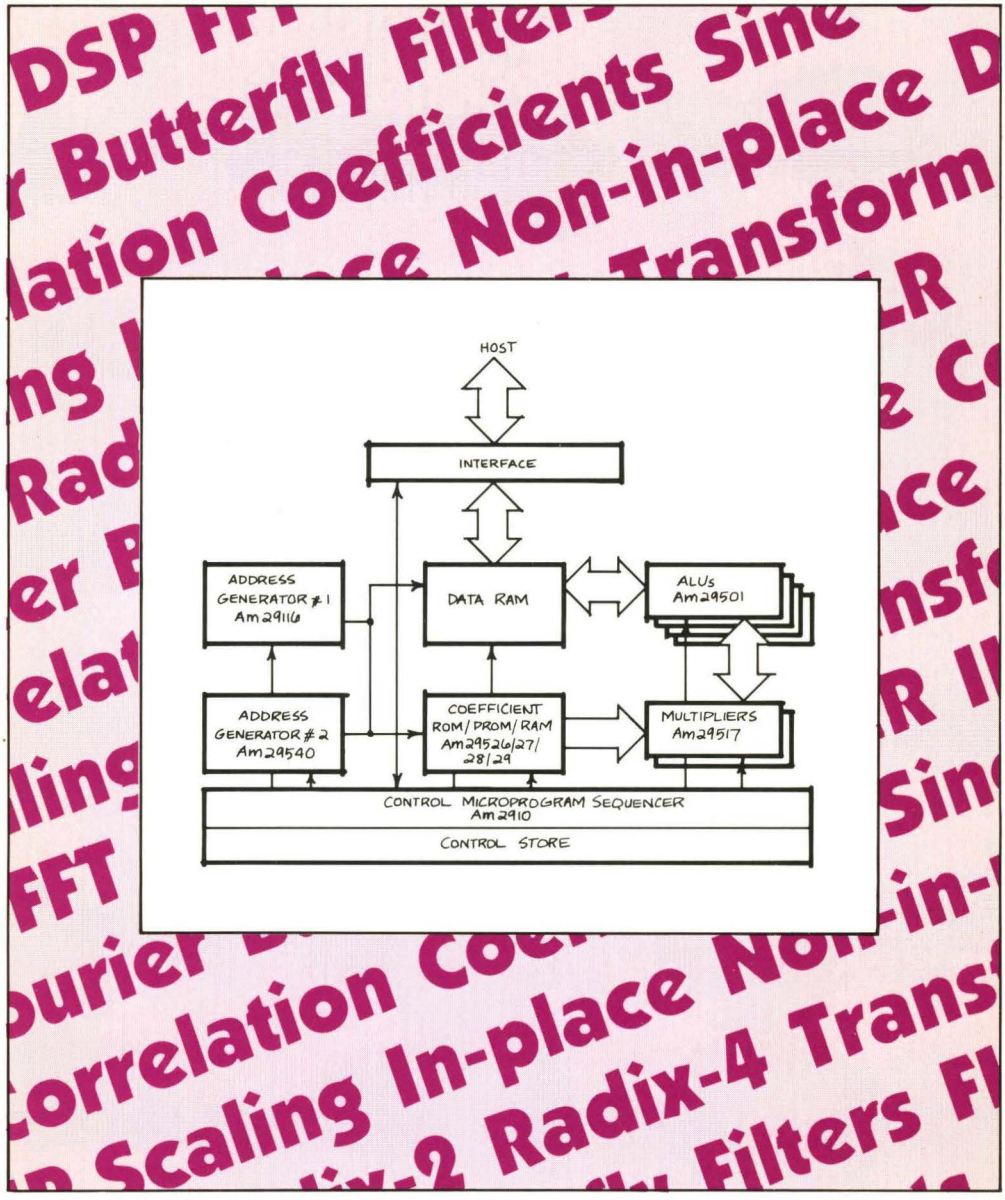
Am29500 Family Handbook

Array Processing and Digital Signal Processing

January 1986



ADVANCED MICRO DEVICES

Advanced Micro Devices

Am29500 Family Handbook

The International Standard of Quality guarantees a 0.05% AQL on all electrical parameters, AC and DC, over the entire operating range.

INT·STD·500

901 Thompson Place, P.O. Box 3453, Sunnyvale, California 94088
(408) 732-2400 TWX: 910-339-9280 TELEX: 34-6306

Table of Contents

CHAPTER 1
INTRODUCTION

What Is an Array Processor?

In recent years, Array Processing has become an increasingly significant aspect of computing. What once was a mysterious art, is now becoming common practice. Array processing is a form of computing that uses specialized hardware for special results—the array processor. This machine is characterized by its ability to handle many arithmetic computations at high speed. In other words, it is a "number cruncher." However, the specialization goes beyond a powerful hardware arithmetic unit. The machine invariably performs best when the data it is processing is structured in an array, such as a matrix or vector. Hence the name, Array Processor.

How Do Array Processors and General-Purpose Computers Differ?

Array processors and general purpose computers differ in a number of important aspects. The general purpose computer is usually of the classical Von Neumann architecture that was implemented by Mauchley and Eckert in the ENIAC machine, which became the first electronic Stored Program General Purpose Digital Computer. This machine had a memory area that allowed instructions and data to be intermixed. Conversely, array processors have separate memories for instructions and data (Harvard architecture). There may in fact be separate data memories for coefficients and variable data. The machines tend to be highly parallel, to allow for simultaneous multiplying and adding in the arithmetic section, while also performing address calculations for retrieving and storing the required data values.

What Is the Usual Architecture of Array Processors?

While there is no one standard architecture for all array processors, there are a number of characteristics that make the machine recognizable as an array processor. In addition to the above mentioned feature of separate data and instruction memories, pipelining is a common architectural attribute. This technique consists of placing intermediate registers in the data path, breaking up long combinatorial delay paths into shorter paths terminating in registers. The rate at which these registers can be clocked determines how fast the system runs. The register clock rate is obviously dependent upon the delay time of the operation preceding it. Pipelined systems have 'latency', which is the number of clock cycles that passes before the first valid result appears, but this value will generally be low compared to the large number of calculations performed.

Array processors also have dedicated hardware multipliers as part of their powerful arithmetic sections. Hardware floating point arithmetic may also be available, whereas some machines may have shifters to accomodate block floating point, and others may be dedicated to integer operation only. The basic operations that consume most of the processing time in matrix operations are multiplication and addition, as well as accessing the actual data values. Thus the architecture tends to be optimized for these operations.

Microprogrammed architectures tend to be popular in array processing machines. This is the familiar AMD 2900 family "bit-slice" structure that allows a machine to be constructed from "building blocks" of ALU, sequencer, memory access and other required functions, without any predetermined instruction set or architectural constraints that might be imposed by fixed-instruction devices. Microprogramming is the technique of giving a machine its instruction set by means of microinstructions stored in a high speed memory and accessed by a special sequencer. These microinstructions operate at the primitive level of register, bus, and ALU function control. Since the control store is a memory, machine behavior may be modified by changing one or more of the microinstruction bits or words. The technique leads to an extremely flexible, often very high speed, implementation.

Where Are Array Processors Used?

If array processors are indeed so powerful, then one might reasonably ask why they don't replace general purpose machines. While array processors do an excellent job of handling their specialized type of problem, they are rather clumsy when confronted with problems that require a lot of branching within the program, which is where the general purpose computer performs so well. So rather than replacing the general purpose computer with an array processor, the tendency is to create an enhanced machine by using an array processor as a peripheral, as an 'accelerator' to the arithmetic-intensive portions of problems.

Array processors have traditionally been implemented in peripheral fashion and as specialized Super Computers that had array processing type architectures. This latter class of machine tended to be extremely expensive, but very powerful. The peripheral array processor was

such a desirable approach that a number of companies make their entire livelihood from offering such devices. These peripheral array processors, or accelerators, generally attach to the bus or I/O structure of a scientific minicomputer. Now a tendency is evolving to offer array processors as options on engineering work stations, or even as option boards for personal computers.

Various types of problems lend themselves to array processing solutions, and so they are considered here. We wish to distinguish array processing by the type of hardware solution required, rather than the end application. Thus, robotics controllers, radar and sonar processors, flight simulators, graphics terminals, communications processors, medical analyzers, intelligent vision systems, and speech recognizers are all of interest here. If the problem is solved using multiply-and-add intensive algorithms, and the data and coefficients are structured in arrays, then we have an array processing application.

Digital Signal Processing

Digital signal processing is treated here as a subset of array processing because of the similarities that exist in the hardware and algorithms. The DSP engine tends to be more specific in its design and is generally imbedded in other hardware. DSP also tends to start out with a "live" analog signal, which is A/D converted, processed, and may or may not be converted back to analog. One could put an A/D converter in front of an array processor board and solve DSP problems.

