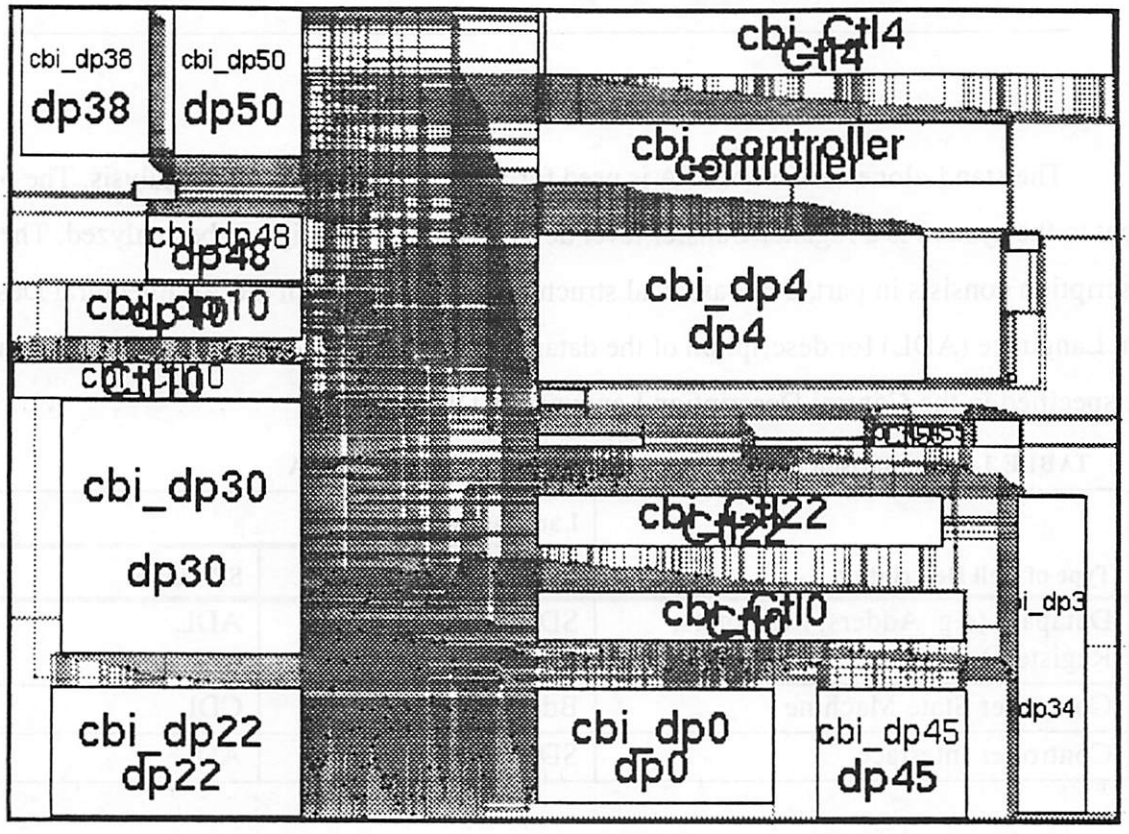


FIGURE 3.11 Layout of Critical Band Filters

4.0 SPA Analysis of FFT Architecture

The stand-alone version of SPA is used for architectural power/area analysis. The primary input to the system is a register-transfer level description of the design to be analyzed. The description consists in part, a hierarchical structural netlist written in the Architectural Description Language (ADL) for description of the datapath. The datapath control signals in the structure are specified in the Control Description Language (CDL).

TABLE 14. Comparison between input languages to LagerIV and SPA

	Language used:	
Type of Cell Description	**LagerIV input**	**SPA input**
Datapath (e.g. Adders, Memories, Registers)	SDL	ADL
Controller State Machine	Bdsyn	CDL
Controller Interface	SDL	ADL

Given the architectural description, SPA leads the user through four phases of analysis: parsing, VHDL analysis, VHDL simulation, and power/area analysis. In order to accurately account for the effect of hardware activity on power, the system performs functional simulation of the architecture to gather activity statistics. Therefore, the user must also provide a set of input vectors to be used during simulation. These input vectors were chosen from a set of speech samples with both voiced and unvoiced data, normalized so that the maximum magnitude of the entire set of data is 1, and quantized to 13 bits, with 11 decimal bits.

4.1 Input Description

Both ADL and CDL are Lisp-like languages. ADL is used for specifying the structural view of an architecture, while CDL is used for specifying the control path components in terms of high-level symbolic truth tables. Parsing of the ADL/CDL description translates the netlist into VHDL description. Besides describing the primary functions of the EXU's and their control signals, the VHDL description also includes the necessary SPA processes that monitor the hardware switching activity during VHDL simulation.

The main tool used for the design, Hyper, only outputs a netlist in either VHDL or SDL/Bdsyn description after its hardware mapping step. While the structure of ADL/CDL closely resembles that of SDL/Bdsyn, a SPA-compatible VHDL description was the actual code that would be simulated to obtain the switching statistics. The SPA-compatible VHDL is obtained by parsing the ADL/CDL description, and consists of a description of the architecture, as well as additional SPA processes (DBT and ABC [13] models) required to generate the simulation data necessary for the SPA power analysis.

Thus, a designer could either choose to translate the SDL/Bdsyn output of the Hyper hardware mapper to ADL/CDL description, or append the SPA processes to the VHDL output of the hardware mapper. Either way has to be done manually as there is currently no parser that can automatically handle the SDL/Bdsyn translation or the VHDL correction. It was decided finally that the simplicity of the ADL/CDL language made it preferable over the VHDL description. In addition, ADL was structurally a subset-like of SDL. Both describe a datapath in terms of hierarchical cells built from a hardware library. Both make use of a set of parameters to indicate variables such as the bit widths of the EXU, or word sizes of memories. But while ADL treats its inputs/outputs as symbolic variables or integer variables quantized to the specified number of bits, SDL treats all its inputs/outputs as nets made up of binary bits. It turns out that this abstraction actually makes description of the control tables easier then the bit stream method in Bdsyn.

The following sections 4.1.1 and 4.1.2 describes the translation process from SDL/Bdsyn to ADL/CDL. The knowledge will be helpful towards any future development of an automated translator.

4.1.1 SDL to ADL translation

The differences between ADL and SDL, using a 32-bit register file example with 4 registers, are highlighted in Figure 4.2.

The initial task during SDL to ADL translation is to consolidate all the control signals, and rename them in terms of symbolic variables. These control signals include the register file READ/WRITE controls described, as well as other control signals that determine the execution

patterns of tristate buffers, multiplexers, and memories. Table 15 shows a comparison between the symbols used in ADL and those used in SDL.

TABLE 15. List of Control Variable differences between library cells written in ADL and SDL

Type of Cell	Type of ADL Control	Type of SDL Control
Register File	1 RD Terminal & 1 WR Terminal; Integer inputs which determine which register to Read/Write	N RD terminals, and NC WR terminals for N registers, with NC of them storing non-constant values. At any time, there can only be at most one "1" bit on any of the N RD terminals, and NC Write terminals respectively.
Tristate Buffers	FN terminal. Input Symbols: {TRI (tristate), OE (operation enable)}	EN (enable) bit control
Multiplexers	SEL terminal. Input Symbols: {SEL1, SEL2, etc.}	Binary number inputs. A 2-input mux will be controlled by a bit. 3 or 4-input muxes input a 2-bit binary number: '00' selects data1, '01' selects data2, etc.
SRAM	FN terminal. Input Symbols: {RD, WR, NOP (No Operation)}	WRITEB bit which determines Read or Write operations. If NOP is desired, the Clock input is gated.
SROM	FN terminal. Input Symbols: {RD, NOP}	None. Operation is purely Read only. If NOP is desired, the Clock input is gated.

The other major task in the translation from SDL to ADL is the bit manipulation during connection of some of the buses. For instance, the data on a 8-bit bus can be shifted 1-bit left/right by simply hardwiring.

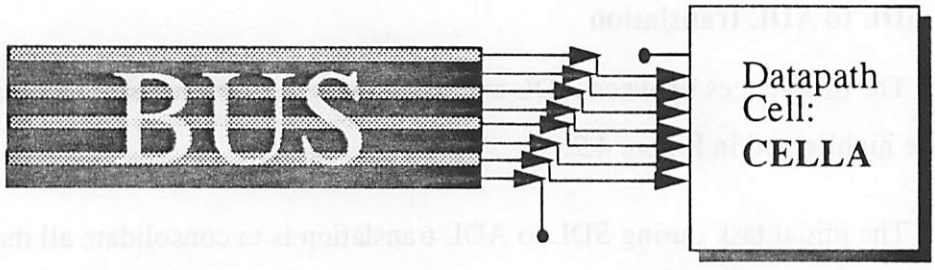


FIGURE 4.1 Hardwiring Connection between data bus and datapath cell, CELLA.

ADL

Declares add320_reg0 as a register file with 4 registers and 32 bits (parameters 4 32) and make net connections.

```
(instance add320_reg0 regfile (parameters 4 32) (nets mux_22_out
                add320_reg0_out
                add320_reg0_RD
                add320_reg0_WR
                CK1 ))
```

mux_22_out and add320_reg0_out are represented as integers within range of 0 to 2^{32} - 1.

Read and Write signals are described in terms of integer symbols. e.g. a '2' on add320_reg0_WR would set the WRITE signal of register 2 on.

SDL

Declares add320_reg0 as a register file (regfile) with 32 bits (N) and 4 registers (4)

```
(regfile
      ( add320_reg0)
        ((N 32)(R 4)(NC 3)(REGPLANE '("00000000000000000000000000000001")))
)
```

Describes a constant value to be stored in one of the 4 registers.

Connects nets to instance of add320_reg0

```
(instance add320_reg0 (
   (IN mux_22_out)
   (OUT add320_reg0_out)
   (CLK CK1)
   (RD CtlSgl_46 (term-index 0))
   (RD CtlSgl_47 (term-index 1))
   (RD CtlSgl_48 (term-index 2))
   (RD CtlSgl_49 (term-index 3))
   (WR CtlSgl_43 (term-index 0))
   (WR CtlSgl_41 (term-index 1))
   (WR CtlSgl_66 (term-index 2))
))
```

mux_22_out and add320_reg0_out are represented by a 32 bit. wide net.

Read and Write signals are described in terms of individual bits. A high bit sets the READ/WRITE signal of a register on.

FIGURE 4.2 Example of a 32-bit register file with 4 registers in both ADL and SDL descriptions

This can be described in SDL by:-

```
instance cellA (
        (busdata CELLAinput (width 8) (term-base 1))
)
```

which would simply connect *busdata* pins 0 through 6 to *CELLAinput* pins 1 through 7.

In ADL, however, there is no such bit-wise representation of the data. If the bus is 8-bits wide, its data is represented by an integer ranging from 0 through 255 (2^8-1). In order to achieve the same result as a left-shift of one bit, the following behavior is added in the VHDL code:

CELLAinput = (busdata * 2) rem 256;

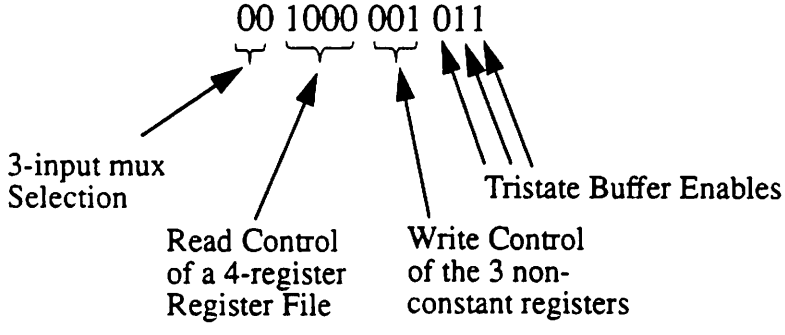
which basically shifts the busdata data by 1 bit to the left, and truncates to 8 bits wide with the remainder function. Table 16 summarizes the methods used to manipulate bit connections.

TABLE 16. Operations used in VHDL to simulate bit operations

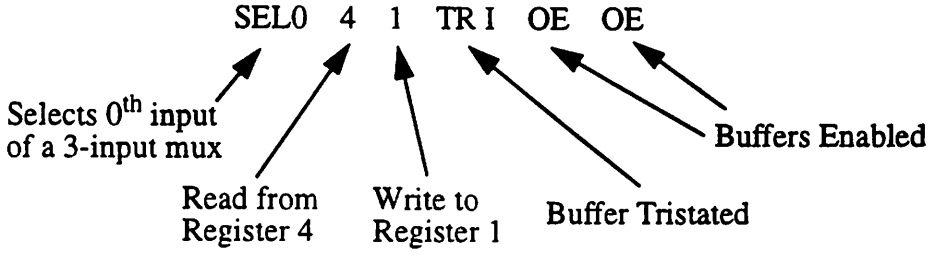
Type of Bit Manipulation	Type of Operation
Left Shift by N bits	$* 2^N$
Right Shift by N bits	$* 2^{-N}$
Truncate to lower M bits	rem 2^M
Extract the M MSB bits from a N-bit data.	$2^{-(M-N)}$; Equivalent to Right Shift by M-N bits

4.1.2 Bdsyn to CDL translation

Since the control signals in the SDL description of the architecture are all binary bits, the Bdsyn description of the control cells describe a bit stream output in relation to the 2 global control signals, the Current State, and the clock. A simple example would be a 12-bit wide output bit stream:



On the other hand, the CDL description describes these signals in terms of symbolic variables. Thus it would be necessary to decipher the **Bdsyn** bit stream output and replace it with several different control variables. For the 12-bit example, the equivalent CDL description would be:



The interface cell identifies the kind of functionality of each bit-position in the bit stream. This information is compared with the **Bdsyn** output bit stream to provide the symbolic CDL control variables.Since the functionality of each bit-position varies from cell to cell, a separate script was written for each control cell to analyze the bit stream output and translate it to a series of control variables. The process could be automated in future by a fairly simple program which builts a hash table that specifies the relationship between type of control and the position of the bit in the bit stream, and then compares the **Bdsyn** description with the table, to give the correct symbolic CDL control variables.

4.2 Simulations

After parsing the ADL/CDL architecture description and analyzing the VHDL description, the Synopses VHDL simulator was used to gather data on the switching activity. An approximately 3.2s segment of speech, sampled at 16kHz is shown in Figure 4.3. It is impractical to simulate the entire segment becuase the full simnulation[1] of each set of 512-point FFT takes about 2 hours. Therefore, only 4 sets of data, each consisting of 512 samples which represent a 30ms window, were chosen for simulattion. They roughly represent inputs with both highest power, lowest power.

1. Full simulation refers to the entire simulation of the 10ms required to obtain the FFT output from a set of data consisting of 512 samples.

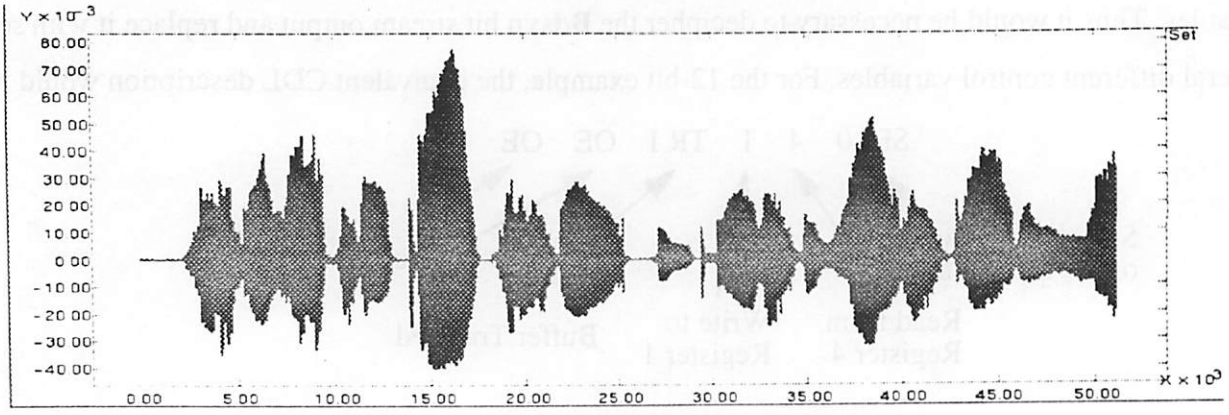


FIGURE 4.3 Approximately 3.2s segment of speech sampled at 16kHz. Since simulation of entire segment is impractical, 4 segments of 30ms window were selected for VHDL simulation in order to obtain the switching statistics for power analysis.

4.3 Power Analysis

The SPA analysis charts for one particular 512-point FFT is shown in Figure 4.4, with the numerical results consolidated in Table 17. They provide a breakdown of the power and area estimates based on previously derived models [13] in the low power hardware library.

As will be seen later, the implementation overhead consisting of the control, interconnect and implementation related memory/register take its toll on the power consumption. Reason for this is that the algorithm lacks both spatial and temporal locality. Spatial locality refers to the extent to which an algorithm can itself be partitioned into natural clusters based on connectivity while temporaral locality refers to the average lifetimes of variables.

TABLE 17. Output of SPA Analysis for FFT block

CLASS	AREA (mm^2)	% Area	Cap (pF)	Energy (nJ)	Power (mW)	% Power
DATA_WIRE	57.26	36.3	1016108	1463.2	0.1	12.7
CTRL_WIRE	1.91	1.2	222966	321.1	0.0	2.8
EXU	15.56	9.9	638197	919.0	0.1	8.0
MUX	2.53	1.6	727946	1048.2	0.1	9.1
MEMORY	18.08	11.4	1186234	1708.9	0.2	14.9
CLK_WIRE	0.81	0.5	632371	910.6	0.1	7.9

TABLE 17. Output of SPA Analysis for FFT block

CLASS	AREA (mm^2)	% Area	Cap (pF)	Energy (nJ)	Power (mW)	% Power
CONTROLLER	3.93	2.5	2059134	2965.2	0.3	25.8
REGISTER	4.45	2.8	1279922	1843.1	0.2	16.0
BUFFER	2.19	1.4	219490	316.1	0.0	2.8
OVERHEAD	51.02	32.4	0	0.0	0.0	0.0
TOTAL	157.65	100	7982368	11494.3	1.1	100

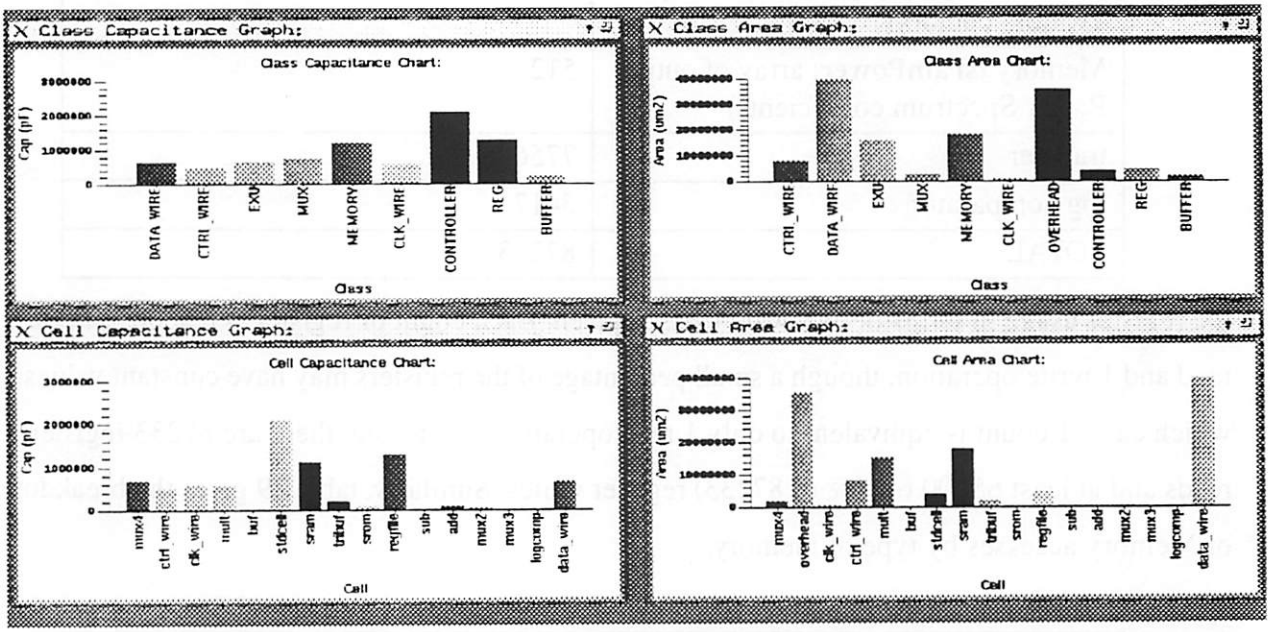


FIGURE 4.4 Graphical Output of SPA Analysis for FFT block

4.3.1 Controllers and Registers

Clearly, the major source of power consumption comes from the controllers. At approximately 26% of the total power consumed, it should be the primary target for power reduction. Recall from Section 3.5 that the controllers are responsible for the functionality of primarily 4 kinds of hardware: registers, memories, multiplexers and tristate buffers.The statistics (Table 9,

and Table 18) obtained earlier using Hyper also support this finding. In Table 18, a breakdown of

TABLE 18. Register Count obtained from Hyper Estimator based on CDFG

Type of EXU	Count
subtract	9856
add	24662
multiply	19014
Memory (**sramIn1**; array of sampled speech inputs)	512
Memory (**sramBF3R**, **sramBF3i**; array of intermediate data between blocks of butterflies)	18432
Memory (**sromWr**, **sromWi**; array of Twiddle Factors)	3072
Memory (**sramPower**; array of output Power Spectrum coefficients)	512
transfer	7756
log comparator	3417
TOTAL	87233

the register usage in terms of hardware class is given. Each count of register is responsible for 1 read and 1 write operation, though a small percentage of the registers may have constant values, in which case. 1 count is equivalent to only 1 read operation. Therefore, there are 87233 register reads and at least 65000 (~75% of 87233) register writes. Similarly, table 19 gives the breakdown of Memory accesses by type of memory.

TABLE 19. Memory Accesses obtained from Hyper Estimator based on CDFG

Memory	Access
sramIn1	512 Reads
sramBF3R	3072 Reads; 3072 Writes
sramBF3i	3072 Reads; 3072 Writes
sromWr	1536 Reads
sromWi	1536 Reads
sramPower	256 Writes
TOTAL	9728 Reads; 6400 Writes

With the total number of read or writes (registers and memories combined) in excess of 168 000, these operations outnumber the EXU execution counts (52184 combined from all the

different EXUs) roughly 3:1. This is explained by the fact that most EXUs obtain their inputs by reading from 2 registers, and write the output onto a 3rd register. In some cases, when the EXU evaluates a common subexpression of several different computations, the output may be written to more then one register, thus further increasing the 3:1 ratio of read/write operations to EXU executions. Furthermore, each register write is usually preceded by a selection of mux inputs, while each EXU execution is usually followed by the enabling of at least one tristate buffer which outputs the result onto a global data bus. Thus, the massive number of control signals required.

Besides using the switching statistics, SPA also uses a Min-Term[1] parameter declared in the ADL description of the controller to determine the size and capacitance of the controllers, since the number of standard cells used to construct the controller depends on the number of min terms. The min-term parameter could be obtained accurately by running the CDL truth table through a minimization tool such as U.C. Berkeley's espresso. However, a faster, more pessimistic estimate is used here by counting the number of rows in the CDL description, and excluding those that cause the controller outputs to idle.

As a demonstration of the effect of the min-term parameter, one of the controllers which magnages a datapath consisting of 1 adder, 2 register files, 6 multiplexers, and 6 output buffers was analyzed using SPA. The switching statistics used in all cases was the same:-

TABLE 20. Effect of MIN-TERM parameter in ADL description

MIN-TERM parameter set to:	SPA Estimates	
	AREA (μm^2)	Capacitance (pF)
30	214239.60	100977.10
40	345268.80	130270.16
50	506106.00	159563.22

The results suggest that with a 10-20% optimization of the CDL truth tables, the control power is likely to decrease by the same percentage.

1. The minimum equivalent boolean representation of the function of the controller. The controller matches its outputs with a CurrentState input through a series of boolean equation.

The control power depends on both the regularity of the algorithm, as well as the number of execution counts. A regular structure such as the consistent looping of computations keeps the number of min-terms down by repeatedly using the same set of controls. This results in a reduction of the number of standard cells used, thus achieving reduced control power as well as complexity of controller structure. At the same time, the number of register read/write controls, mux controls, and buffer controls are observed earlier to have a positive correlation with the number of EXU execution counts. Therefore the control power is expected to decrease with reduced execution counts. However, most algorithms obtain a reduction in execution counts at the expense of increased non-regularity. It is therefore, difficult to lower the control power by simply changing the algorithm. As seen earlier in Section 2.1, the Radix2-Winograd FFT algorithm used was a compromise between control complexity and number of execution counts.

4.3.2 Memories

Besides aiming the low power efforts at the registers, it might also be helpful to target the memories. The intermediate memories sramBF3R and sramBF3i (for real and imaginary intermediate results respectively) were introduced to allow more hardware resource sharing. Recall that the radix-2 part of the FFT algorithm basically consists of 6 cascaded blocks. Each block is further divided into a series of butterflies whose computations could be carried out in parallel, but were implemented in series since the hardware requirement for a parallel computation would be too huge. The series application meant that memories are necessary to store the results of each butterfly computation until the entire block has been processed. This is where the algorithm lacks temporal locality in terms of memories. The results of each iteration of the butterfly is not used until the next block

Memory power consumption is kept low by gating the clock input to the memory. Since the memories are static, and precharges the bit lines at the beginning of each cycle, the load capacitance is relatively data independent. Due to the huge size of memories (e.g. sramBF3i and sramBF3R each consists of 2 huge blocks of 22 bits by 256 words) the precharge is a major power concern. By gating the clock (Figure 4.5) until a read or write is intended, the memory, which is triggered by the positive edge of the clock, is turned off when not in use.

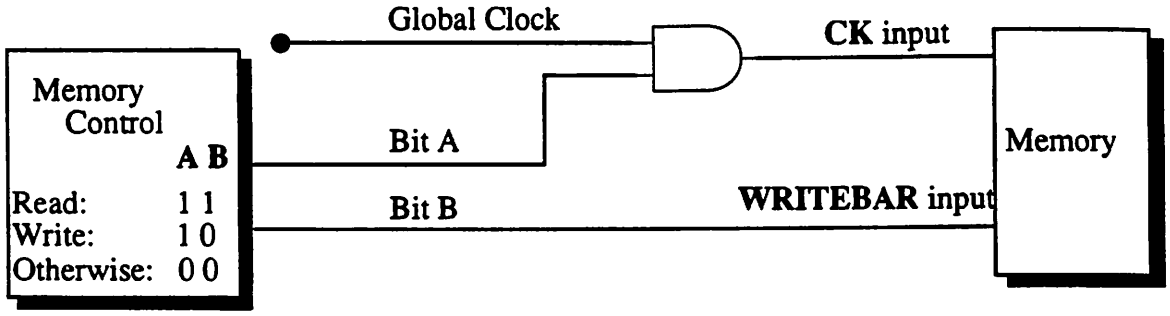


FIGURE 4.5 Gating CLK input to memory

The sram/srom models used in SPA and HYPER were inappropriate as the memories have been overhauled since the models were built. Instead, the power estimates were obtained from IRSIM by simulating the read and write operations with the data obtained from the VHDL simulation done prior to running SPA. As expected, the switch capacitance was relatively data independent. More work is currently being done to model the new srams/sroms used here and incorporate them into the Hyper/Spa estimation model. The values shown in Figure 4.4 and Table 17 show the corrected power figures for the memories.

TABLE 21. Power Breakdown of Memories

Type of Memory	Average Capacitance Switched per Read Cycle	Average Capacitance Switched per Write Cycle
sramBF3r, sramBF3i (22bits x 512 words)	65pF	100pF
sromWr, sromWi (13bits x 256 words)	22pF	N.A
sramIn1 (13bits x 512 words)	50pF	N.A.
sramPower (32bits x 256 words)	N.A.	82pF

4.3.3 Interconnect, Data/Control Buses

Data bus power consists of power dissipated in global data buses which connect different datapath cells, as well as the data wire within each datapath cell. Since the local datapath control cell is described separately from the datapath cell it controls, the control signal buses are also considered as part of the global interconnect. Bus power estimation is based on an estimated wire capacitance, as well as switching activity of the bus accesses. The main factors determining the physical capacitance associated with a wire are its width, its length and the thickness of oxide separating it from the substrate. In the two-level metal process used in this design, the oxide layer thickness is estimated as the average of that found under metal 1 and metal 2 layers, with the assumption that half the routing is expected to be in metal 1, while the rest is in metal 2. Width of data wires/ data buses is the minimum allowed by the 1.2 micron process (3λ width) in order to reduce wiring capacitance. Since the clock frequency is sufficiently low, the RC-delay effect of these minimum-width wires does not affect the results.