What Is in this Book

We have attempted to collect in this book background and applications material that will motivate and guide the independent study of Array Processing. This book does not attempt to be a comprehensive text on the subject, but tends to emphasize the practical aspects of building array processor boards, implementing FFT's and FIR/IIR filters, selecting appropriate hardware, and writing microcode.

Finally, note that the devices whose data sheet summaries are included here are not only suitable for array processing applications. For example, for a fast eight-bit ALU with multiple I/O ports, the Am29501 is ideal. The part certainly won't be aware of what type of problem it is solving, so this should not limit the innovative spirit of the design engineer. For more information on the devices and systems described herein, contact your local AMD sales office, or AMD Headquarter Applications Department (408) 982-6266.

CHAPTER 2
NUMBER SYSTEMS

2.1 Fixed Point Numbers

A binary number is an ordered set of binary digits (bits), each of which has a value 0 or 1. Each bit, b_i, is assigned a binary weight, 2^i, and the value of the number is the sum of the weighted digits.

$$V = \sum_i b_i * 2^i$$

The simplest form of binary number is the unsigned integer. In an N-bit unsigned integer the index, i, ranges from 0 to N-1. The value is given by:

$$V_{integer} = \sum_{i=0}^{N-1} b_i * 2^i$$

The range of V is from 0 to 2^N-1. This type of number has two obvious limitations; it cannot represent quantities which are negative or fractional. There are many ways to represent negative numbers. The simplest method is to use an unsigned number to represent the magnitude, and a flag to indicate the sign. Not suprisingly this scheme is known as signed-magnitude. In an N-bit representation, the most significant bit, b_{N-1}, is taken for the flag, with a 0 signifying positive, and a 1 negative. This leaves N–1 bits for the magnitude, giving a range from $-2^{(N-1)}+1$ to $+2^{(N-1)}-1$. The value is given by:

$$V_{sign\ magnitude} = (-1)^{b_{N-1}} * \sum_{i=0}^{N-2} b_i * 2^i$$

An idiosyncrasy of signed-magnitude numbers is that there are two representations for zero, positive and negative. A similar scheme, which shares this characteristic, is one's complement. In one's complement, negative numbers are represented by inverting all bits of an unsigned number representing the magnitude. In order to distinguish positive and negative numbers, the magnitude range is restricted such that it can be expressed in N–1 bits. Thus the most significant bit is 0 for a positive number, and 1 for a negative number. One's complement numbers differ from signed-magnitude numbers only in that the magnitude bits are inverted in negative numbers.

Obviously, the range of numbers represented by a one's complement number must be the same as for signed-magnitude numbers. The value of a one's complement word may most easily be determined by treating it as an unsigned word, after having inverted all bits if the most significant bit is 1, in which case the value is negative. Alternatively, the value of the magnitude bits may be calculated, and if the most significant bit is 1, $2^{(N-1)}-1$ subtracted from this value.

$$V_{1's\ complement} = -b_{N-1} * 2^{N-1} - 1 + \sum_{i=0}^{N-2} b_i * 2^i$$

A simple technique by which negative numbers may be represented without double representation of zero, is to add to the desired value the magnitude of the most negative representable number. This gives a positive number which may be represented in the unsigned format. The value of the number may be obtained by simply reversing this process. This scheme is known as offset binary, or excess-M, where M is the number added. The number M is often, but not always a power of two.

$$V_{excess\ M} = -M + \sum_{i=0}^{N-1} b_i * 2^i$$

The special case, where $M = 2^{(N-1)}$, has the property that all negative numbers have a most significant bit which is 0, while zero and all postive numbers have a 1. Inverting this most significant bit leads to a scheme known as two's complement, which may be interpreted in several ways. The most significant bit is often treated as a sign flag, as it is in signed- magnitude. If the sign bit is 1, the number is negative and the following serial process is applied to convert it to an unsigned number representing its magnitude. Starting at the least significant bit, the bits are inspected in turn until the first 1 is encountered. This and all lesser significant bits are left unchanged. All more significant bits are inverted. This process operates in both directions, converting a negative number to a positive number of the same magnitude, and vice versa. It may be viewed as inverting the number (one's complement) and incrementing, or as subtracting the number from $2^{(N-1)}$.

$$V_{2's\ complement} = -\left(b_{N-1} * 2^{N-1} - \sum_{i=0}^{N-1} b_i * 2^i \right)$$

$$= -b_{N-1} * 2^{N-1} + \sum_{i=0}^{N-2} b_i * 2^i$$

Inspection of this formula shows that the magnitude of the sign bit's weighting is consistent with its position if the number were unsigned, but that the weighting is negative. This is an important conclusion, and leads to the most useful interpretation of two's complement numbers; they are identical to unsigned numbers except the most significant bit is weighted negatively. The range of values which can be represented by an N-bit two's complement number is $-2^{(N-1)}$ to $+2^{(N-1)}-1$.

Any of the above schemes may also be used to represent fractional numbers. This is achieved simply by adopting a convention that the weighting of the least significant bit is $2^{(-P)}$ rather than 2^0, and adjusting the other weights accordingly. Conceptually, this locates the binary point P bits from the least significant end. Because such a convention must be chosen in advance, and adhered to for all numbers, this is known as a fixed point number scheme. Other schemes, where the number contains a parameter locating the binary point, are known as floating point.

2.1.1 Fixed Point Operations

Three basic operators are described here: addition, subtraction and multiplication. Only unsigned and two's complement formats are described in detail. They are the two formats most commonly used in fixed point operation. Signed-magnitude and offset binary are commonly used in floating point (see below), and are usually treated in fixed point by converting them to unsigned or two's complement, performing the operation and reconverting. One's complement is not in general usage.

Addition of unsigned numbers is most easily performed by an iterative process known as ripple carry. The iterative block is shown in Figure 2-1.1. This has three inputs, which are equally weighted. Two of these are for operand bits, A_i and B_i, and the third is a carry input, C_{i-1}. The two outputs may be considered as a 2-bit word, representing the number of 1's present at the inputs. The unit weighted bit is the sum output, S_i, and the bit with weight two is the carry output, C_i. In cascade, the carry output of one cell becomes the carry input to the next more significant cell, maintaining the equal weighting in that cell, Figure 2-1.2.

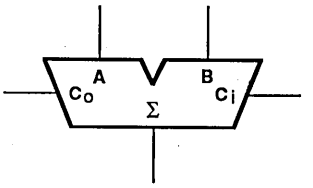


Figure 2-1.1

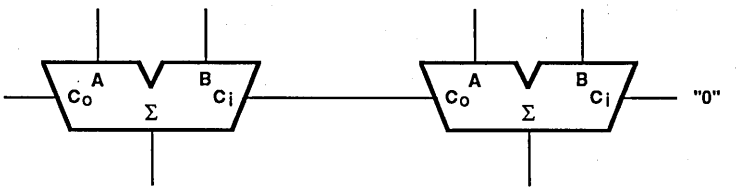


Figure 2-1.2

The carry input to the least significant cell, C_{-1}, is assumed to be 0. For the complete multi-bit adder, for each pair of input bits A_i and B_i, there is one output bit, S_i. This new number is the sum. The characteristic that each carry out must be generated in turn before higher bits can be determined leads to the name "ripple carry."

It is also possible to generate the carry inputs to each cell without waiting for the ripple. Each cell is capable of creating a carry into the next cell in two ways; it can generate a carry if both A_i and B_i are 1s or it can propagate a carry if either A_i or B_i is a 1 and the carry input C_{i-1} is a 1. This is called "lookahead carry" and can be expressed by the equation:

$$C_i = A_i * B_i + (A_i + B_i) * C_{i-1}$$

C_{i-1} can be expanded so that the equation for a carry lookahead of two cells is:

$$C_i = A_i * B_i + (A_i + B_i) * (A_{i-1} * B_{i-1} + (A_{i-1} + B_{i-1}) * C_{i-2})$$

This expansion can continue until the carry is expressed entirely in terms of the inputs and each cell can produce its output without waiting for output from any other cell.

As noted above, the range of an N-bit number is limited, and it is possible to overflow this range when adding two numbers. This may be overcome by making the adder one bit longer, thus doubling the range of the output. In order to provide the additional inputs, the operands are zero-extended. Inspection of the above formulae will show that the value of the unsigned words are not affected by leading 0's. Zero-extension is also used when adding numbers of dissimilar length. This is performed in an adder long enough to handle the larger input, the other being zero-extended. When the operand inputs to a cell are both 0, as in extension to prevent overflow, the sum output is equal to the carry input, and the carry output is identically 0. This fact may be exploited to save hardware, the carry line being used directly as the extra sum.