The remaining parameter required to determine wire capacitance is the wire length. The top-down wire length estimation process begins with the composite datapath at the top level of hierarchy. Moving down, each datapath cell is invoked with the estimation process which gives an average wire length that will be used for all nets (both data buses and control buses) within that cell. In the actual implementation, wire lengths vary, with shorter, local wires, and longer, more global wires. Datapath cells are placed such that high activity buses are shorter then low activity buses. For instance, the control buses that connect the registers and buffers of an adder datapath cell with the controller cell are switching constantly in order to handle the large number of additions. The controller is naturally placed near the datapath cell in order to take advantage of a shorter wire length. However, SPA interprets any signal transfer between these cells as global bus accesses, on a global wire with the estimated average wire length. Power estimates are therefore pessimistic.

A few comparisons were made between estimated wire length of the data wires and the actual wire lengths in the implementation (Table 22).

TABLE 22. Comparison of data wire area. All data wires are minimum width in order to minimize capacitance, and since clock frequency is sufficiently low to allow slower data transitions.

Description of Bus	SPA Estimate (μm^2)	Implemented (μm^2)
GLOBAL BUSES		
Adder to Subtractor	4917	13810
Multiplier to Adder	4917	5000
Memory to memory buffers	4917	1309
Datapath Cell: dp2		
Mux to Register	361	2133
Adder sum to output buffer	361	372
Mux to Mux	361	215

Since there are fewer degrees of freedom compared to a 2-dimensional placement/routing, the wire length estimates were generally more accurate for datapath cells where one dimensional placement strategies have been used.

The lack of spatial locality in the algorithm causes a high bus power consumption. The pseudo code in Figure 3.2 shows that the EXU operations such as multiply, add and subtract, as well as memory accesses are intertwined with one another. Temporary results or data are transferred from the output of a EXU to the input registers of another EXU through these global buses, thereby causing a high amount of bus activity.

4.3.4 EXU

The main source of EXU power consumption is the multipliers. Even so, the total power consumed by all EXUs is only about 8% of the total power consumption (see Table 17).Therefore, the overall gain of any improvement done to the EXU power is likely to be negligible (5% improvement versus a power accuracy of about 20%?). Nevertheless, 2 suggestions will be made to improve the multiplier power.

Note that in the algorithm repeated below, the result of *(64*j3)* is used repeatedly for

```
                (j3 : 0 .. 7)::
                  begin
                   (k3 : 0 .. 31)::
                   begin
(i)                 R5high = BF5R[64 * j3 + k3];
(ii)                R5low = BF5R[64 * j3 + k3 + 32];
(iii)               I5high = BF5i[64 * j3 + k3];
(iv)                I5low = BF5i[64 * j3 + k3 + 32];
(v)                 v = Wr[8 * k3];
(vi)                w = Wi[8 * k3];
(vii)               temp6r = word(v * R5low) - word(w * I5low);
(viii)              temp6i = word(w * R5low) + word(v * I5low);
(ix)                BF6R[64 * j3 + k3] = R5high + temp6r;
(x)                 BF6i[64 * j3 + k3] = I5high + temp6i;
(xi)                BF6R[64 * j3 + k3 + 32] = R5high - temp6r;
(xii)               BF6i[64 * j3 + k3 + 32] = I5high - temp6i;
                  end;
```

address generation of every memory access. It is also repeated over each of the 32 iterations where *k3* increases from 0 to 31, before the value *j3* is updated. Although common-subexpression elimination prevents the computation of the product repeatedly for the addresses, each new iteration of the inner loop still computes the product regardless of whether the value of j3 has changed. The reason lies with the representation of nested loops in the Hyper CDFG. The Silage parser in Hyper represents the array subscripts as coefficients of a basis spanned by the variables in the loop. For the nested loops just presented, the variables would be *j3* and *k3*, and a typical set of coefficients would be (64, 1, 32) which represents *(64*j3 + 1*k3 + 32)*. The parser does not allow an array address index to be described as a data edge. It is therefore *not* possible to describe the algorithm as below, which would have forced the computation of the product outside the innermost loop.

```
(j3:0..70
begin
      temp = 64*j3;              /* temp becomes a data edge */
      (k3:0..32)::
      begin
         R5high = BF5R[temp + k3];    /* Using temp as an array index */
            .
              .
      .
      .
```

Instead, the parser produces a CDFG which 'sees' the *64*j3* product only in the innermost loop. Memory management, which follows the parser, then takes care of address generation by creating nodes to compute the product, but only in the innermost loop. An obvious improvement would

have been to move the multiply node out of the innermost loop, thus saving 31 out of every 32 multiplications.

5.0 Conclusion

This work describes the design path to realize a memory-intensive, low-power speech recognizer implementation. A high level analysis of the entire RASTA-PLP front end of a speech recognition chip has been done. In particular, the FFT block and the Critical Band Filtering blocks were each mapped onto a RTL description, and a layout has been generated for each of these two blocks. An architectural power analysis was also performed on the FFT block using SPA.

The FFT block, which actually computes a periodogram estimate of the power spectrum of the speech input, from a 512-point FFT, has an estimated core power of 1.1mW with a 1.2V supply voltage. Much of this power comes from the generation of the massive number of control signals, a consequence of the register-intensive architecture. Despite this, the registers allow the designer to avoid detailed timing analysis, and allow automatic scheduling which basically discretizes the execution of all operations.

The most beneficial improvement to the Hyper framework may be the C++ parser that is currently being developed. The Split-Radix FFT algorithm is certainly better then the Radix2-Winograd which was implemented, in terms of number of execution counts. It would be interesting to see a way to describe the algorithm in a more structured manner, given the procedural style of C++ that also doesn't require manifest loops, or manifest array indices. Other suggested improvements include a scheduler that recognizes and utilizes regularity of algorithms, and a possible automated translator that handles the SDL/Bdsyn to ADL/CDL process.

6.0 References

[1]. J. Rabaey, et. al. "Fast Prototyping of Datapath-Intensive Architecture". IEEE Design and Test of Computer, pp40-51, 1991

[2]. L. Rabiner and B. Juang, "Signal Processing and Analysis Methods for Speech Recognition," Prentice Hall, 1993

[3]. P. M. Clarkson, "Durbin's Algorithm," Optimal and Adaptive Signal Processing, CRC Press, Chap 3, Appendix C, pp 136-140

[4]. P. M. Clarkson, "Periodogram Methods of Spectral Estimation," Optimal and Adaptive Signal Processing, CRC Press, Chap 7.2, pp 407-418

[5]. H. Hermansky, "Perceptual Linear Predictive (PLP) analysis of speech," J. Acoust. Soc. Am., pp1738-1752, 1990

[6]. H. Hermansky, N. Morgan, A. Bayya and P. Kohn, "Compensation for the Effect of the Communication Channel in Auditory-Like Analysis of Speech (RASTA-PLP)," Proc. Eurospeech, Genova, Italy, pp. 1367-1371, December 1991

[7]. P. Hilfinger and J. Rabaey, "DSP Specification Using the Silage Language." Anatomy of a Silicon Compiler, Chap. 15, pp 200-220

[8]. C.S. Shung, et al., "An Integrated CAD System for Algorithm-Specific IC Design," International Conference on System Design, Hawaii, January 1989.

[9]. M.R. Schroeder, "Recognition of Complex Acoustic Signals,",Life Sciences Research Report 5, edited by T.Y. Bullock, pp 324, 1977

[10]. E. Zwicker, "Masking and psychological excitation as consequences of ear's frequency analysis." Frequency Analysis and Periodicity Detection in Hearing, edited by R.Plomp and G.F. Smoorenburg, 1970.

[11]. D. Elliott, "Handbook of Digital Signal Processing Engineering Applications," p. 597

[12]. H. V. Sorensen, et al. "On Computing the Split-Radix FFT", IEEE Transactions on Acoustics, Speech and Signal Processin, Vol ASSP-34, No. 1, pp 152-156, Feb 1986

[13]. P. Landman, "Low-Power Architectural Design Methodologies," Ph.D. dissertation, UC Berkeley, August 1994

[14]. R. Mehra, et al. "Algorithm and Architectural Level Methodologies for Low Power", Low Power Design Methodologies, Chap 11, pp 335-362

[15]. L. Guerra and J. Rabaey, "System Level Design Guidance using Algorithm Properties", Proc. of IEEE Workshop on VLSI Signal Processing, San Diego, pp 73-82, October 1994.

[16]. L. Jamieson, "Charaterzing Parallel Algorithms," The Characteristics of Parallel Algorithms, L Jamieson, D. Gannon, R. Douglass (editors), MIT Press, Cambridge, Mass., 1987

[17]. B. P. Bogert, M.J.R. Healy, and J. W. Tukey, "The Quefrency Alanysis of Time Series for Echoes" Cepstrum Pseudo-autocovariance, Cross-Cepstrum, and Saphe Cracking," Proc. Symposium Time Series Analysis, M. Rosenblatt, Ed., John Wiley and Sons, New York, pp209-243, 1963

[18]. S.B. Davis and P. Mermelstein, "Comparison of Parametric Representations for Monosyllabic Word Recognition in Continuously Spoken Sentence," IEEE Trans. on Acoustics, Speech and Signal Processing, Vol ASSP-28, No. 4, pp. 357-366, August 1980.

[19]. D.W. Robinson and R.S. Dadson, "A redetermination of the equal-loudness relations for pure tones," Br. J. Appl. Phys. 7, pp 166-181, 1956

[20]. J. Makhoul and L. Cosell. "LPCW: An LPC vocoder with linear predictive spectral warping," Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing 76, pp466-469, Philadelphia.

[21]. S.S. Stevens, "On the psychophysical law," Psychol. Rev. 64, pp153-181.

```
/* Radix 2 algorith for 512-point FFT , 2^9 */
#define wordin fix<13,12>
#define word fix<22,12>
#define wordout fix<32,17>

func main(In1:wordin[512]; Wr, Wi:word[256])
  Power:wordout[] =
begin
  (j:0..255)::
  begin
        a = In1[(2*j)];
        b = In1[(2*j)+1];
        BFr1[2*j] = a + b;
        BFr1[(2*j)+1] = a - b;
  end;


  (j:0..127)::
  begin
   (k:0..1)::
   begin
        R1low = BFr1[(4*j)+2+k];
        R1high = BFr1[(4*j)+k];
        temp2r = word(Wr[128*k] * R1low);
        temp2i = word( Wi[128*k] * R1low);
        BF2R[(4*j)+k] = R1high + temp2r;
        BF2i[(4*j)+k] = temp2i;
        BF2R[(4*j)+k+2] = R1high - temp2r;
        BF2i[(4*j)+k+2] = - temp2i;
   end;
  end;


  (j:0..63)::
  begin
    (k:0..3)::
    begin

        c = Wr[64*k];
        d = Wi[64*k];
        R2high = BF2R[(8*j)+k];
        R2low = BF2R[(8*j)+k+4];
        I2high = BF2i[(8*j)+k];
        I2low = BF2i[(8*j)+k+4];
        temp3r =  word(c * R2low) - word(d * I2low);
        temp3i =  word(d * R2low) + word(c * I2low);
        BF3R[(8*j)+k] = R2high + temp3r;
```

```
        BF3i[(8*j)+k] = I2high + temp3i;
        BF3R[(8*j)+4+k] = R2high - temp3r;
        BF3i[(8*j)+4+k] = I2high - temp3i;
  end;
end;


(j:0..31)::
begin
  (k:0..7)::
  begin
        e = Wr[32*k];
        f = Wi[32*k];
        R3high = BF3R[(16*j)+k];
        R3low = BF3R[(16*j)+8+k];
        I3high = BF3i[(16*j)+k];
        I3low = BF3i[(16*j)+8+k];
        temp4r = word(e * R3low) - word(f * I3low);
        temp4i = word(f * R3low) + word(e * I3low);
        BF4R[(16*j)+k]  = R3high + temp4r;
        BF4i[(16*j)+k]  = I3high + temp4i;
        BF4R[(16*j)+k+8] = R3high - temp4r;
        BF4i[(16*j)+k+8] = I3high - temp4i;


   end;
 end;


(j : 0 .. 15)::
 begin
  (k : 0 .. 15)::
   begin
    R4low = BF4R[32 * j + k + 16];
    R4high = BF4R[32 * j + k];
    I4low = BF4i[32 * j + k + 16];
    I4high = BF4i[32 * j + k];
    g = Wr[16*k];
    h = Wi[16*k];
    temp5r = word(g * R4low) - word(h * I4low);
    temp5i = word(h * R4low) + word(g * I4low);
    BF5R[32 * j + k] = R4high + temp5r;
    BF5i[32 * j + k] = I4high + temp5i;
    BF5R[32 * j + k + 16] = R4high - temp5r;
    BF5i[32 * j + k + 16] = I4high - temp5i;

   end;
```

```
end;
(j : 0 .. 7)::
 begin
  (k : 0 .. 31)::
   begin
    R5high = BF5R[64 * j + k];
    R5low = BF5R[64 * j + k + 32];
    I5high = BF5i[64 * j + k];
    I5low = BF5i[64 * j + k + 32];
    v = Wr[8 * k];
    w = Wi[8 * k];
    temp6r = word(v * R5low) - word(w * I5low);
    temp6i = word(w * R5low) + word(v * I5low);
    BF6R[64 * j + k] = R5high + temp6r;
    BF6i[64 * j + k] = I5high + temp6i;
    BF6R[64 * j + k + 32] = R5high - temp6r;
    BF6i[64 * j + k + 32] = I5high - temp6i;
   end;
end;
(j : 0 .. 3)::
 begin
  (k : 0 .. 63)::
   begin
    R6high = BF6R[128 * j + k];
    I6high = BF6i[128 * j + k];
    R6low = BF6R[128 * j + k + 64];
    I6low = BF6i[128 * j + k + 64];
    r = Wr[4 * k];
    p = Wi[4 * k];
    temp7r = word(r * R6low) - word(p * I6low);
    temp7i = word(p * R6low) + word(r * I6low);
    BF7R[128 * j + k] = R6high + temp7r;
    BF7i[128 * j + k] = I6high + temp7i;
    BF7R[128 * j + k + 64] = R6high - temp7r;
    BF7i[128 * j + k + 64] = I6high - temp7i;
   end;
end;
(j : 0 .. 1)::
 begin
  (k : 0 .. 127)::
   begin
    R7high = BF7R[256 * j + k];
    I7high = BF7i[256 * j + k];
    R7low = BF7R[256 * j + k + 128];
    I7low = BF7i[256 * j + k + 128];
    s = Wr[2*k];
    t = Wi[2*k];
```

```
    temp8r = word(s * R7low) - word(t * I7low);
    temp8i = word(t * R7low) + word(s * I7low);
    BF8R[256 * j + k] = R7high + temp8r;
    BF8i[256 * j + k] = I7high + temp8i;
    BF8R[256 * j + k + 128] = R7high - temp8r;
    BF8i[256 * j + k + 128] = I7high - temp8i;
  end;
 end;
 (k : 0 .. 255)::
  begin
   R8low = BF8R[k + 256];
   I8low = BF8i[k + 256];
   x = Wr[k];
   y = Wi[k];
   temp9r = word(x * R8low) - word(y * I8low);
   temp9i = word(y * R8low) + word(x * I8low);
   BF9R[k] = BF8R[k] + temp9r;
   BF9i[k] = BF8i[k] + temp9i;
   Power[k] = wordout(BF9R[k] * BF9R[k]) + wordout(BF9i[k] * BF9i[k]);
  end;
end;
```

```
#define wordin fix<13,11>
#define word fix<22,12>
#define wordout fix<32,17>
#define cosu word(0.707106781)
#define sinu (0.707106781)

func main(In1: wordin[512]; Wr, Wi: wordin[256]) Power :wordout[] =
begin
 (i:0..7)::
 begin
   (j:0..7)::
   begin
        (m0, m1, m2, m5, s1, s2, s3, s4) =
          fft8(In1[64*i+8*j], In1[64*i+8*j+1], In1[64*i+8*j+2],
                In1[64*i+8*j+3], In1[64*i+8*j+4], In1[64*i+8*j+5],
                In1[64*i+8*j+6], In1[64*i+8*j+7]);

        (BF3R[64*i+8*j],BF3R[64*i+8*j+1], BF3R[64*i+8*j+2],
         BF3R[64*i+8*j+3], BF3R[64*i+8*j+4], BF3R[64*i+8*j+5],
         BF3R[64*i+8*j+6], BF3R[64*i+8*j+7]) =
                                              (m0, s1, m2, s2, m1, s2, m2, s1);

        (BF3i[64*i+8*j],BF3i[64*i+8*j+1], BF3i[64*i+8*j+2],
         BF3i[64*i+8*j+3], BF3i[64*i+8*j+4], BF3i[64*i+8*j+5],
         BF3i[64*i+8*j+6], BF3i[64*i+8*j+7]) = (0, s4, m5, -s3, 0, s3, -m5, -s4);

   end;
 end;


  (j1:0..31)::
  begin
    (k1:0..7)::
    begin
         e = Wr[32*k1];
         f = Wi[32*k1];
         R3high = BF3R[(16*j1)+k1];
         R3low = BF3R[(16*j1)+8+k1];
         I3high = BF3i[(16*j1)+k1];
         I3low = BF3i[(16*j1)+8+k1];
         temp4r = word(e * R3low) - word(f * I3low);
         temp4i = word(f * R3low) + word(e * I3low);
         BF4R[(16*j1)+k1]  = R3high + temp4r;
         BF4i[(16*j1)+k1]  = I3high + temp4i;
         BF4R[(16*j1)+k1+8] = R3high - temp4r;
         BF4i[(16*j1)+k1+8] = I3high - temp4i;
```

```
  end;
 end;


(j2 : 0 .. 15)::
 begin
  (k2 : 0 .. 15)::
   begin
    R4low = BF4R[32 * j2 + k2 + 16];
    R4high = BF4R[32 * j2 + k2];
    I4low = BF4i[32 * j2 + k2 + 16];
    I4high = BF4i[32 * j2 + k2];
    g = Wr[16*k2];
    h = Wi[16*k2];
    temp5r = word(g * R4low) - word(h * I4low);
    temp5i = word(h * R4low) + word(g * I4low);
    BF5R[32 * j2 + k2] = R4high + temp5r;
    BF5i[32 * j2 + k2] = I4high + temp5i;
    BF5R[32 * j2 + k2 + 16] = R4high - temp5r;
    BF5i[32 * j2 + k2 + 16] = I4high - temp5i;

   end;
 end;
(j3 : 0 .. 7)::
 begin
  (k3 : 0 .. 31)::
   begin
    R5high = BF5R[64 * j3 + k3];
    R5low = BF5R[64 * j3 + k3 + 32];
    I5high = BF5i[64 * j3 + k3];
    I5low = BF5i[64 * j3 + k3 + 32];
    v = Wr[8 * k3];
    w = Wi[8 * k3];
    temp6r = word(v * R5low) - word(w * I5low);
    temp6i = word(w * R5low) + word(v * I5low);
    BF6R[64 * j3 + k3] = R5high + temp6r;
    BF6i[64 * j3 + k3] = I5high + temp6i;
    BF6R[64 * j3 + k3 + 32] = R5high - temp6r;
    BF6i[64 * j3 + k3 + 32] = I5high - temp6i;
   end;
 end;
 (j4 : 0 .. 3)::
  begin
   (k4 : 0 .. 63)::
    begin
     R6high = BF6R[128 * j4 + k4];
```

```
    I6high = BF6i[128 * j4 + k4];
    R6low = BF6R[128 * j4 + k4 + 64];
    I6low = BF6i[128 * j4 + k4 + 64];
    r = Wr[4 * k4];
    p = Wi[4 * k4];
    temp7r = word(r * R6low) - word(p * I6low);
    temp7i = word(p * R6low) + word(r * I6low);
    BF7R[128 * j4 + k4] = R6high + temp7r;
    BF7i[128 * j4 + k4] = I6high + temp7i;
    BF7R[128 * j4 + k4 + 64] = R6high - temp7r;
    BF7i[128 * j4 + k4 + 64] = I6high - temp7i;
   end;
  end;
  (j5 : 0 .. 1)::
  begin
   (k5 : 0 .. 127)::
   begin
    R7high = BF7R[256 * j5 + k5];
    I7high = BF7i[256 * j5 + k5];
    R7low = BF7R[256 * j5 + k5 + 128];
    I7low = BF7i[256 * j5 + k5 + 128];
    s = Wr[2*k5];
    t = Wi[2*k5];
    temp8r = word(s * R7low) - word(t * I7low);
    temp8i = word(t * R7low) + word(s * I7low);
    BF8R[256 * j5 + k5] = R7high + temp8r;
    BF8i[256 * j5 + k5] = I7high + temp8i;
    BF8R[256 * j5 + k5 + 128] = R7high - temp8r;
    BF8i[256 * j5 + k5 + 128] = I7high - temp8i;
   end;
  end;
  (k6 : 0 .. 255)::
  begin
   R8low = BF8R[k6 + 256];
   I8low = BF8i[k6 + 256];
   x = Wr[k6];
   y = Wi[k6];
   temp9r = word(x * R8low) - word(y * I8low);
   temp9i = word(y * R8low) + word(x * I8low);
   BF9R[k6] = BF8R[k6] + temp9r;
   BF9i[k6] = BF8i[k6] + temp9i;
   Power[k6] = wordout(BF9R[k6] * BF9R[k6]) + wordout(BF9i[k6] *
BF9i[k6]);
  end;
end;
```

```
func fft8 (x0,x1,x2,x3,x4,x5,x6,x7: wordin) m0,m1,m2,m5,s1,s2,s3,s4 : word =
begin

/*
 Source - Elliott, D., "Handbook of Digital Signal Processing
          Engineering Applications," p. 597.
 Description - S. Winograd small-N DFT for N = 8, u = 2/8*pi.
 Output -
  X(0) = m0;
  X(1) = s1 + j*s4;   Imaginary indices of X(1), X(2), X(5), X(6)
                      checked, and adjusted. -Engling 10/7/94

  X(2) = m2 + j*m5;
  X(3) = s2 - j*s3;
  X(4) = m1;
  X(5) = s2 + j*s3;
  X(6) = m2 - j*m5;
  X(7) = s1 - j*s4;
*/

  t1 = x0 + x4;
  t2 = x2 + x6;
  t3 = x1 + x5;
  t4 = x1 - x5;
  t5 = x3 + x7;
  t6 = x3 - x7;
  t7 = t1 + t2;
  t8 = t3 + t5;

  m0 = word(t7 + t8);
  m1 = word(t7 - t8);
  m2 = word(t1 - t2);
  m3 = word(x0 - x4);
  m4 = word(cosu * (t4 - t6));
  m5 = word(t5 - t3);
  m6 = word(x6 - x2);
  m7 = word(wordin(1.0 * sinu) * (t4 + t6));

  s1 = word(m3 + m4);
  s2 = word(m3 - m4);
  s3 = word(m6 + m7);
  s4 = word(m6 - m7);

end;
```

```
#define twopi 6.283185307179586
#define wordin  fix<13,12>
#define word  fix<22,12>
#define wordout fix<45,24>

func main(x: word[512]; Wc, Ws: word[384]) Power: wordout[] =
begin
                /* Length two transforms */
  (J:0..0)::
  begin
        (k4:0..0)::
        begin
                r1five = x[510];
                s1five = x[511];
                x1[510] = r1five + s1five;
                x1[511] = r1five - s1five;
        end:
        (k3:0..1)::
        begin
                        r1four = x[126+256*k3];
                        s1four = x[127+256*k3];
                        x1[126 + 256*k3] = r1four + s1four;
                        x1[127 + 256*k3] = r1four - s1four;
        end;
        (k2:0..7)::
        begin
                        r1three = x[30+64*k2];
                        s1three = x[31+64*k2];
                        x1[30+64*k2] = r1three + s1three;
                        x1[31+64*k2] = r1three - s1three;
        end;
        (k1:0..31)::
        begin
                        r1two = x[6+16*k1];
                        s1two = x[7+16*k1];
                        x1[6+16*k1] = r1two + s1two;
                        x1[7+16*k1] = r1two - s1two;
        end;
        (k:0..127)::
        begin
                        r1one = x[4*k];
                        s1one= x[4*k+1];
                        x1[4*k] = r1one + s1one;
                        x1[4*k+1] = r1one - s1one;
        end:
```

```
end;

        /* L Shaped Butterflies */



        /* k = 2 */
(J:0..0)::
begin
        (io:0..0)::
        begin
                ya24 = 0;
                yb24 = 0;
                yc24 = 0;
                yd24 = 0;
                xa24 = x1[254];
                xb24 = x1[255];
                xc24 = x1[252];
                xd24 = x1[253];

                R12four = xa24;
                S12four = ya24;
                R22four = xb24;
                S22four = yb24;
                r32four = R12four + R22four;
                r22four = R12four - R22four;
                r12four = S12four + S22four;
                s22four = S12four - S22four;

                x2[254] = xc24 - r32four;
                x2[252] = xc24 + r32four;
                x2[255] = xd24 - s22four;