As the magnitude bits of a two's complement number are weighted the same as in an unsigned number, the same adder scheme may be used. In the most significant cell, the bits all have the same magnitude weighting, but while the carry remains positive, the operand inputs become negative. The cell is required to output a 2-bit two's complement number, that again represents the number of 1's present at the input, but taking into account their sign.

As in the unsigned case, the carry would represent the sign-bit of a one bit longer word, which allows for overflow. Appropriately, the sum bit of this cell would be positively weighted as a magnitude-bit of the new word.

Adding two's complement numbers of dissimilar length raises a question. The value of a two's complement number is not affected by adding leading 0's if the existing weights remain unchanged. However, this gives a number with a negatively weighted bit other than in the most significant position, which does not agree with the definition. This problem is overcome by using sign-extension, rather than zero-extension.

In sign-extension, additional bits are added which match the sign-bit. It must be stressed, however, that these are not extra sign-bits. A two's complement word can, by definition, only have one sign-bit; the negatively weighted most significant bit. When a new sign-bit is added, the old sign-bit reverts to a magnitude bit. If the number is positive, both old and new sign bits are 0, and the value cannot be changed by altering the weighting of 0's.

If the number is negative, the reversion of the old sign-bit to a magnitude bit changes the value positively by twice its weight (from -2^{N-1} to 2^{N-1}). This exactly cancels the contribution to the value from the new, more significant sign-bit, thus leaving the value unchanged. This proceedure may be repeated, making the number as long as necessary.

This process is reversible. If the most significant magnitude bit matches the sign-bit, then the number may be reduced in length by eliminating the sign-bit, the new most significant bit becoming the sign-bit. This fact is exploited to simplify the logic in the adder. If it is known that the sum cannot overflow into the N+1th bit, the sum output of the sign-bit adder cell can be taken as the sign-bit of the N-bit word. This is logically equivalent to the sum output of an unsigned adder cell, which may be used in place of the special cell. The carries, however, are not equivalent. If overflow is possible, it may be protected against by sign-extending both inputs one bit, and using an additional adder cell to give the N+1-bit output.

Two's complement subtraction is often described by modifying the adder, and re-interpreting the carry to become a borrow-not (analogous to decimal subtraction). This is confusing. A more straightforward explanation is that the number to be subtracted has its sign changed by being two's complemented, and the result added to the other

operand, using the adder as an adder. As described above, two's complementing a number involves one's complementing it and then incrementing it. The one's complement can be performed with inverters, and the incrementation effected by entering an extra 1 in the unused least significant carry input, Figure 2-1.3. If exclusive ORs are used in place of inverters to allow controlled inversion, and the least significant carry also controlled, an adder/subtractor is obtained.

Unsigned subtraction does not really exist. The unsigned operands are converted to two's complement by adding 0 (positive) sign-bits. These are subtracted as above, giving a signed result. If this is positive, the 0 sign-bit can be dropped, reverting to an unsigned format. If it is known in advance that this is case, then it is not necessary to calculate the redundant sign-bit. This is what is sometimes referred to as an unsigned subtractor.

When adding two unsigned numbers with the result expressed in the same number of bits as the inputs, overflow is signified by a carry out of the most significant bit. In unsigned subtraction, a negative result is signalled by the absence of a carry out. In all two's complement operations, overflow is signalled by the carry out of the sign-bit being different from the carry into it. This may be detected with an exclusive OR (XOR).

Unsigned multiplication consists of addition of the partial products formed by weighting the multiplicand by each bit of the multiplier. Weighting by a multiplier bit which is 0 results in a partial product that is zero and weighting by a bit which is a 1 is a left shift of the multiplicand by a number of places equal to the bit position of the multiplier bit. A simple algorithm to perform multiplication is a shift and add procedure in which the multiplicand is shifted one bit position at a time and added to the product based on the corresponding multiplier bit and using the rules for adding operands of unequal lengths as stated in the discussion of addition above. Multiplication of two's complement numbers is the same when the rules for two's complement are applied, i.e., when the partial products are sign-extended instead of zero-extended for the shifted addition and the partial product from the multiplier sign bit is weighted by $-b_{N-1}*2^{N-1}$ instead of $+b_{N-1}*2^{N-1}$.

Multiplication of unsigned fractions is identical to multiplication of unsigned integers. Placement of the binary point is a matter of interpretation and does not require alteration of the algorithm. However, multiplication of two's complement fractions results in a product with different bit weights than the operands as shown in Figure 2-1.4. To obtain the same bit weights in the product as the operands it is necessary to shift the product left one position as shown in Figure 2-1.5. A product which has the same format as the operands is obtained when the less significant product is truncated or causes rounding of the more significant part. One problem with this shift is the possibility of overflow when both multiplier and multiplicand are equal to –1.0. The product is +1.0 which cannot be represented in two's complement fractional notation.

A common technique used to speed up multiplication is the Booth algorithm which examines consecutive bits of the multiplier. Whenever the consecutive multiplier bits change from a 1 to a 0, the multiplicand is added to the product with the proper bit weighting. When the bits change from a 0 to a 1, the multiplicand is subtracted; no operation is necessary when the bits are the same. This algorithm is based on the identities 3 = 2+1 = 4–1, 7 = 4+2+1 = 8–1, etc. which allow two operations (one addition and one subtraction) to replace a potentially larger number of operations (two additions, three additions, etc.). The worst case is a multiplier with alternating 1s and 0s for which there is the same number of operations as the add and shift algorithm when the multiplier is all 1s, i.e., one operation per multiplier

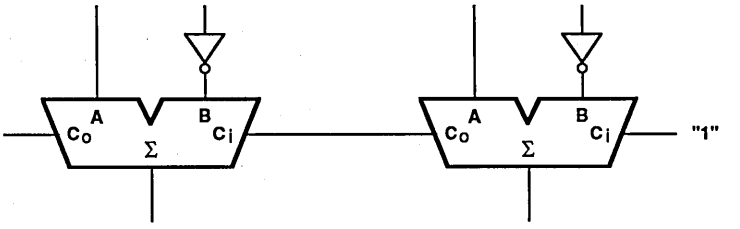


Figure 2-1.3

bit. A modification to the algorithm examines pairs of multiplier bits as shown in Figure 2-1.6 This modification is usually done to modularize hardware into a cell which can be repeated for each pair of multiplier bits.

2.2 What Is A Floating-Point Number?

The numbers one encounters everyday, such as 12, 34.56, 0.0789, etc., are known as fixed-point numbers, because the decimal point is in a fixed position. Such numbers are fairly closely matched in magnitude and within about ten orders of magnitude from unity. Examples of such numbers are found in bank accounts, unit prices of store items and paychecks.

In scientific notation applications, the numbers encountered can be very large. Avogadro's number expressed in fixed-point notation is approximately

602,250,000,000,000,000,000,000.

A scientist may also use Planck's constant which would be approximately

0.000000000000000000000000006626196

erg.sec. in fixed-point notation. These examples demonstrate the undesirability of writing fixed-point notation to represent numbers such as Avogadro's number and Planck's constant.

When a scientist writes the value of Avogadro's number, he writes 6.0225×10^{23}. Similarly, he would express Planck's constant as 6.626196×10^{-27} erg.sec.

The number 6.0225×10^{23} is thus observed to consist of four parts:

Sign. The sign of the number (+ or –). The plus sign is usually assumed when no sign is shown.

Mantissa. Sometimes also known as the fraction. The mantissa describes the actual number. In the example, the mantissa is 6.0225.

Exponent. Sometimes also known as the characteristic. The exponent describes the order of magnitude of the number. In the example, the exponent is 23.

Base. Sometimes also known as the radix. The base is the number base in which the exponent is raised. In the example, the base is 10.

The parts of a floating-point number can then be represented by the following equation:

$$F = (-1)^S \cdot M \cdot B^E$$

where

F = Floating-point number
S = Sign of the Floating-point number, so that S= 0 if the number is positive and S=1 if the number is negative
M = Mantissa of the floating-point number
B = Base of the floating-point number
E = Exponent of the floating-point number.