                x2[253] = xd24 + s22four;
                y2[254] = yc24 - r12four;
                y2[252] = yc24 + r12four;
                y2[255] = yd24 + r22four;
                y2[253] = yd24 - r22four;
        end;
        (io:0..3)::
        begin
                ya23 = 0;
                yb23 = 0;
                yc23 = 0;
                yd23 = 0;
                xa23 = x1[128*io+62];
                xb23 = x1[128*io+63];
```

```
        xc23 = x1[128*io+60];
        xd23 = x1[128*io+61];

        R12three = xa23;
        S12three = ya23;
        R22three = xb23;
        S22three = yb23;
        r32three = R12three + R22three;
        r22three = R12three - R22three;
        r12three = S12three + S22three;
        s22three = S12three - S22three;

        x2[128*io+62] = xc23 - r32three;
        x2[128*io+60] = xc23 + r32three;
        x2[128*io+63] = xd23 - s22three;
        x2[128*io+61] = xd23 + s22three;
        y2[128*io+62] = yc23 - r12three;
        y2[128*io+60] = yc23 + r12three;
        y2[128*io+63] = yd23 + r22three;
        y2[128*io+61] = yd23 - r22three;
end;

(io:0..15)::
begin
        ya22 = 0;
        yb22 = 0;
        yc22 = 0;
        yd22 = 0;
        xa22 = x1[32*io+14];
        xb22 = x1[32*io+15];
        xc22 = x1[32*io+12];
        xd22 = x1[32*io+13];

        R12two = xa22;
        S12two = ya22;
        R22two = xb22;
        S22two = yb22;
        r32two = R12two + R22two;
        r22two = R12two - R22two;
        r12two = S12two + S22two;
        s22two = S12two - S22two;

        x2[32*io+14] = xc22 - r32two;
        x2[32*io+12] = xc22 + r32two;
        x2[32*io+15] = xd22 - s22two;
        x2[32*io+13] = xd22 + s22two;
        y2[32*io+14] = yc22 - r12two;
```

```
                y2[32*io+12] = yc22 + r12two;
                y2[32*io+15] = yd22 + r22two;
                y2[32*io+13] = yd22 - r22two;


        end;

        (io:0..63)::
        begin
                ya21 = 0;
                yb21 = 0;
                yc21 = 0;
                yd21 = 0;
                xa21 = x1[8*io+2];
                xb21 = x1[8*io+3];
                xc21 = x1[8*io];
                xd21 = x1[8*io+1];

                R12one = xa21;
                S12one = ya21;
                R22one = xb21;
                S22one = yb21;
                r32one = R12one + R22one;
                r22one = R12one - R22one;
                r12one = S12one + S22one;
                s22one = S12one - S22one;

                x2[8*io+2] = xc21 - r32one;
                x2[8*io] = xc21 + r32one;
                x2[8*io+3] = xd21 - s22one;
                x2[8*io+1] = xd21 + s22one;
                y2[8*io+2] = yc21 - r12one;
                y2[8*io] = yc21  + r12one;
                y2[8*io+3] = yd21 + r22one;
                y2[8*io+1] = yd21 - r22one;
        end;
end;
 /* k = 3 */
 (j:0..1)::
 begin
        (io:0..0)::
        begin
                ya34 = y2[508+j];
                yb34 = y2[510+j];
                yc34 = y2[504+j];
                yd34 = y2[506+j];
                xa34 = x2[508+j];
```

```
        xb34 = x2[510+j];
        xc34 = x2[504+j];
        xd34 = x2[506+j];

        R13four = word(xa34*Wc[64*j]) + word(ya34*Ws[64*j]);
        S13four = word(ya34*Wc[64*j]) - word(xa34*Ws[64*j]);
        R23four = word(xb34*Wc[192*j]) + word(yb34*Ws[192*j]);
        S23four = word(yb34*Wc[192*j]) - word(xb34*Ws[192*j]);
        r33four = R13four + R23four;
        r23four = R13four - R23four;
        r13four = S13four + S23four;
        s23four = S13four - S23four;

        x3[508+j] = xc34 - r33four;
        x3[504+j] = xc34 + r33four;
        x3[510+j] = xd34 - s23four;
        x3[506+j] = xd34 + s23four;
        y3[508+j] = yc34 - r13four;
        y3[504+j] = yc34 + r13four;
        y3[510+j] = yd34 + r23four;
        y3[506+j] = yd34 - r23four;
end;
(io:0..1)::
begin
        ya33 = y2[256*io+124+j];
        yb33 = y2[256*io+126+j];
        yc33 = y2[256*io+120+j];
        yd33 = y2[256*io+122+j];
        xa33 = x2[256*io+124+j];
        xb33 = x2[256*io+126+j];
        xc33 = x2[256*io+120+j];
        xd33 = x2[256*io+122+j];

        R13three = word(xa33*Wc[64*j]) + word(ya33*Ws[64*j]);
        S13three = word(ya33*Wc[64*j]) - word(xa33*Ws[64*j]);
        R23three = word(xb33*Wc[192*j]) + word(yb33*Ws[192*j]);
        S23three = word(yb33*Wc[192*j]) - word(xb33*Ws[192*j]);
        r33three = R13three + R23three;
        r23three = R13three - R23three;
        r13three = S13three + S23three;
        s23three = S13three - S23three;

        x3[256*io+124+j] = xc33 - r33three;
        x3[256*io+120+j] = xc33 + r33three;
        x3[256*io+126+j] = xd33 - s23three;
        x3[256*io+122+j] = xd33 + s23three;
        y3[256*io+124+j] = yc33 - r13three;
```

```
        y3[256*io+120+j] = yc33 + r13three;
        y3[256*io+126+j] = yd33 + r23three;
        y3[256*io+122+j] = yd33 - r23three;
end;
(io:0..7)::
begin
        ya32 = y2[64*io+28+j];
        yb32 = y2[64*io+30+j];
        yc32 = y2[64*io+24+j];
        yd32 = y2[64*io+26+j];
        xa32 = x2[64*io+28+j];
        xb32 = x2[64*io+30+j];
        xc32 = x2[64*io+24+j];
        xd32 = x2[64*io+26+j];

        R13two = word(xa32*Wc[64*j]) + word(ya32*Ws[64*j]);
        S13two = word(ya32*Wc[64*j]) - word(xa32*Ws[64*j]);
        R23two = word(xb32*Wc[192*j]) + word(yb32*Ws[192*j]);
        S23two = word(yb32*Wc[192*j]) - word(xb32*Ws[192*j]);
        r33two = R13two + R23two;
        r23two = R13two - R23two;
        r13two = S13two + S23two;
        s23two = S13two - S23two;

        x3[64*io+28+j] = xc32 - r33two;
        x3[64*io+24+j] = xc32 + r33two;
        x3[64*io+30+j] = xd32 - s23two;
        x3[64*io+26+j] = xd32 + s23two;
        y3[64*io+28+j] = yc32 - r13two;
        y3[64*io+24+j] = yc32 + r13two;
        y3[64*io+30+j] = yd32 + r23two;
        y3[64*io+26+j] = yd32 - r23two;


end;
(io:0..31)::
begin
        ya31 = y2[16*io+4+j];
        yb31 = y2[16*io+6+j];
        yc31 = y2[16*io+j];
        yd31 = y2[16*io+2+j] ;
        xa31 = x2[16*io+4+j];
        xb31 = x2[16*io+6+j];
        xc31 = x2[16*io+j];
        xd31 = x2[16*io+2+j];

        R13one = word(xa31*Wc[64*j]) + word(ya31*Ws[64*j]);
```

```
                S13one = word(ya31*Wc[64*j]) - word(xa31*Ws[64*j]);
                R23one = word(xb31*Wc[192*j]) + word(yb31*Ws[192*j]);
                S23one = word(yb31*Wc[192*j]) - word(xb31*Ws[192*j]);
                r33one = R13one + R23one;
                r23one = R13one - R23one;
                r13one = S13one + S23one;
                s23one = S13one - S23one;

                x3[16*io+4+j] = xc31 - r33one;
                x3[16*io+j] = xc31 +  r33one;
                x3[16*io+6+j] = xd31 - s23one;
                x3[16*io+2+j] = xd31 + s23one;
                y3[16*io+4+j] = yc31 - r13one;
                y3[16*io+j] = yc31 + r13one;
                y3[16*io+6+j] = yd31 + r23one;
                y3[16*io+2+j] = yd31 - r23one;
        end;


end;


/* k = 4 */
(j:0..3)::
begin
        (io:0..0)::
        begin
                ya43 = y3[240+2*4+j];
                yb43 = y3[240+3*4+j];
                yc43 = y3[240+j]:
                yd43 = y3[240+4+j] ;
                xa43 = x3[240+2*4+j];
                xb43 = x3[240+3*4+j];
                xc43 = x3[240+j];
                xd43 = x3[240+4+j];

                R14three = word(xa43*Wc[32*j]) + word(ya43*Ws[32*j]);
                S14three = word(ya43*Wc[32*j]) - word(xa43*Ws[32*j]);
                R24three = word(xb43*Wc[96*j]) + word(yb43*Ws[96*j]);
                S24three = word(yb43*Wc[96*j]) - word(xb43*Ws[96*j]);
                r34three = R14three + R24three;
                r24three = R14three - R24three;
                r14three = S14three + S24three;
                s24three = S14three - S24three;

                x4[240+j+2*4] = xc43 - r34three;
                x4[240+j] = xc43 +  r34three;
```

```
        x4[240+j+3*4] = xd43 - s24three;
        x4[240+j+4] = xd43 + s24three;
        y4[240+j+2*4] = yc43 - r14three;
        y4[240+j] = yc43 + r14three;
        y4[240+j+3*4] = yd43 + r24three;
        y4[240+j+4] = yd43 - r24three;
end;
(io:0..3)::
begin
        ya42 = y3[128*io+48+2*4+j];
        yb42 = y3[128*io+48+3*4+j];
        yc42 = y3[128*io+48+j];
        yd42 = y3[128*io+48+4+j] ;
        xa42 = x3[128*io+48+2*4+j];
        xb42 = x3[128*io+48+3*4+j];
        xc42 = x3[128*io+48+j];
        xd42 = x3[128*io+48+4+j];

        R14two = word(xa42*Wc[32*j]) + word(ya42*Ws[32*j]);
        S14two = word(ya42*Wc[32*j]) - word(xa42*Ws[32*j]);
        R24two = word(xb42*Wc[96*j]) + word(yb42*Ws[96*j]);
        S24two = word(yb42*Wc[96*j]) - word(xb42*Ws[96*j]);
        r34two = R14two + R24two;
        r24two = R14two - R24two;
        r14two = S14two + S24two;
        s24two = S14two - S24two;

        x4[128*io+48+2*4+j] = xc42 - r34two;
        x4[128*io+48+j] = xc42 + r34two;
        x4[128*io+48+3*4+j] = xd42 - r24two;
        x4[128*io+48+4+j] = xd42 + r24two;
        y4[128*io+48+2*4+j] = yc42 - r14two;
        y4[128*io+48+j] = yc42 +  r14two;
        y4[128*io+48+3*4+j] = yd42 + r24two;
        y4[128*io+48+4+j] = yd42 - r24two;


end;
(io:0..15)::
begin
        ya41 = y3[32*io + 2*4+j];
        yb41 = y3[32*io + 3*4+j];
        yc41 = y3[32*io+j];
        yd41 = y3[32*io+4+j] ;
        xa41 = x3[32*io+2*4+j];
        xb41 = x3[32*io+3*4+j];
        xc41 = x3[32*io+j];
```

```
                xd41 = x3[32*io+4+j];

                R14one = word(xa41*Wc[32*j]) + word(ya41*Ws[32*j]);
                S14one = word(ya41*Wc[32*j]) - word(xa41*Ws[32*j]);
                R24one = word(xb41*Wc[96*j]) + word(yb41*Ws[96*j]);
                S24one = word(yb41*Wc[96*j]) - word(xb41*Ws[96*j]);
                r34one = R14one + R24one;
                r24one = R14one - R24one;
                r14one = S14one + S24one;
                s24one = S14one - S24one;

                x4[32*io+2*4+j] = xc41 - r34one;
                x4[32*io+j] = xc41 + r34one;
                x4[32*io+3*4+j] = xd41 - s24one;
                x4[32*io+4+j] = xd41 + s24one;
                y4[32*io+2*4+j] = yc41 - r14one;
                y4[32*io+j] = yc41 + r14one;
                y4[32*io+3*4+j] = yd41 + r24one;
                y4[32*io+4+j] = yd41 - r24one;
        end;

end;

/* k = 5 */
(j:0..7)::
begin
        (io:0..0)::
        begin
                ya53 = y4[480 + 2*8+j];
                yb53 = y4[480 + 3*8+j];
                yc53 = y4[480+j];
                yd53 = y4[480+8+j] ;
                xa53 = x4[480+2*8+j];
                xb53 = x4[480+3*8+j];
                xc53 = x4[480+j];
                xd53 = x4[480+8+j];

                R15three = word(xa53*Wc[16*j]) + word(ya53*Ws[16*j]);
                S15three = word(ya53*Wc[16*j]) - word(xa53*Ws[16*j]);
                R25three = word(xb53*Wc[48*j]) + word(yb53*Ws[48*j]);
                S25three = word(yb53*Wc[48*j]) - word(xb53*Ws[48*j]);
                r35three = R15three + R25three;
                r25three = R15three - R25three;
                r15three = S15three + S25three;
                s25three = S15three - S25three;

                x5[480+j+2*8] = xc53 - r35three;
```

```
        x5[480+j] = xc53  + r35three;
        x5[480+j+3*8] = xd53 - s25three;
        x5[480+j+8] = xd53 + s25three;
        y5[480+j+2*8] = yc53 - r15three;
        y5[480+j] = yc53 + r15three;
        y5[480+j+3*8] = yd53 + r25three;
        y5[480+j+8] = yd53 - r25three;
end;
(io:0..1)::
begin
        ya52 = y4[256*io+96 + 2*8+j];
        yb52 = y4[256*io+96 + 3*8+j];
        yc52 = y4[256*io+96+j];
        yd52 = y4[256*io+96+8+j] ;
        xa52 = x4[256*io+96+2*8+j];
        xb52 = x4[256*io+96+3*8+j];
        xc52 = x4[256*io+96+j];
        xd52 = x4[256*io+96+8+j];

        R15two = word(xa52*Wc[16*j]) + word(ya52*Ws[16*j]);
        S15two = word(ya52*Wc[16*j]) - word(xa52*Ws[16*j]);
        R25two = word(xb52*Wc[48*j]) + word(yb52*Ws[48*j]);
        S25two = word(yb52*Wc[48*j]) - word(xb52*Ws[48*j]);
        r35two = R15two + R25two;
        r25two = R15two - R25two;
        r15two = S15two + S25two;
        s25two = S15two - S25two;

        x5[256*io+96+2*8+j] = xc52 - r35two;
        x5[256*io+96+j] = xc52 + r35two;
        x5[256*io+96+3*8+j] = xd52 - s25two;
        x5[256*io+96+8+j] = xd52 + s25two;
        y5[256*io+96+2*8+j] = yc52 - r15two;
        y5[256*io+96+j] = yc52 + r15two;
        y5[256*io+96+3*8+j] = yd52 + r25two;
        y5[256*io+96+8+j] = yd52 - r25two;


end;
(io:0..7)::
begin
        ya51 = y4[64*io + 2*8+j];
        yb51 = y4[64*io +3*8+j];
        yc51 = y4[64*io+j];
        yd51 = y4[64*io+8+j] ;
        xa51 = x4[64*io+2*8+j];
        xb51 = x4[64*io+3*8+j];
```

```
                xc51 = x4[64*io+j];
                xd51 = x4[64*io+8+j];

                R15one = word(xa51*Wc[16*j]) + word(ya51*Ws[16*j]);
                S15one = word(ya51*Wc[16*j]) - word(xa51*Ws[16*j]);
                R25one = word(xb51*Wc[48*j]) + word(yb51*Ws[48*j]);
                S25one = word(yb51*Wc[48*j]) - word(xb51*Ws[48*j]);
                r35one = R15one + R25one;
                r25one = R15one - R25one;
                r15one = S15one + S25one;
                s25one = S15one - S25one;

                x5[64*io+2*8+j] = xc51 - r35one;
                x5[64*io+j] = xc51 + r35one;
                x5[64*io+3*8+j] = xd51- s25one;
                x5[64*io+8+j] = xd51 + s25one;
                y5[64*io+2*8+j] = yc51 - r15one;
                y5[64*io+j] = yc51 + r15one;
                y5[64*io+3*8+j] = yd51 + r25one;
                y5[64*io+8+j] = yd51 - r25one;
        end;


end;

 /* k = 6 */
 (j:0..15)::
 begin
        (io:0..0)::
        begin
                ya62 = y5[192 + 2*16+j];
                yb62 = y5[192 + 3*16+j];
                yc62 = y5[192+j];
                yd62 = y5[192+16+j] ;
                xa62 = x5[192+2*16+j];
                xb62 = x5[192+3*16+j];
                xc62 = x5[192+j];
                xd62 = x5[192+16+j];

                R16two = word(xa62*Wc[8*j]) + word(ya62*Ws[8*j]);
                S16two = word(ya62*Wc[8*j]) - word(xa62*Ws[8*j]);
                R26two = word(xb62*Wc[24*j]) + word(yb62*Ws[24*j]);
                S26two = word(yb62*Wc[24*j]) - word(xb62*Ws[24*j]);

                r36two = R16two + R26two;
                r26two = R16two - R26two;
                r16two = S16two + S26two;
```

```
                s26two = S16two - S26two;

                x6[192+j+2*16] = xc62 - r36two;
                x6[192+j] = xc62 + r36two;
                x6[192+j+3*16] = xd62 - s26two;
                x6[192+j+16] = xd62 + s26two;
                y6[192+j+2*16] = yc62 - r16two;
                y6[192+j] = yc62 + r16two;
                y6[192+j+3*16] = yd62 + r26two;
                y6[192+j+16] = yd62 - r26two;
        end;
        (io:0..3)::
        begin
                ya61 = y5[128*io + 2*16+j];
                yb61 = y5[128*io +3*16+j];
                yc61 = y5[128*io+j];
                yd61 = y5[128*io+16+j] ;
                xa61 = x5[128*io+2*16+j];
                xb61 = x5[128*io+3*16+j];
                xc61 = x5[128*io+j];
                xd61 = x5[128*io+16+j];

                R16one = word(xa61*Wc[8*j]) + word(ya61*Ws[8*j]);
                S16one = word(ya61*Wc[8*j]) - word(xa61*Ws[8*j]);
                R26one = word(xb61*Wc[24*j]) + word(yb61*Ws[24*j]);
                S26one = word(yb61*Wc[24*j]) - word(xb61*Ws[24*j]);
                r36one = R16one + R26one;
                r26one = R16one - R26one;
                r16one = S16one + S26one;
                s26one = S16one - S26one;

                x6[128*io+2*16+j] = xc61 - r36one;
                x6[128*io+j] = xc61 + r36one;
                x6[128*io+3*16+j] = xd61 - s26one;
                x6[128*io+16+j] = xd61 + s26one;
                y6[128*io+2*16+j] = yc61 - r16one;
                y6[128*io+j] = yc61 + r16one;
                y6[128*io+3*16+j] = yd61 + r26one;
                y6[128*io+16+j] = yd61 - r26one;
        end;

end;

 /* k = 7 */
 (j:0..31)::
 begin
      (io:0..0)::
```

```
begin
        ya72 = y6[384 + 2*32+j];
        yb72 = y6[384 +3*32+j];
        yc72 = y6[384+j];
        yd72 = y6[384+32+j] ;
        xa72 = x6[384+2*32+j];
        xb72 = x6[384+3*32+j];
        xc72 = x6[384+j];
        xd72 = x6[384+32+j];

        R17two = word(xa72*Wc[4*j]) + word(ya72*Ws[4*j]);
        S17two = word(ya72*Wc[4*j]) - word(xa72*Ws[4*j]);
        R27two = word(xb72*Wc[12*j]) + word(yb72*Ws[12*j]);
        S27two = word(yb72*Wc[12*j]) - word(xb72*Ws[12*j]);
        r37two = R17two + R27two;
        r27two = R17two - R27two;
        r17two = S17two + S27two;
        s27two = S17two - S27two;

        x7[384+j+2*32] = xc72 - r37two;
        x7[384+j] = xc72 + r37two;
        x7[384+j+3*32] = xd72 - s27two;
        x7[384+j+32] = xd72 + s27two;
        y7[384+j+2*32] = yc72 - r17two;
        y7[384+j] = yc72 + r17two;
        y7[384+j+3*32] = yd72 + r27two;
        y7[384+j+32] = yd72 - r27two;
end;
(io:0..1)::
begin
        ya71 = y6[256*io + 2*32+j];
        yb71 = y6[256*io +3*32+j];
        yc71 = y6[256*io+j];
        yd71 = y6[256*io+32+j] ;
        xa71 = x6[256*io+2*32+j];
        xb71 = x6[256*io+3*32+j];
        xc71 = x6[256*io+j];
        xd71 = x6[256*io+32+j];

        R17one = word(xa71*Wc[4*j]) + word(ya71*Ws[4*j]);
        S17one = word(ya71*Wc[4*j]) - word(xa71*Ws[4*j]);
        R27one = word(xb71*Wc[12*j]) + word(yb71*Ws[12*j]);
        S27one = word(yb71*Wc[12*j]) - word(xb71*Ws[12*j]);
        r37one = R17one + R27one;
        r27one = R17one - R27one;
        r17one = S17one + S27one;
        s27one = S17one - S27one;
```

```
                x7[256*io+2*32+j] = xc71 - r37one;
                x7[256*io+j] = xc71 + r37one;
                x7[256*io+3*32+j] = xd71 - s27one;
                x7[256*io+32+j] = xd71 + s27one;
                y7[256*io+64+j] = yc71 - r17one;
                y7[256*io+j] = yc71+ r17one;
                y7[256*io+3*32+j] = yd71 + r27one;
                y7[256*io+32+j] = yd71 - r27one;
        end;

end;

 /* k = 8 */
 (j:0..63)::
 begin
                ya81 = y7[2*64+j];
                yb81 = y7[3*64+j];
                yc81 = y7[j];
                yd81 = y7[64+j] ;
                xa81 = x7[2*64+j];
                xb81 = x7[3*64+j];
                xc81 = x7[j];
                xd81 = x7[64+j];


                R18one = word(xa81*Wc[2*j]) + word(ya81*Ws[2*j]);
                S18one = word(ya81*Wc[2*j]) - word(xa81*Ws[2*j]);
                R28one = word(xb81*Wc[6*j]) + word(yb81*Ws[6*j]);
                S28one = word(yb81*Wc[6*j]) - word(xb81*Ws[6*j]);
                r38one = R18one + R28one;
                r28one = R18one - R28one;
                r18one = S18one + S28one;
                s28one = S18one - S28one;

                x8[j+2*64] = xc81 - r38one;
                x8[j] = xc81 + r38one;
                x8[j+3*64] = xd81 - s28one;
                x8[j+64] = xd81 + s28one;
                y8[j+2*64] = yc81 - r18one;
                y8[j] = yc81 + r18one;
                y8[j+3*64] = yd81 + r28one;
                y8[j+64] = yd81 - r28one;

 end;

  /* k = 9 */
```

```
(j:0..127)::
begin
        ya91 = y8[2*128+j];
        yb91 = y8[3*128+j];
        yc91 = y8[j];
        yd91 = y8[128+j] ;
        xa91 = x8[2*128+j];
        xb91 = x8[3*128+j];
        xc91 = x8[j];
        xd91 = x8[128+j];


        R19 = word(xa91*Wc[j]) + word(ya91*Ws[j]);
        S19 = word(ya91*Wc[j]) - word(xa91*Ws[j]);
        R29 = word(xb91*Wc[3*j]) + word(yb91*Ws[3*j]);
        S29 = word(yb91*Wc[3*j]) - word(xb91*Ws[3*j]);
        r39 = R19 + R29;
        r29 = R19 - R29;
        r19 = S19 + S29;
        s29 = S19 - S29;
    xa = xc91+ r39;
        xb = xd91 + s29;
        ya = yc91 + r19;
        yb = yd91 - r29;
        Power[j] = wordout(xa * xa) + wordout(ya * ya);
        Power[j+128] = wordout(xb * xb) + wordout(yb * yb);

  end;
end;
```

```
/*
 *Author - Engling Yeo
 * cbi.sil implements all cbweights as constants by putting the numbers
 * into an array.  Could be implemented on hardware as constants in ROMs
 *
*/

#define word fix<13,12>

func main(In:word[256]; cb1:word[8]; cb2:word[12];
        cb3:word[15]; cb4:word[15]; cb5:word[17];
        cb6:word[18]; cb7:word[21]; cb8:word[23];
        cb9:word[27]; cb10:word[31]; cb11:word[36];
        cb12:word[42]; cb13:word[49]; cb14:word[57];
        cb15:word[67]; cb16:word[79]; cb17:word[93];
        cb18:word[109]; cb19:word[113]) Out: word[] =
begin


  Out1##1 = word(0);
  (i1:0..7)::
  begin
    Out1 = word(In[i1] * cb1[i1]) + Out1#1;
  end;
  Out[0] = Out1;


  Out2##1 = word(0);
  (i2:0..11)::
  begin
    Out2 = word(In[i2] * cb2[i2]) + Out2#1;
  end;
  Out[1] = Out2;

  Out3##1 = word(0);
  (i3:0..14)::
  begin
     Out3  = word(In[i3+1] * cb3[i3]) + Out3#1;
  end;
  Out[2] = Out3;


  Out4##1 = word(0);
  (i4:0..14)::
  begin
     Out4 = word(In[i4+5] * cb4[i4]) + Out4#1;
  end;
```

```
Out[3] = Out4;


Out5##1 = word(0);
(i5:0..16)::
begin
  Out5 = word(In[i5+8] * cb5[i5]) + Out5#1;
end;
Out[4] = Out5;


Out6##1 = word(0);
(i6:0..17)::
begin
  Out6 = word(In[i6+12] * cb6[i6]) + Out6#1;
end;
Out[5] = Out6;


Out7##1 = word(0);
(i7:0..20)::
begin
  Out7 = word(In[i7+15] * cb7[i7]) + Out7#1;
end;
Out[6] = Out7;


Out8##1 = word(0);
(i8:0..22)::
begin
  Out8 = word(In[i8 + 20] * cb8[i8]) + Out8#1;
end;
Out[7] = Out8;


Out9##1 = word(0);
(i9:0..26)::
begin
  Out9  = word(In[i9+24] * cb9[i9]) + Out9#1;
end;
Out[8] = Out9;


Out10##1 = word(0);
(i10:0..30)::
begin
  Out10 = word(In[i10+30] * cb10[i10]) + Out10#1;
```

```
end;
Out[9] = Out10;


Out11##1 = word(0);
(i11:0..35)::
begin
  Out11 = word(In[i11+36] * cb11[i11]) + Out11#1;
end;
Out[10] = Out11;


Out12##1 = word(0);
(i12:0..41)::
begin
  Out12 = word(In[i12+43] * cb12[i12]) + Out12#1;
end;
Out[11] = Out12;


Out13##1 = word(0);
(i13:0..48)::
begin
  Out13 = word(In[i13+52] * cb13[i13]) + Out13#1;
end;
Out[12] = Out13;


Out14##1 = word(0);
(i14:0..56)::
begin
  Out14 = word(In[i14+62] * cb14[i14]) + Out14#1;
end;
Out[13] = Out14;


Out15##1 = word(0);
(i15:0..66)::
begin
  Out15 = word(In[i15+73] * cb15[i15]) + Out15#1;
end;
Out[14] = Out15;


Out16##1 = word(0);
(i16:0..78)::
begin
```

```
    Out16= word(In[i16+86] * cb16[i16]) + Out16#1;
  end;
  Out[15] = Out16;


  Out17##1 = word(0);
  (i17:0..92)::
  begin
    Out17 = word(In[i17+102] * cb17[i17]) + Out17#1;
  end;
  Out[16] = Out17;


  Out18##1 = word(0);
  (i18:0..108)::
  begin
    Out18 = word(In[i18+121] * cb18[i18]) + Out18#1;
  end;
  Out[17] = Out18;

  Out19##1 = word(0);
  (i19:0..112)::
  begin
    Out19 = word(In[i19+143] * cb19[i19]) + Out19#1;
  end;
  Out[18] = Out19;



end;
```