0 0 0	NOP
0 0 1	x1
0 1 0	x1
0 1 1	x2
1 0 0	–x2
1 0 1	–x1
1 1 0	–x1
1 1 1	NOP

Figure 2-1.6

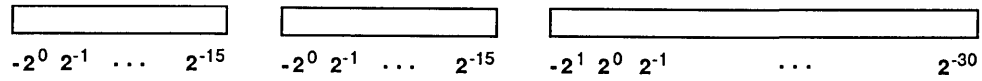


Figure 2-1.5

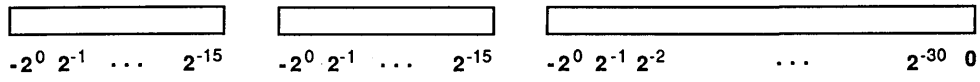


Figure 2-1.4

2.2.1 When Should Floating Point Be Used?

Although floating-point numbers are useful when numbers of very different magnitude are used, they should not be used indiscriminately. There is an inherent loss of accuracy and increased execution time for floating-point computation on most computers. Floating-point computation suffers the greatest loss of accuracy when two numbers of closely matched magnitude are subtracted from each other, or two numbers of opposite sign—but almost equal magnitude—are added together. Therefore, the Associative Law in arithmetic

$$A + (B + C) = (A + B) + C$$

does not always hold true if B is of opposite sign to A and C and very similar in magnitude to either A or C.

In most computers, hardware floating-point multiply and divide takes approximately the same amount of execution time as hardware fixed-point multiply and divide, but hardware floating-point add and subtract usually takes considerably more time than hardware fixed-point add and subtract. If the computer lacks floating-point hardware, all floating-point computations will consume more CPU time than fixed-point computations.

2.2.2 Floating-Point Formats

The following three number bases are commonly used in floating-point number systems:

1. Binary—The base is 2.
2. Binary Code Decimal—The base is 10.
3. Hexadecimal—The base is 16.

The two types of exponents used in floating-point number systems are the biased exponent and the unbiased exponent. An unbiased exponent is a two's complement number. An exponent said to be biased by N (or excess N notation) means that the coded exponent is formed by adding N to the actual exponent in two's complement form. Any overflow generated from the addition is ignored. The result becomes an unsigned number. Most common floating-point systems use a biased exponent because it simplfies floating-point hardware. During floating-point computation, arithmetic operations such as add and subtract need to be performed on the exponent of the operands. If a biased exponent is used, the arithmetic logic unit (ALU) needs only to perform unsigned arithmetic. If an unbiased exponent is used, the ALU must perform two's complement arithmetic; and overflow conditions are more difficult to detect.

Floating-point numbers must always be presented to the computer in "normalized" form (i.e., the most significant digit of the mantissa is always non-zero, except if the number is zero). For a binary floating-point system, this means that the leading binary bit of the mantissa is always 1 (except when the number is zero). In some floating-point number systems, this 1 bit is suppressed on input or output to the floating-point processor. The saved bit can be used for one more bit of precision or one more bit of exponent range.

The IEEE and DEC floating-point formats are covered in significant detail in AMD's data sheet.

CHAPTER 3
ARRAY PROCESSING ALGORITHMS

3.1 DIGITAL FILTERS IN THE TIME DOMAIN

Digital filters are devices which accept a sequence of digital data samples and produce a modified sequence which is a linear combination of previous inputs and outputs. The sequence is usually data which has been obtained from a continous analog waveform that has been digitized at uniform time intervals. Uniform sampling is used since it simplifies analysis of the system performance and does not create any undesirable side effects or limitations. The usual notation when dealing with the digitized sequences is to associate each sample with a polynomial term, e.g.,

$$\{X_n\} = \ldots a,b,c,d,e \ldots \text{ for } n=-2,-1,0,1,2$$
$$= \ldots az^{-2}+bz^{-1}+cz^{0}+dz^{1}+ez^{2}\ldots$$

This polynomial is the Z-transform of the original sequence and can also be written:

$$X(z)=\sum x_n z^{-n}$$

The relationship between the input and output of a digital filter is:

$$y_n=\sum x_m h_{n-m}$$

where $\{x_m\}$ and $\{y_n\}$ are the input and output sequences and $\{h_{n-m}\}$ is the impulse response of the filter. When $\{h_{n-m}\}$ has a finite number of non-zero terms, the filter has a Finite Impulse Response (FIR). Otherwise it is described as having an Infinite Impulse Response (IIR).

If we delay an input sequence by one sample period, the result is the same as the input sequence multiplied by z^{-1}. The combination of previous inputs and outputs of a digital filter can therefore be represented by:

$$y_n=\sum a_k x_{n-k}-\sum b_k y_{n-k}$$

where the first sum is a linear combination of the current and previous inputs of the sequence and the second sum is a combination of the previous outputs. A recursive filter of this type has an infinite impulse response. When the coefficients of the second sum are all zero, the filter becomes non-recursive and the impulse response have a finite number of terms (an FIR filter). A Z-transform of each sequence in the digital filter can be written:

EQ. 1

$$Y(z)=\sum y_n z^{-n}$$
$$=\sum\left[\sum a_k x_{n-k}-\sum b_k y_{n-k}\right] z^{-n}$$

Using the equality $X(n-k) = z^{-k}X(n)$ this can be rewritten and rearranged as:

EQ. 2

$$Y(z)=\sum a_k z^{-k} X(z)-\sum b_k z^{-k} Y(z)$$

The transfer function of the filter is therefore:

$$H(z)=\frac{Y(z)}{X(z)}=\frac{\sum a_k z^{-k}}{1+\sum b_k z^{-k}}$$

When working with linear time-invariant systems, filters can be cascaded (either serially or in parallel) to obtain a desired transfer function as shown in Figure 3-1.1. A second order filter has a transfer function of the form:

$$H(z)=\frac{a_0 z^{0}+a_1 z^{-1}+a_2 z^{-2}}{1+b_1 z^{-1}+b_2 z^{-2}}$$

$$=A\frac{z^2+Dz+E}{z^2+Bz+C}$$

This is a basic building block for digital filters and is called a biquadratic filter with a gain of A. By cascading a number of these sections and selecting the proper values for the coefficients, it is possible to produce any frequency response with the same structure, e.g., high-pass, low-pass, band-pass or band-stop filter. Computer programs are available to assist the filter designer in choosing the proper coefficients for control of filter skirts, ripple, etc. These programs also allow the designer to select which parameter to compromise when the desired performance is unrealisable.

Implementation of a digital filter is frequently a hardware embodiment of the equations expressed above. Adders and multipliers perform the arithmetic on data that have been appropriately delayed in registers. A filter section might use an adder for each addition and a multiplier for each multiplication or it might sequence data through a single unit. Cascading of sections could also take the form of multiple hardware sections or could be

a single section with programmable coefficients that operates iteratively on the data. A block diagram of a second order filter section is shown in Figure 3-1.2. If the number of terms in the sums is the same (as is shown in the figure), it is possible to rearrange the blocks to reduce the number of delay registers to a single set (sometimes called the canonical form). This is done by factoring the transfer function to separate the numerator and denominator:

$$H(z) = \sum a_k z^{-k} * \frac{1}{1 + \sum b_k z^{-k}}$$

The numerator is implemented by the hardware shown in Figure 3-1.3 and the denominator by that in Figure 3-1.4. When the two halves are cascaded, a single set of delay registers can be used as shown in Figure 3-1.5.

Implementation of an FIR filter is a simple sum-of-products as shown in the block diagram in Figure 3-1.6 Since the number of terms in an FIR filter is significantly larger than the number in an IIR filter with equivalent performance, most implementations use algorithms that reduce the computational requirements wherever possible by taking advantage of symmetry or by decimating the