```
/*
 *Author - Engling Yeo
 *        - Sean Huang
 * cbiNew.sil implements all cbweights as constants by putting the numbers
 * into an array.  Could be implemented on hardware as constants in ROMs
 *
*/

#define wordc fix<13,12>
#define wordin fix<43,24>
#define word fix<50,24>
#define cosu word(0.707106781)
#define sinu (0.707106781)
#define half_pi word(1.57079633)
#define pi word(3.14159265)
#define b1_0 word(0.987862135574673807)
#define b3_0 word(-0.155271410633428645)
#define b5_0 word(0.0056431179763468104)
#define WN word(0.012271846) /* = (2*pi)/512 */
#define barkstep word(1.09)


func main(In:wordin[256]; cb1:wordc[8];
                          cb2:wordc[12];
                          cb3:wordc[15];
                          cb4:wordc[15];
                          cb5:wordc[17];
                          cb6:wordc[18];
                          cb7:wordc[21];
                          cb8:wordc[23];
                          cb9:wordc[27];
                          cb10:wordc[31];
                          cb11:wordc[36];
                          cb12:wordc[42];
                          cb13:wordc[49];
                          cb14:wordc[57];
                          cb15:wordc[67];
                          cb16:wordc[79];
                          cb17:wordc[93];
                          cb18:wordc[109];
                          cb19:wordc[113])
                          Out: wordc[] =
begin
/*
 * In: 0
 */
  a1 = In[0];
```

```
  Filt11 = word(a1 * cb1[0]);
  Filt21 = word(a1 * cb2[0]);


/*
 * In: 1..4
 */
  Filt12##1 = Filt11;
  Filt22##1 = Filt21;
  Filt31##1 = word(0);
  (i2:0..3)::
  begin
    a2 = In[i2+1];
    Filt12 = word(a2 * cb1[i2+1]) + Filt12#1;
    Filt22 = word(a2 * cb2[i2+1]) + Filt22#1;
    Filt31 = word(a2 * cb3[i2]) + Filt31#1;
  end;

/*
 * In: 5..7
 */
  Filt13##1 = Filt12;
  Filt23##1 = Filt22;
  Filt32##1 = Filt31;
  Filt41##1 = word(0);
  (i3:0..2)::
  begin
    a3 = In[i3+5];
    Filt13 = word(a3 * cb1[i3+5]) + Filt13#1;
    Filt23 = word(a3 * cb2[i3+5]) + Filt23#1;
    Filt32 = word(a3 * cb3[i3+4]) + Filt32#1;
    Filt41 = word(a3 * cb4[i3]) + Filt41#1;
  end;


/*
 * In: 8..11
 */
  Filt24##1 = Filt23;
  Filt33##1 = Filt32;
  Filt42##1 = Filt41;
  Filt51##1 = word(0);
  (i4:0..3)::
  begin
    a5 = In[i4+8];
    Filt24 = word(a5 * cb2[i4+8]) + Filt24#1;
    Filt33 = word(a5 * cb3[i4+7]) + Filt33#1;
```

```
    Filt42 = word(a5 * cb4[i4+3]) + Filt42#1;
    Filt51 = word(a5 * cb5[i4]) + Filt51#1;
   end;

 /*
  * In: 12..14
  */
   Filt34##1 = Filt33;
   Filt43##1 = Filt42;
   Filt52##1 = Filt51;
   Filt61##1 = word(0);
   (i6:0..2)::
   begin
    a6 = In[i6+12];
    Filt34 = word(a6 * cb3[i6+11]) + Filt34#1;
    Filt43 = word(a6 * cb4[i6+7]) + Filt43#1;
    Filt52 = word(a6 * cb5[i6+4]) + Filt52#1;
    Filt61 = word(a6 * cb6[i6]) + Filt61#1;
   end;

 /*
  * In: 15
  */
  Filt35##1 = Filt34;
  Filt44##1 = Filt43;
  Filt53##1 = Filt52;
  Filt62##1 = Filt61;
  Filt71##1 = word(0);
 (i7:0..0)::
 begin
    a7 = In[15];
    Filt35 = word(a7 * cb3[i7+14]) + Filt35#1;
    Filt44 = word(a7 * cb4[i7+10]) + Filt44#1;
    Filt53 = word(a7 * cb5[i7+7]) + Filt53#1;
    Filt62 = word(a7 * cb6[i7+3]) + Filt62#1;
    Filt71 = word(a7 * cb7[i7+0]) + Filt71#1;
 end;

 /*
  * In: 16..19
  */
   Filt45##1 = Filt44;
   Filt54##1 = Filt53;
   Filt63##1 = Filt62;
   Filt72##1 = Filt71;
   (i8:0..3)::
   begin
```

```
  a8 = In[i8+16];
  Filt45 = word(a8 * cb4[i8+11]) + Filt45#1;
  Filt54 = word(a8 * cb5[i8+8]) + Filt54#1;
  Filt63 = word(a8 * cb6[i8+4]) + Filt63#1;
  Filt72 = word(a8 * cb7[i8+1]) + Filt72#1;
 end;

/*
 * In: 20..24
 */
 Filt55##1 = Filt54;
 Filt64##1 = Filt63;
 Filt73##1 = Filt72;
 Filt81##1 = word(0);
 (i9:0..4)::
 begin
  a9 = In[i9+20];
  Filt55 = word(a9 * cb5[i9+12]) + Filt55#1;
  Filt64 = word(a9 * cb6[i9+8]) + Filt64#1;
  Filt73 = word(a9 * cb7[i9+5]) + Filt73#1;
  Filt81 = word(a9 * cb8[i9]) + Filt81#1;
 end;

/*
 * In: 25..29
 */
 Filt65##1 = Filt64;
 Filt74##1 = Filt73;
 Filt82##1 = Filt81;
 Filt91##1 = word(In[24] * cb9[0]);
 (i10:0..4)::
 begin
  a10 = In[i10+25];
  Filt65 = word(a10 * cb6[i10+13]) + Filt65#1;
  Filt74 = word(a10 * cb7[i10+10]) + Filt74#1;
  Filt82 = word(a10 * cb8[i10+5]) + Filt82#1;
  Filt91 = word(a10 * cb9[i10+1]) + Filt91#1;
 end;

/*
 * In: 30..35
 */
 Filt75##1 = Filt74;
 Filt83##1 = Filt82;
 Filt92##1 = Filt91;
 Filt101##1 = word(0);
 (i11:0..5)::
```

```
 begin
  a11 = In[i11+30];
  Filt75 = word(a11 * cb7[i11+15]) + Filt75#1;
  Filt83 = word(a11 * cb8[i11+10]) + Filt83#1;
  Filt92 = word(a11 * cb9[i11+6]) + Filt92#1;
  Filt101 = word(a11 * cb10[i11]) + Filt101#1;
 end;

/*
 * In: 36..42
 */
 Filt84##1 = Filt83;
 Filt93##1 = Filt92;
 Filt102##1 = Filt101;
 Filt111##1 = word(0);
 (i12:0..6)::
 begin
  a12 = In[i12+36];
  Filt84 = word(a12 * cb8[i12+16]) + Filt84#1;
  Filt93 = word(a12 * cb9[i12+12]) + Filt93#1;
  Filt102 = word(a12 * cb10[i12+6]) + Filt102#1;
  Filt111 = word(a12 * cb11[i12]) + Filt111#1;
 end;

/*
 * In: 43..50
 */
 Filt94##1 = Filt93;
 Filt103##1 = Filt102;
 Filt112##1 = Filt111;
 Filt121##1 = word(0);
 (i13:0..7)::
 begin
  a13 = In[i13+43];
  Filt94 = word(a13 * cb9[i13+19]) + Filt94#1;
  Filt103 = word(a13 * cb10[i13+13]) + Filt103#1;
  Filt112 = word(a13 * cb11[i13+7]) + Filt112#1;
  Filt121 = word(a13 * cb12[i13]) + Filt121#1;
 end;


/*
 * In: 51
 */
 Filt104##1 = Filt103;
 Filt113##1 = Filt112;
 Filt122##1 = Filt121;
```

```
 (i14:0..0)::
 begin
   a14 = In[51];
   Filt104 = word(a14 * cb10[21+i14]) + Filt104#1;
   Filt113 = word(a14 * cb11[15+i14]) + Filt113#1;
   Filt122 = word(a14 * cb12[8+i14]) + Filt122#1;
 end;

/*
 * In: 52..60
 */
 Filt105##1 = Filt104;
 Filt114##1 = Filt113;
 Filt123##1 = Filt122;
 Filt131##1 = word(0);
 (i15:0..8)::
 begin
   a15 = In[i15+52];
   Filt105 = word(a15 * cb10[i15+22]) + Filt105#1;
   Filt114 = word(a15 * cb11[i15+16]) + Filt114#1;
   Filt123 = word(a15 * cb12[i15+9]) + Filt123#1;
   Filt131 = word(a15 * cb13[i15]) + Filt131#1;
 end;

/*
 * In: 61
 */
 Filt115##1 = Filt114;
 Filt124##1 = Filt123;
 Filt132##1 = Filt131;
 (i16:0..0)::
 begin
   a16 = In[61];
   Filt115 = word(a16 * cb11[25+i16]) + Filt115#1;
   Filt124 = word(a16 * cb12[18+i16]) + Filt124#1;
   Filt132 = word(a16 * cb13[9+i16] ) + Filt132#1;
 end;


/*
 * In: 62..71::
 */
Filt116##1 = Filt115;
Filt125##1 = Filt124;
Filt133##1 = Filt132;
Filt141##1 = word(0);
(i17:0..9)::
```

```
begin
   a17 = In[i17+62];
   Filt116 = word(a17 * cb11[i17+26]) + Filt116#1;
   Filt125 = word(a17 * cb12[i17+19]) + Filt125#1;
   Filt133 = word(a17 * cb13[i17+10]) + Filt133#1;
   Filt141 = word(a17 * cb14[i17]) + Filt141#1;
end;

/*
* In: 72
*/
Filt126##1 = Filt125;
Filt134##1 = Filt133;
Filt142##1 = Filt141;
(i18:0..0)::
begin
   a18 = In[72];
   Filt126 = word(a18 * cb12[29+i18]) + Filt126#1;
   Filt134 = word(a18 * cb13[20+i18]) + Filt134#1;
   Filt142 = word(a18 * cb14[10+i18]) + Filt142#1;
end;

/*
* In: 73..84
*/
Filt127##1 = Filt126;
Filt135##1 = Filt134;
Filt143##1 = Filt142;
Filt151##1 = word(0);
(i19:0..11)::
begin
   a19 = In[i19+73];
   Filt127 = word(a19 * cb12[i19+30]) + Filt127#1;
   Filt135 = word(a19 * cb13[i19+21]) + Filt135#1;
   Filt143 = word(a19 * cb14[i19+11]) + Filt143#1;
   Filt151 = word(a19 * cb15[i19]) + Filt151#1;
end;

/*
* In: 85
*/
Filt136##1 = Filt135;
Filt144##1 = Filt143;
Filt152##1 = Filt151;
(i20:0..0)::
begin
   a20 = In[85];
```

```
    Filt136 = word(a20 * cb13[i20+33]) + Filt136#1;
    Filt144 = word(a20 * cb14[i20+23]) + Filt144#1;
    Filt152 = word(a20 * cb15[i20+12]) + Filt152#1;
end;

/*
 * In: 86..100
 */
Filt137##1 = Filt136;
Filt145##1 = Filt144;
Filt153##1 = Filt152;
Filt161##1 = word(0);
(i21:0..14)::
begin
    a21 = In[i21+86];
    Filt137 = word(a21 * cb13[i21+34]) + Filt137#1;
    Filt145 = word(a21 * cb14[i21+24]) + Filt145#1;
    Filt153 = word(a21 * cb15[i21+13]) + Filt153#1;
    Filt161 = word(a21 * cb16[i21]) + Filt161#1;
end;

/*
 * In:  101
 */
Filt146##1 = Filt145;
Filt154##1 = Filt153;
Filt162##1 = Filt161;
(i22:0..0)::
begin
    a22 = In[101];
    Filt146 = word(a22 * cb14[39+i22]) + Filt146#1;
    Filt154 = word(a22 * cb15[28+i22]) + Filt154#1;
    Filt162 = word(a22 * cb16[15+i22]) + Filt162#1;
end;


/*
 * In: 102..118
 */
Filt147##1 = Filt146;
Filt155##1 = Filt154;
Filt163##1 = Filt162;
Filt171##1 = word(0);
(i23:0..16)::
begin
    a23 = In[i23+102];
    Filt147 = word(a23 * cb14[i23+40]) + Filt147#1;
```

```
    Filt155 = word(a23 * cb15[i23+29]) + Filt155#1;
    Filt163 = word(a23 * cb16[i23+16]) + Filt163#1;
    Filt171 = word(a23 * cb17[i23]) + Filt171#1;
end;


/*
 * In: 119..120
 */
Filt156##1 = Filt155;
Filt164##1 = Filt163;
Filt172##1 = Filt171;
(i24:0..1)::
begin
    a24 = In[i24+119];
    Filt156 = word(a24 * cb15[i24+46]) + Filt156#1;
    Filt164 = word(a24 * cb16[i24+33]) + Filt164#1;
    Filt172 = word(a24 * cb17[i24+17]) + Filt172#1;
end;

/*
 * In: 121..139
 */
Filt157##1 = Filt156;
Filt165##1 = Filt164;
Filt173##1 = Filt172;
Filt181##1 = word(0);
(i25:0..18)::
begin
    a25 = In[i25+121];
    Filt157 = word(a25 * cb15[i25+48]) + Filt157#1;
    Filt165 = word(a25 * cb16[i25+35]) + Filt165#1;
    Filt173 = word(a25 * cb17[i25+19]) + Filt173#1;
    Filt181 = word(a25 * cb18[i25]) + Filt181#1;
end;

/*
 * In: 140..142
 */
Filt166##1 = Filt165;
Filt174##1 = Filt173;
Filt182##1 = Filt181;
(i26:0..2)::
begin
    a26 = In[i26+140];
    Filt166 = word(a26 * cb16[i26+54]) + Filt166#1;
    Filt174 = word(a26 * cb17[i26+38]) + Filt174#1;
```

```
   Filt182 = word(a26 * cb18[i26+19]) + Filt182#1;
end;

/*

/*
 * In:143..164
 */
Filt167##1 = Filt166;
Filt175##1 = Filt174;
Filt183##1 = Filt182;
Filt191##1 = word(0);
(i27:0..21)::
begin
   a27 = In[i27+143];
   Filt167 = word(a27 * cb16[i27+57]) + Filt167#1;
   Filt175 = word(a27 * cb17[i27+41]) + Filt175#1;
   Filt183 = word(a27 * cb18[i27+22]) + Filt183#1;
   Filt191 = word(a27 * cb19[i27]) + Filt191#1;
end;

/*
 * In: 165..194
 */
Filt176##1 = Filt175;
Filt184##1 = Filt183;
Filt192##1 = Filt191;
(i28:0..29)::
begin
   a28 = In[i28+165];
   Filt176 = word(a28 * cb17[i28+63]) + Filt176#1;
   Filt184 = word(a28 * cb18[i28+44]) + Filt184#1;
   Filt192 = word(a28 * cb19[i28+22]) + Filt192#1;
end;

/*
 * In: 195..229
 */
Filt185##1 = Filt184;
Filt193##1 = Filt192;
(i29:0..34)::
begin
   a29 = In[i29+195];
   Filt185 = word(a29 * cb18[i29+74]) + Filt185#1;
   Filt193 = word(a29 * cb19[i29+52]) + Filt193#1;
end;
```

```
/*
 *In: 230..255
 */
 Filt194##1 = Filt193;
 (i30:0..25)::
 begin
    a30 = In[i30+230];
    Filt194 = word(a30 * cb19[i30+87]) + Filt194#1;
 end;


/*****************************/

  Out[0] = Filt13;
  Out[1] = Filt24;
  Out[2] = Filt35;
  Out[3] = Filt45;
  Out[4] = Filt55;
  Out[5] = Filt65;
  Out[6] = Filt75;
  Out[7] = Filt84;
  Out[8] = Filt94;
  Out[9] = Filt105;
  Out[10] = Filt116;
  Out[11] = Filt127;
  Out[12] = Filt137;
  Out[13] = Filt147;
  Out[14] = Filt157;
  Out[15] = Filt167;
  Out[16] = Filt176;
  Out[17] = Filt185;
  Out[18] = Filt194;
 end;
```

```
/*
 *Author - Engling Yeo
 *
*/

#define word fix<25,20>
#define cosu word(0.707106781)
#define sinu (0.707106781)
#define half_pi word(1.57079633)
#define pi word(3.14159265)
#define b1_0 word(0.987862135574673807)
#define b3_0 word(-0.155271410633428645)
#define b5_0 word(0.0056431179763468104)
#define WN word(0.012271846) /* = (2*pi)/512 */



func main(In:word[19]) RastaOut: word[] =
begin
 (n:0..18)::
 begin
   Out[n]@@1 = word(0);
   In1[n]@@1 = word(0);
   In1[n]@@2 = word(0);
   In1[n]@@3 = word(0);
   In1[n]@@4 = word(0);
 end;

 (i:0..18)::  /* Rasta Filtering */
 begin
   In1[i] = In[i];
   Out[i] = word(word(Out[i]@1 * word(0.94))
                 + word(word(0.2) * In1[i])
                 + word(word(0.1) * In1[i]@1)
                 - word(word(0.1) * In1[i]@3)
                 - word(word(0.2) * In1[i]@4));
 end;

/* Power Law and Equal Loudness Preemphasis */

  RastaOut[0] = word(word(0.33) * (word(-7.598859) + Out[0]));
  RastaOut[1] = word(word(0.33) * (word(-5.084743) + Out[1]));
  RastaOut[2] = word(word(0.33) * (word(-3.824229) + Out[2]));
  RastaOut[3] = word(word(0.33) * (word(-3.077419) + Out[3]));
  RastaOut[4] = word(word(0.33) * (word(-2.588522) + Out[4]));
  RastaOut[5] = word(word(0.33) * (word(-2.237648) + Out[5]));
  RastaOut[6] = word(word(0.33) * (word(-1.96175) + Out[6]));
```

```
  RastaOut[7] = word(word(0.33) * (word(-1.726257) + Out[7]));
  RastaOut[8] = word(word(0.33) * (word(-1.512501) + Out[8]));
  RastaOut[9] = word(word(0.33) * (word(-1.311570) + Out[9]));
  RastaOut[10] = word(word(0.33) * (word(-1.12068) + Out[10]));
  RastaOut[11] = word(word(0.33) * (word(-0.940791) + Out[11]));
  RastaOut[12] = word(word(0.33) * (word(-0.774623) + Out[12]));
  RastaOut[13] = word(word(0.33) * (word(-0.625262) + Out[13]));
  RastaOut[14] = word(word(0.33) * (word(-0.495046) + Out[14]));
  RastaOut[15] = word(word(0.33) * (word(-0.384992) + Out[15]));
  RastaOut[16] = word(word(0.33) * (word(-0.294683) + Out[16]));
  RastaOut[17] = word(word(0.33) * (word(-0.222522) + Out[17]));
  RastaOut[18] = word(word(0.33) * (word(-0.166165) + Out[18]));

end;
```

```
/*
 * File:
 *   ln.sil by Steve Stoiber & Engling Yeo
 *   Latest, best working ln operation 3/24/95
 *
 * Purpose:
 *   return ln(x) if x > 0
 *   else return EXCEPTION
 *
 *
 * How it works:
 *   ln(x) = ln(2^p * y)     1 <= y < 2
 *         = p*ln(2) + ln(y)
 *         = p*ln(2) + a1*y + a2*y^2 + a3*y^3 + a4*y^4 + a5*y^5
*/

#define word     fix<33,17>
#define wordout fix<25,17>

#define N14
#define MIN_NUMword(6.103515e-05)
#define MSB_POW14 /* mag msb has weight 2^MSB_POW */
#define MSB_WGHT16384 /* mag msb has weight MSB_WGHT */

#define a1        wordout( 0.99949556)
#define a2        wordout(-0.49190896)
#define a3        wordout( 0.28947478)
#define a4        wordout(-0.13606275)
#define a5        wordout( 0.03215845)
#define ln2     wordout( 0.69314718)
#define One       wordout(1)



func main (x : word[19]) y: wordout[] =
begin
  (i:0..18)::
  begin
    y[i] = log_gtz(x[i]);
  end;
end;


func log_gtz(x : word) y : wordout =
begin
  x_tmp[0] = x;
  y_tmp[0] = x;
```

```
  lead_one_pow[0] = word(MSB_POW);

  (i : 1 .. N) ::
    begin
      (x_tmp[i], y_tmp[i], lead_one_pow[i]) =
        if ((x_tmp[i - 1] & MSB_WGHT) ==0) ->
              (x_tmp[i-1] << 1, y_tmp[i-1], lead_one_pow[i-1] - 1)
        ||
              (x_tmp[i-1], y_tmp[i-1] >> 1, lead_one_pow[i-1])
        fi;
    end;
  ln_mantissa = log_restricted(wordout(y_tmp[N]));
  ln_power = wordout(ln2 * lead_one_pow[N]);

  y = ln_power + ln_mantissa;
end;


func log_restricted (in : wordout) out : wordout =
begin
  out = wordout(a1*(in-One)) +
      wordout(a2*wordout((in-One)*(in-One))) +
      wordout(wordout(a3*(in-One))*wordout((in-One)*(in-One))) +
      wordout(wordout(wordout(a4*(in-One))*(in-One))*wordout((in-One)*(in-
One))) +
      wordout(wordout(wordout(a5*(in-One))*wordout((in-One)*(in-
One)))*wordout((in-One)*(in-One)));
end;
```

```
/*
 * File:
 *   exp.sil by Steve Stoiber
 *
 *
 * Purpose:
 *   return exp(x)
 *
 *
 * How it works:
 *   first notice that the convergence rate of the geometric series
 *   is awful (consider x < 0), so we need to do something else
 *
 *   e^x = 2^p
 *       = 2^i * 2^f where i is an integer and 0 <= f < 1
 *       = (shift) * (polynomial approximation)
 *
 *   how to find p?
 *     e^x = 2^p
 *     ln(e^x) = ln(2^p)
 *     x = p*ln(2)
 *     p = x / ln(2)
 *       = x * log2(e)
*/


#define word     fix<25,20>
#define LOG_2_eword(1.442695)
#define INT_MASK15
#define FRAC_MASK0.999999046
#define W_BITS4 /* No. of integer bits */

#define a1       word(0.693147) /* log_e_2 */
#define a2       word(0.240227) /* 1/2 * log_e_2^2 */
#define a3       word(0.055504) /* 1/6 * log_e_2^3 */
#define a4       word(0.009618) /* 1/24 * log_e_2^4 */


func main (x : word[19]) y : word[19] =
begin
 (b:0..18)::
 begin
   y[b] = exp_generic(x[b]);
 end;
end;
```

```
func exp_generic (x : word) y : word =
begin
  p = word(LOG_2_e * x);
  z =
    if (p >= 0.0) ->
          get_integer(p)
    ||
          word(1.0 + get_integer(-p))
    fi;
  f =
    if (p >= 0.0) ->
          get_frac(p)
    ||
          word(1.0 - get_frac(-p))
    fi;
  a = pow2(f);
  y =
    if (p >= 0.0) ->
          word(a * mult_by_two(z))
    ||
          word(a * div_by_two(z))
    fi;
end;


func mult_by_two(miter : word) y : word =
begin
  y_tmp[0] = word(1.0);
  count[0] = miter;

  (i : 1 .. W_BITS) ::
    begin
      y_tmp[i] =
          if (count[i-1] == 0) ->
            y_tmp[i-1]
        ||
            word(y_tmp[i-1]<< 1)
        fi;
      count[i] =
          if (count[i-1] == 0) ->
            0
        ||
            count[i-1] - 1
        fi;
    end;
  y = y_tmp[W_BITS];
end;
```

```
func div_by_two(diter : word) y : word =
begin
  y_tmp[0] = 1.0;
  count[0] = diter;

  (i : 1 .. W_BITS) ::
    begin
      y_tmp[i] =
           if (count[i-1] == 0) ->
             y_tmp[i-1]
        ||
           word(y_tmp[i-1]>>1)
        fi;
      count[i] =
           if (count[i-1] == 0) ->
             0
        ||
           count[i-1] - 1
        fi;
    end;
  y = y_tmp[W_BITS];
end;


func pow2(f : word) y : word =
begin
  t0 = word(1.0);
  t1 = word(a1 * f);
  t2 = word(word(a2 * f) * f);
  t3 = word(word(a3*f) * word(f*f));
  t4 = word(word(a4 * word(f*f)) * word(f*f));
  y = word(t0 + t1 + t2 + t3 + t4);
end;


func get_integer (x : word) y : word =
begin
  y = x & INT_MASK;
end;


func get_frac (x : word) y : word =
begin
  y = x & FRAC_MASK;
end;
```

```
#define wordfix<25,20>
#define NFREQ  21
#define NAUTO9

/*  IDFT computes the first 9 autocorrelation coefficients from power
    spectrum coefficients.  freq[1 throough 19] are obtained directly
    from previous computations, while the first and last values ( freq[0]
    and freq[20] ) are copied from their neighbors.  i.e. freq[0] = freq[1],
    and freq[20] = freq[19]
*/
func main (freq : word[NFREQ]; wcos: word[NAUTO][NFREQ-2]) autoid :
word[] =
begin

  (i : 0 .. NAUTO - 1) ::
    begin
         tmp[i][0] = freq[0]-freq[20];
         (j : 1 .. NFREQ-2) ::
           begin
             tmp[i][j] = tmp[i][j-1] + word(freq[j] * wcos[i][j-1]);
        end;
      autoid[i] = tmp[i][NFREQ-2];
    end;
end;
```

```
/* Ingrid Verbauwhede */
/* Durbin algorithm */
/* see Golub, "Matrix Computations", p. 127
 * 13 aug 1990
 * works for the example of pp. 127 !!
 *  Solves the problem
 *
 *  T x y = r,
 *
 * where:  r = -(r1, r2, ..., rN)
 *         T : a persymmetric NxN matrix
 *         y = (y1, y2, ..., yN)
 */

#define N       8 /* for an NxN matrix */
#define NDIV    24
#define real  fix<25,20>
#define NDIV1  25
#define numDIVfix<NDIV+1,0>
#define num1DIVfix<NDIV+2,0>
#define num2DIVfix<NDIV+1,NDIV>
#define num3DIVfix<NDIV+2,NDIV>
#define num4DIV fix<(2*NDIV)+1,NDIV>



func main (In : real[N+1] )
        out : real[] =

begin
 norm = ReciprocalnonFraction(In[0]);
 (j:1 .. N)::
 begin
   r[j-1] = real(norm*In[j]);
 end;
 y[0][0] = - r[0];
 beta##1 = real(1);
 alpha##1 = - r[0];
(k : 1 .. N-1) ::
 begin
  lastalpha = alpha#1;
  beta = real(( 1 - real(lastalpha*lastalpha)) * beta#1);
  sum[0][k] = r[k] ;
 (i : 0 .. k-1) ::
  begin
   sum[i+1][k] = sum[i][k]  + real(r[k-i-1]*y[i][k-1]) ;
  end;
```

```
  B = ReciprocalFraction(beta);
        /* Since 1< beta < 0.5 for all samples tested */
  alpha = - real(sum[k][k] * B);
 (i : 0 .. k-1) ::
  begin
   y[i][k]    = y[i][k-1]    + real(alpha*y[k-i-1][k-1]);
  end;
  y[k][k] = alpha;
 end;
 (i : 0 .. N-1 ) ::
 begin
  out[i+1] = y[i][N-1] ;
 end;
end;


func divfix2(a,b: num2DIV) quo: num2DIV =
begin
        R[0] = a;
        P[0] = num2DIV(0);
        (m: 1 .. NDIV) ::
        begin
                (R[m], P[m]) =
                 if ((R[m-1]>=0 & b>=0) | (R[m-1]<0 &  b<0))
                  -> (num2DIV(num3DIV(R[m-1])<<1 - num3DIV(b)),
                        num2DIV(P[m-1]<<1 + num2DIV(6.08e-8)))
                 || ((R[m-1]>=0 &  b<0) | (R[m-1]<0 & b>=0))
                  -> (num2DIV(num3DIV(R[m-1])<<1 + num3DIV(b)),
                        num2DIV(P[m-1]<<1))
                 fi;
        end;

        quo = num2DIV(num3DIV(P[NDIV])<<1 - num3DIV(1) +
                num3DIV(6.08e-8)) ;
end;


func divfix(a,b: num2DIV) quo: num2DIV =
begin
        R[0] = a;
        P[0] = num2DIV(0);
        (m: 1 .. NDIV) ::
        begin
                (R[m], P[m]) =
                 if ((R[m-1]>=0 & b>=0) | (R[m-1]<0 &  b<0))
                  -> (num2DIV(num3DIV(R[m-1])<<1 - num3DIV(b)),
                        num2DIV(P[m-1]<<1 + num2DIV(6.08e-8)))
```

```
                || ((R[m-1]>=0 &  b<0) | (R[m-1]<0 & b>=0))
                 -> (num2DIV(num3DIV(R[m-1])<<1 + num3DIV(b)),
                        num2DIV(P[m-1]<<1))
                fi;
        end;

        quo = num2DIV(num3DIV(P[NDIV])<<1 - num3DIV(1) +
                num3DIV(6.08e-8)) ;
end;


func ReciprocalnonFraction(x: real) q: num4DIV =
/* |x| > 1 */
begin
        x1 = if (x<0) -> -x
           || x
           fi;
     xtemp =num2DIV(fix<(2*NDIV)+1,NDIV>(x1)>>10);
        q1 = num4DIV(divfix(num2DIV(0.000976563),xtemp));
                                        /* 2^-10 */
     q = if (x<0) -> -q1
         || q1
         fi;

end;

func ReciprocalFraction(x: real) qf: num4DIV =
/* 1 <= |x| <= 0.5   */
begin
        x1 = if (x<0) -> -x
           || x
           fi;
     xtemp =num2DIV(fix<(2*NDIV)+1,NDIV>(x1)>>10);
        q1 = num4DIV (divfix2( (num2DIV(0.000976563) - xtemp),xtemp));
     qf = if (x<0) -> -q1-1
         || q1+1
         fi;
end;
```

```
#define word     fix<25,14>
#define AR_ORDER8

func main (lpc,minverse : word[AR_ORDER + 1] ) cep : word[] =
begin
  /* The LPC coefficients are stored in lpc[1] through lpc[AR_ORDER]. */
  /* The constant gain, A, of the AR model is assumed for simplicity */
  /* to be 1, so that c[0] = ln(1) = 0. */


  /* minverse[ ] is the array of 1/m values for m = 1 through 8,     */
  /* i.e. 0,0, 0.5, 0.333333333, 0.25, 0.2, 0.166666666, 0.142857142, 0.125 */


  cep[0] = 0;
  cep[1] = -lpc[1];
  l1##1 = word(2);
  (l : 2 .. AR_ORDER) ::
    begin
        s[l][0] = word(0);
        /*
         * loop variable cannot be used as a signal right now
         * so j1 is needed simply to replicate j
         * and i1 is needed simply to replicate l
         * This needs duplicates multiplications unneccesarily
         * and may increase the power consumed.
         */
        j1##1 = word(1);
        (j : 1 .. l - 1) ::
          begin
            s[l][j] = s[l][j-1] + word(lpc[j]*cep[l-j]*word(l1#1-j1#1));
            j1 = j1#1 + word(1);
       end;

     s[l][l] = word(word(s[l][l-1]) * word(minverse[l-1]));
        cep[l] = -(lpc[l]+ s[l][l]);
        l1 = l1#1 + 1;
   end;
end;
```