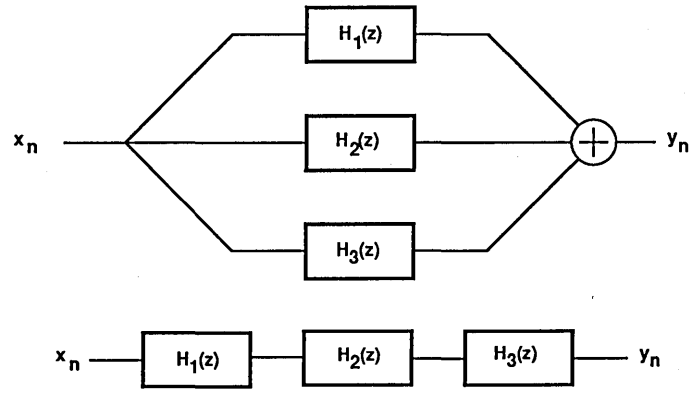


Figure 3-1.1

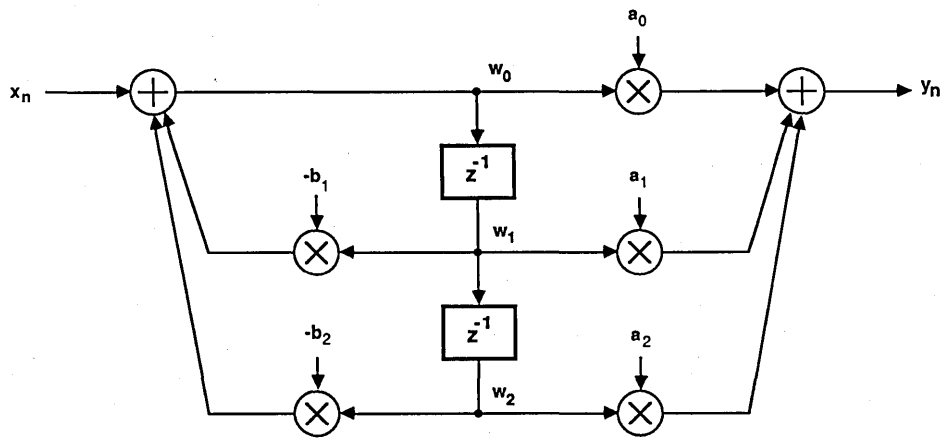


Figure 3-1.2

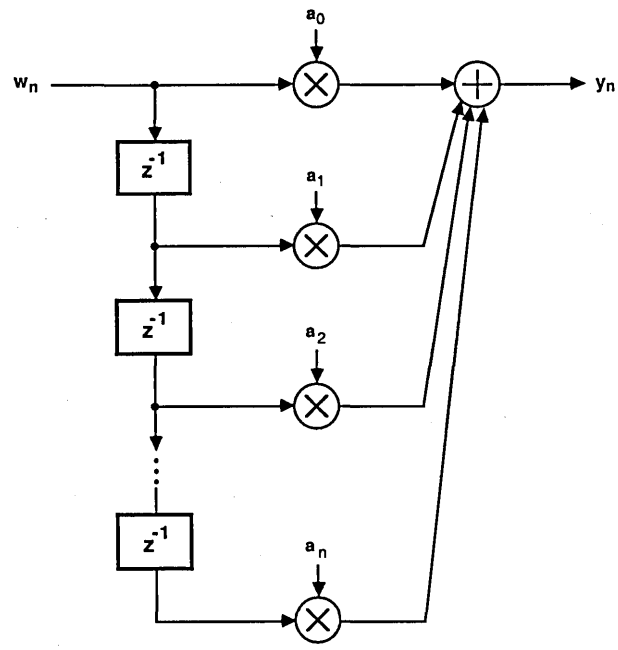


Figure 3-1.3

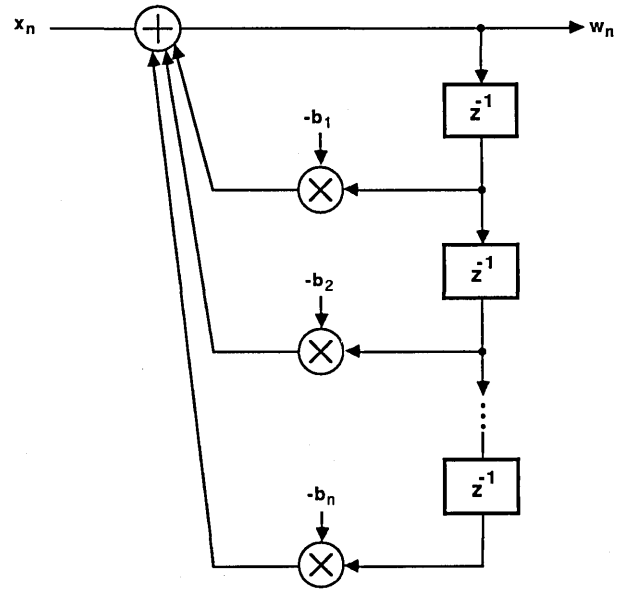


Figure 3-1.4

data. Symmetry trades a less costly computation (addition) for a more expensive one (multiplication). Since the coefficients for an FIR filter are usually symmetrical

$$(a_k = a_{n-k}),$$

it is possible to reduce the number of multiplications required by adding data with the same coefficient before multiplying by the common coefficient as shown in Figure 3-1.7.

Decimation is the process of reducing the sampling rate when the frequency response is limited. For example a signal containing only frequencies below 5 kHz and sampled at 20 kHz can be decimated by 2. Also, a band pass filter can be converted to a low-pass filter and the resulting data could then be decimated with a corresponding reduction in computation requirements. Decimation by N is accomplished by only processing every Nth data sample and discarding the remaining data samples. This reduces the computations of an FIR filter by a factor of N. Decimation does not reduce the number of computations for an IIR filter since the recursive nature of the filter requires that all the previous outputs be computed for the current output.

Digital filters are also used for interpolation. This is an increase in the sampling rate (the opposite of decimation) and is used to produce the data points between the actual samples. The interpolation procedure is to insert N–1 zeros between each data sample and pass the data through a low-pass filter. Since the data insertion is only conceptual (only calculations with non-zero data points are performed), an FIR filter has the same computational advantage over an IIR filter for this operation as for decimation.

Since both IIR and FIR filters can be designed for equivalent performance, it may not be obvious why the FIR filter with its greater number of computations would ever be used. One important characteristic of an FIR filter is that it can be designed to have linear phase response which is not realizable with either IIR or analog filters. A second advantage of an FIR filter is that roundoff errors are limited and easily controlled. The recursion in an IIR filter means that roundoff errors are cumulative.

Another advantage of the FIR implementation of a filter is the ability to partition the processing for the filter into data sets for multiple filter circuits and

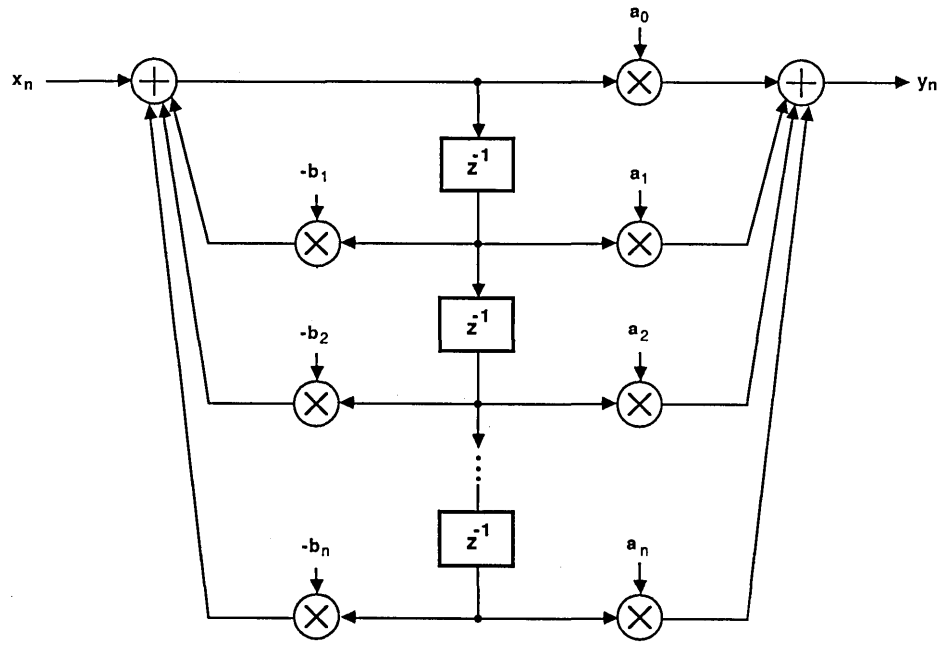


Figure 3-1.5

thus to increase the throughput of the system through parallel processing. This is not possible for IIR filters since the previous outputs are required for calculation of the current output. Although a cascaded IIR filter achieves a similar result, i.e., the processing can be distributed to different pieces of hardware operating in parallel, the complexity of the filter, or lack thereof, limits the potential gain. An FIR filter could use ten circuits to obtain a tenfold throughput gain or twenty circuits to obtain a throughput gain of twenty. An IIR filter might contain five second-order sections and is limited to a potential throughput increase of five.

Although stability of the FIR filter is sometimes cited as an advantage, it is not difficult to design a stable IIR filter, but the designer must pay attention to the warnings from the filter design program.

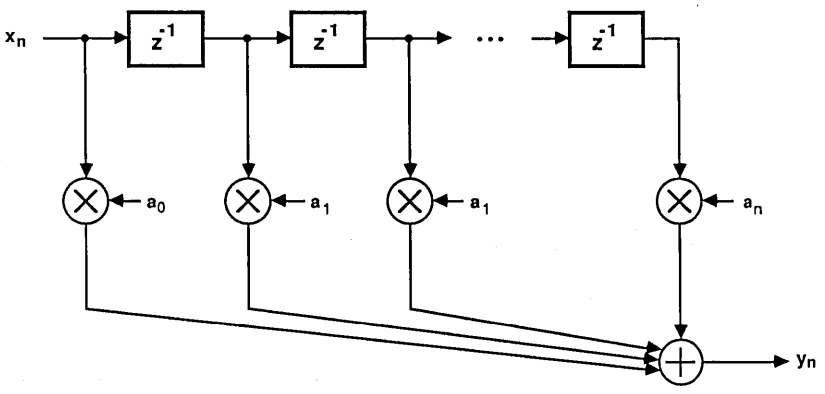


Figure 3-1.6

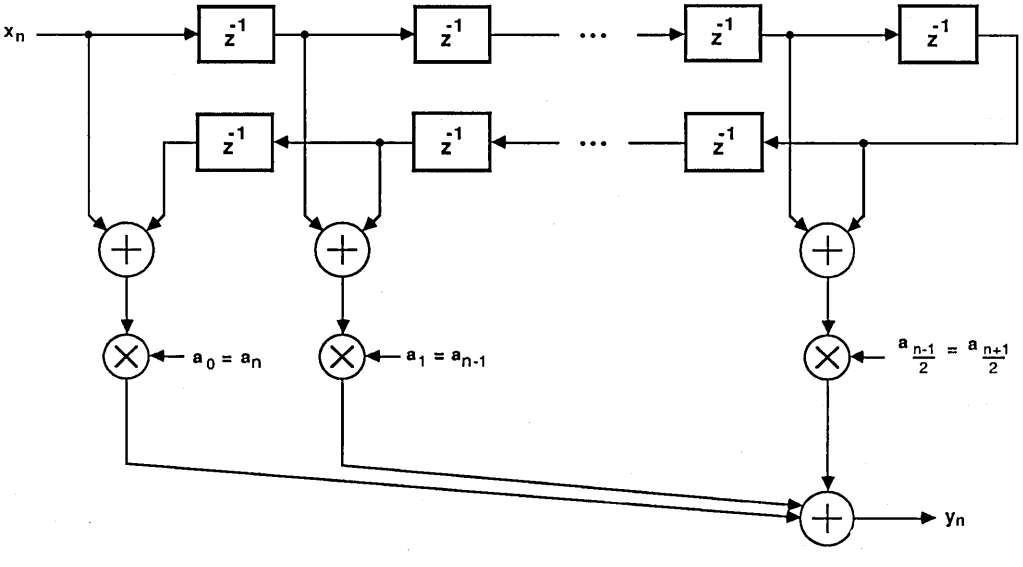


Figure 3-1.7

3.2 FILTERING IN THE FREQUENCY DOMAIN—FOURIER TRANSFORM

Fourier transforms are used to change time domain data into the frequency domain when the processing required involves the measurement of spectra. The processing in the frequency domain is so much simpler compared to that in the time domain to warrant the additional transformations between time and frequency domains. Fourier transforms are performed on a limited series, usually a power of 2 to take advantage of the efficiencies of the fast algorithm described below. The Discrete Fourier Transform (DFT) of a limited series $\{ x(n) \}$, $0 \le n \le N-1$ is defined by:

$$X(k) = \sum_{n=0}^{N-1} X(n) * e^{-j\,2\pi nk/N} \quad \text{for } k = 0, 1, \ldots, N-1$$

Although the transform can be calculated directly by performing the summation of all the products in the equation, the number of calculations becomes prohibitive as the transfform size increases. Defining $\{ x(n) \}$ as a series of N complex numbers, there are $(N-1)^2$ complex multiplications and $N(N-1)$ complex additions to be performed. The following table illustrates the rapid increase in processing requirements as N increases.

N	Multiplication	Addition
64	3969	4032
256	65025	64280
1024	1046529	104755

These numbers represent complex operations which translate to four real multiplies and two real adds for each complex multiply and two real adds for each complex addition. Fortunately, it is possible to reduce the number of calculations necessary for the transform by organizing the sequence in which the calculations are performed. Because of the periodic nature of $e^{(-j*2\pi*n*k/N)}$, product terms in the series appear in more than one summation and need not be recalculated for every occurrence. The Fast Fourier Transform (FFT) removes redundant calculations by repeatedly separating the initial series into two half series until the operation is reduced to a calculation with two data points. There are two different algorithms for doing this—decimation in time and decimation in frequency.

Since the equations defining the DFT and FFT do not contain time or frequency as a variable, the calculations can be applied equally well to data from audio sources to microwave frequencies. For example, if N =1024 data samples are taken at 1msec intervals, each frequency output will represent

$$k * 1\text{kHz}/1024$$

and if the sample interval is 1μsec, then each output will represent

$$k * 1\text{MHz}/1024.$$

The spectrum is also continuous so that the filter output 0 is adjacent to filter output 1023 and it is sometimes convenient to think of the filters as being positive and negative, i.e., filters 0–1023 become filters 0 to 511 and –512 to –1.

Although the FFT is more efficient when computing a full spectrum, some applications are only interested in a limited band of frequencies. In these cases the FFT algorithm calculates many outputs which are not of interest and discarded. In these applications, calculating a small number of DFTs may be more efficient than calculation of the entire spectrum with an FFT. Examples of this restricted band of interest are carrier detection and target tracking in radar or sonar.

3.2.1 Algorithm for Decimation in Time

Assuming that N is an integral power of two ($N = 2^r$), we can separate $\{ x(k) \}$ into two half series:one sequence with even indices

$$\{ x_E(n) \} = \{ x(2n) \}$$

one sequence with odd indices

$$\{ x_O(n) \} = \{ x(2n+1) \}.$$

The DFTs of the sequences constructed over

$\{ x_E(n) \}$ and $\{ x_O(n) \}$ are
$\{ X_E(k) \}$ and $\{ X_O(k) \}$ with

$$X_E(k) = \sum_{n=0}^{(N/2)-1} X(2n) * e^{-j\,2\pi nk/(n/2)}$$

$$X_O(k) = \sum_{n=0}^{(N/2)-1} X(2n+1) * e^{-j\,2\pi nk/(n/2)}$$

The notation is simpler if we use $W_N = e^{(-j*2\pi/N)}$ and the transform of the original series becomes the sum of the transform of the series of even indices and the transform of the series of odd indices multiplied by a twiddle factor.

$$X(k) = \sum_{n=0}^{(N/2)-1} X(2n) W_N^{2nk} + \sum_{n=0}^{(N/2)-1} X(2n+1)\, W_N^{(2n+1)\,k}$$

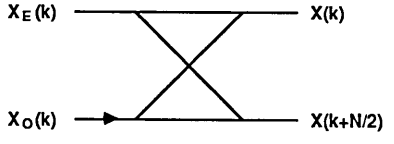
$$= X_E(k) = W_N^k * X_O(k)$$

Since $W_n^2 nk$ is periodic with a length of N/2 and that $W_n^k + N/2 = -W_n^k$ we can write

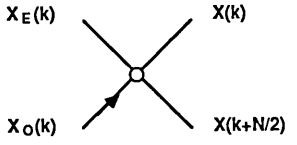
$$X(k+N/2) = X_E(k+N/2) + W_N^{k+N/2} * X_O(k+N/2)$$

$$= X_E(k) - W_N^k * X_O(k)$$

This calculates X(k) and X(k+N/2) in terms of $X_E(k)$ and $X_O(k)$ which is represented graphically by a simple butterfly:



This butterfly is usually simplified to:



The sequences

$\{x(2n)\}$ and $\{x(2n+1)\}$

can each be separated into odd and even again:

$\{x(2n)\}$ can be decomposed into
$\{x(4n)\}$ and $\{x(4n+2)\}$

$\{x(2n+1)\}$ can be decomposed into
$\{x(4n+1)\}$ and $\{x(4n+3)\}$

This results in X(k), X(k+N/4), X(k+N/2) and X(k+3N/4) being calculated from the transforms of each quarter series $X_{Q0}(k)$, $X_{Q1}(k)$, $X_{Q2}(k)$ and $X_{Q3}(k)$ in two stages as shown in the following equations:

$$X_E(k) = X_{Q0}(k) + W_N^{2k} * X_{Q2}(k)$$

$$X_O(k) = X_{Q0}(k)\ W_N^{2k} * X_{Q2}(k)$$

$$X_E(k+N/4) = X_{Q1}(k) + W_N^{2k} * X_{Q3}(k)$$

$$X_O(k+N/4) = X_{Q1}(k) - W_N^{2k} * X_{Q3}(k)$$

Because $N = 2^r$ we can continue decomposing "r" times until the summations in the butterfly calculation become the terms x(k). For N = 8 samples we obtain the results shown in Figure 3-2.1.

The input data in Figure 3-2.1 was ordered according to the needs of decomposition into odd and even series which has the property that the results occur in natural order. On the other hand, if the original data was in chronological order, the results would be in bit-reversed order. Bit-reversal refers to the binary representation of the sequential position of the data values. For a sequence $N = 2^r$ we read the binary code (r bits) of the initial position backwards and obtain the binary code of the bit-reversed position.

For N = 8 the values are:

Binary Value	Bit-Reversed Value
000	000
001	100
010	010
011	110
100	001
101	101
110	011
111	111

Note that the input data order when arranged for decimation in time is bit-reversed order so that the reordering process is the same for either input or output. Figure 3-2.2 shows the butterflies when the input data is ordered (and the results are bit-reversed) for N = 8 samples.

3.2.2 Algorithm for Decimation in Frequency

The decimation in frequency algorithm decomposes the initial series into two consecutive series.

$$X_1(n) = X(n)$$

$$X_2(n) = X(n+N/2) \qquad \text{for } n = 0, 1, \ldots, N/2 - 1$$

Using the preceding notation, the transform of the sequence can be written:

$$X(k) = \sum_{n=0}^{(N/2)-1} X(n) * W_N^{nk} + \sum_{n=N/2}^{n-1} X(n) * W_N^{nk}$$

$$= \sum_{n=0}^{(N/2)-1} X_1(n) * W_N^{nk} + \sum_{n=0}^{(N/2)-1} X_2(n) * W_N^{(n+N/2)k}$$

$$= \sum_{n=0}^{(N/2)-1} \left[X_1(n) + X_2(n) * W_N^{kN/2} \right] * W_N^{nk}$$

Since $W_n^kN/2$ is equal of -1 when k is odd and equal to $+1$ when k is even, the even and odd samples can be written:

$$X(2k) = \sum_{n=0}^{(N/2)-1} \left[X_1(n) + X_2(n) \right] * W_N^{2nk}$$

$$X(2k+1) = \sum_{n=0}^{(N/2)-1} \left[X_1(n) - X_2(n) \right] * W_N^{(2k+1)n}$$

Using $W_n^2n/k = e^{-j2\pi*2nk/N} = e^{-j2\pi*nk/N/2} = W_{N/2}^{nk}$ the equations become:

$$X(2k) = \sum_{n=0}^{(N/2)-1} \left[X_1(n) + X_2(n) \right] * W_{N/2}^{nk}$$

$$X(2k+1) = \sum_{n=0}^{(N/2)-1} \left[X_1(n) - X_2(n) \right] * W_N^n * W_{N/2}^{nk}$$

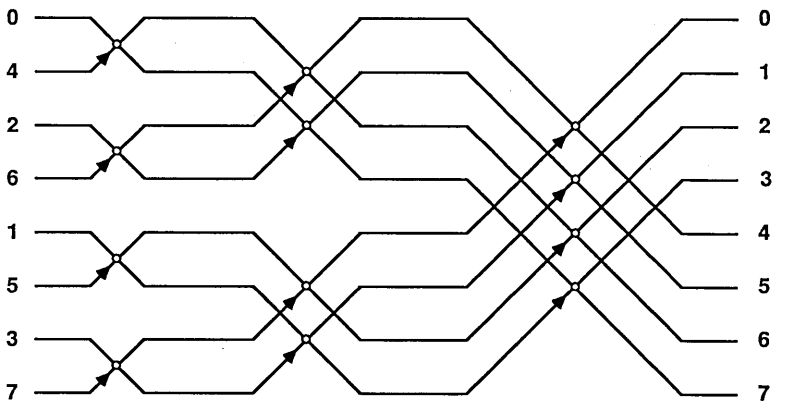


Figure 3-2.1

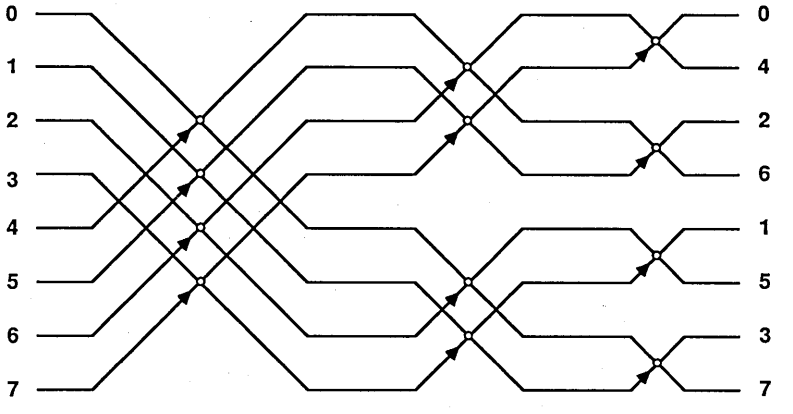


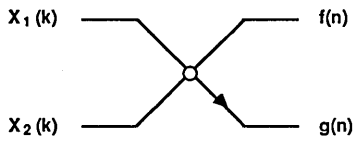
Figure 3-2.2

These equations show that the samples of the DFT can be obtained from the N/2 point DFTs of the sequences f(n) and g(n) where

$$f(n) = X_1(n) + X_2(n)$$

$$g(n) = \left[X_1(n) - X_2(n) \right] \cdot W_N^n$$

This can be represented by the following butterfly:



As in the decimation in time algorithm we can decompose each series into two sequences repeated "r" times. For N = 8 samples we get the graphic of Figure 3-2.3.

Data order is the same as for decimation in time; we have to order the initial values in bit-reversed order (see Figure 3-2.4 for N = 8) in order to get the results in natural order.

The decimation in time and decimation in frequency algorithms are very similar and call for the same number of arithmetic operations. The calculations can be made in-place (results replace the operands used to calculate them) but we must arrange the initial sequence in bit-reversed order to get the results in natural order or we can leave

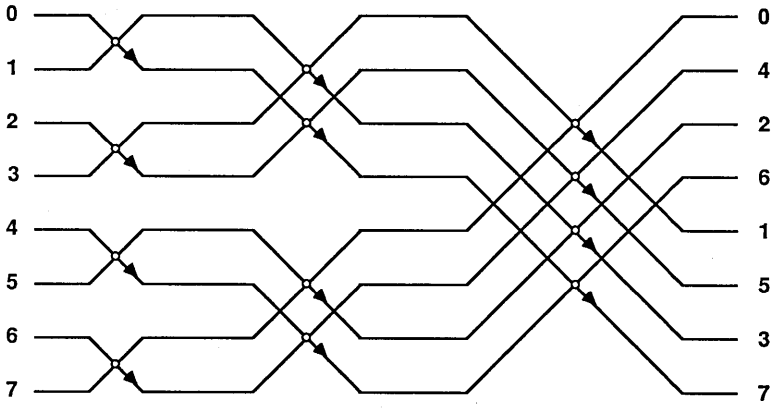


Figure 3-2.3

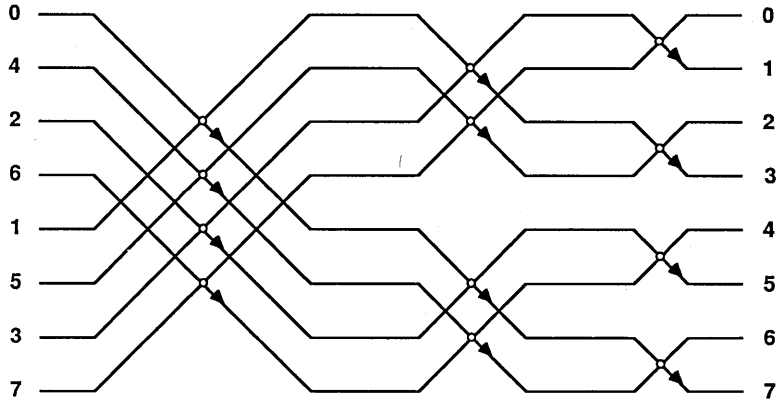


Figure 3-2.4

the initial sequence in chronological order and rearrange the resulting sequence. Reordering the inputs or the results are both performed according to the same rule, i.e., bit-reverse the binary representation of the position. It is also possible to avoid the rearrangement by using a double buffer memory and placing the butterfly results in the alternate buffer instead of in-place. The buffers would be swapped at the end of each FFT iteration to make the output of the previous iteration the current input data.

3.2.3 Comparison of FFT and DFT

The algorithms of the Fast Fourier Transform (FFT) requires $r = \log_2 N$ steps. At each step there are N complex additions and N/2 complex multiplications. Therefore, the total number of operations is:

$$N/2 \log_2 N \text{ multiplications}$$
$$N \log_2 N \text{ additions}$$

The following table compares these results to the ones of the Discrete Fourier Transform.

	Discrete Fourier Transform		Fast Fourier Transform	
N	(*)	(+)	(*)	(+)
64	3969	4032	192	384
256	65025	65280	1024	2048
1024	1046529	1047552	5120	10240

As mentioned previously, this efficiency only applies when a large portion of the spectrum is of interest. If only three frequency outputs were required, the DFT table becomes:

N	(*)	(+)
64	189	189
256	765	765
1024	3069	3069

3.2.4 Inverse Fourier Transform

The inverse Fourier transform of a signal transforms data from the frequency domain to the time domain and is calculated by:

$$X(n) = 1/N \sum_{n=0}^{N-1} X(k) * e^{+j\,2\pi\, nk/N}$$

The similarity to the forward transform allows us to use the same algorithms for the inverse transform by changing the sign of the angle from $(-2\pi*n*k/N)$ to $(2\pi*n*k/N)$ or W_N^{nk} to W_N^{-nk} The scale factor 1/N is typically incorporated in the gain of the system and ignored for the purposes of calculation of the transform.

3.2.5 Radix 4 FFT

It is possible to further reduce the number of calculations required to perform the FFT by using a radix 4 algorithm. If the number of samples N is represented by $N = 4^x$, for x = a positive integer, the butterflies of the preceding algorithms can be rearranged in groups of four. For the decimation in time algorithm with N = 4 we have to perform the operations shown by Figure 3-2.5. Figure 3-2.6 shows the same thing for N = 16. Figures 3-2.7 and 3-2.8 show the requirements for reordering the data for N = 16 but this time using the algorithms of decimation in frequency. The improvement in efficiency is shown in the following table:

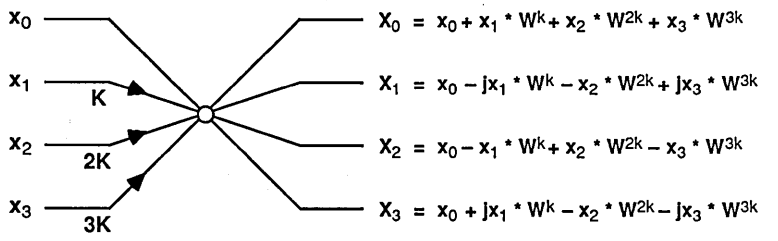


Figure 3-2.5

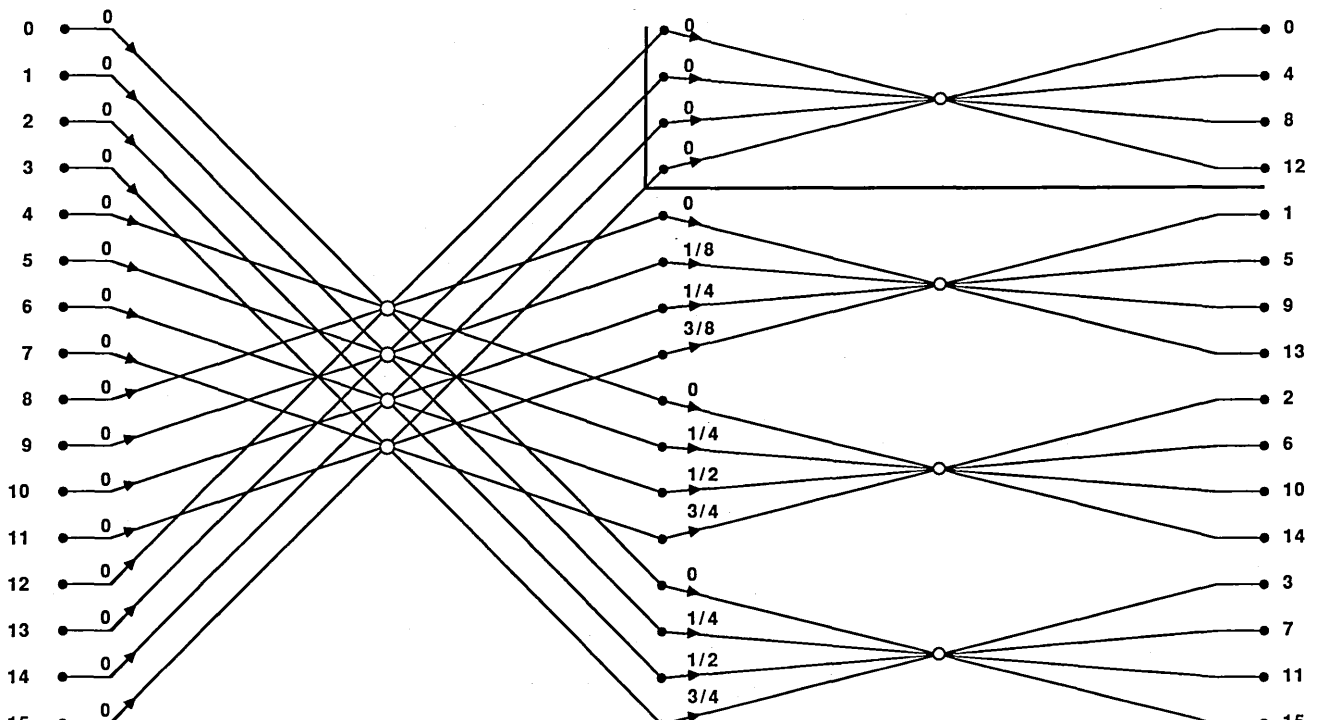

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
1/8
1/4
3/8
0
1/4
1/2
3/4
0
1/4
1/2
3/4
0
4
8
12
1
5
9
13
2
6
10
14
3
7
11
15

Figure 3-2.6

	Radix 2		Radix 4	
N	(*)	(+)	(*)	(+)
64	192	384	144	384
256	1024	2048	768	2048
1024	5120	10240	3840	10240

As shown in the table the advantage of the radix 4 algorithm is in reducing the number of multiplications by 25%.

3.2.6 Real-valued Input Fourier Transforms

Although the FFT processes complex data, the input to a system is frequently real data only. One way to process real-valued input data is to zero-fill the imaginary data and perform the FFT as though the inputs were complex. However, this is wasteful of memory and processing time since half the outputs are discarded when the FFT is complete. One way to overcome this inefficiency is to combine two series of real inputs and process them as a complex number series. This produces two spectra in the processing time required for a single zero-filled data set. However, it does require an additional processing step to separate the outputs into two spectra. If

$$\{x_1(n)\}$$

is treated as the real part of the data and

$$\{x_2(n)\}$$

is treated as the imaginary part, the two spectra can be separated by calculating:

$$X_1(k) = 1/2 \left\{ \left[X_R(k) + X_R(N-k) \right] + j \left[X_I(k) - X_I(N-k) \right] \right\}$$

$$X_2(k) = 1/2 \left\{ \left[X_I(k) + X_I(N-k) \right] + j \left[X_R(k) - X_R(N-k) \right] \right\}$$

This additional processing step is much less than the time that would be required to perform a second FFT.

A second algorithm takes a single series and treats every even data point as the real part and every odd data point as the imaginary part of the data. After performing an N/2-point FFT, the outputs must be calculated using the following:

$$X(k) = 1/2 \left[X_R(k) + X_R(n-k) + \cos\theta \, (X_I(k) + X_I(N-k)) - \sin\theta \, (X_R(k) - X_R(N-k)) \right] + j/2 \left[X_I(k) + X_I(n-k) - \sin\theta \, (X_I(k) + X_I(N-k)) + \cos\theta \, (X_R(k) - X_R(N-k)) \right]$$

$$\text{for } \theta = \pi k/N$$

In this case the additional processing is traded off against the time to perform an N-point FFT instead of an N/2-point FFT. The additional processing is more than the previous method but requires less memory since only one data set is processed instead of two data sets simultaneously. The data flow through the system is also more regular, and that is sometimes important.

A third algorithm developed by Bergland eliminates the calculations which produce redundant results. This method does not use the complex FFT algorithm; it treats the data as real or imaginary numbers during the calculations instead of operating on complex pairs. It also contains two butterfly configurations instead of the single butterfly used in complex algorithms. These butterflies are diagrammed in Figure 3-2.9. Butterfly type one is used when k = 0 and butterfly type two is used for all other coefficients. This algorithm also produces outputs in scrambled order as shown in Table 3-1 with the real and imaginary parts in consecutive positions in the sequence. The first two elements are the exception. These are the real parts of the first and

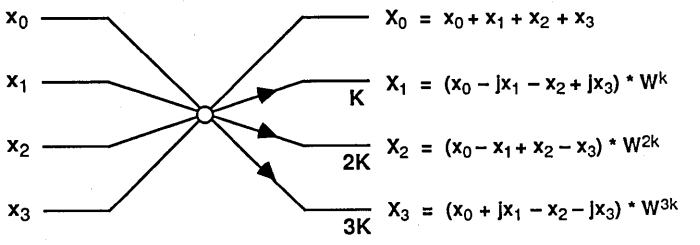


Figure 3-2.7

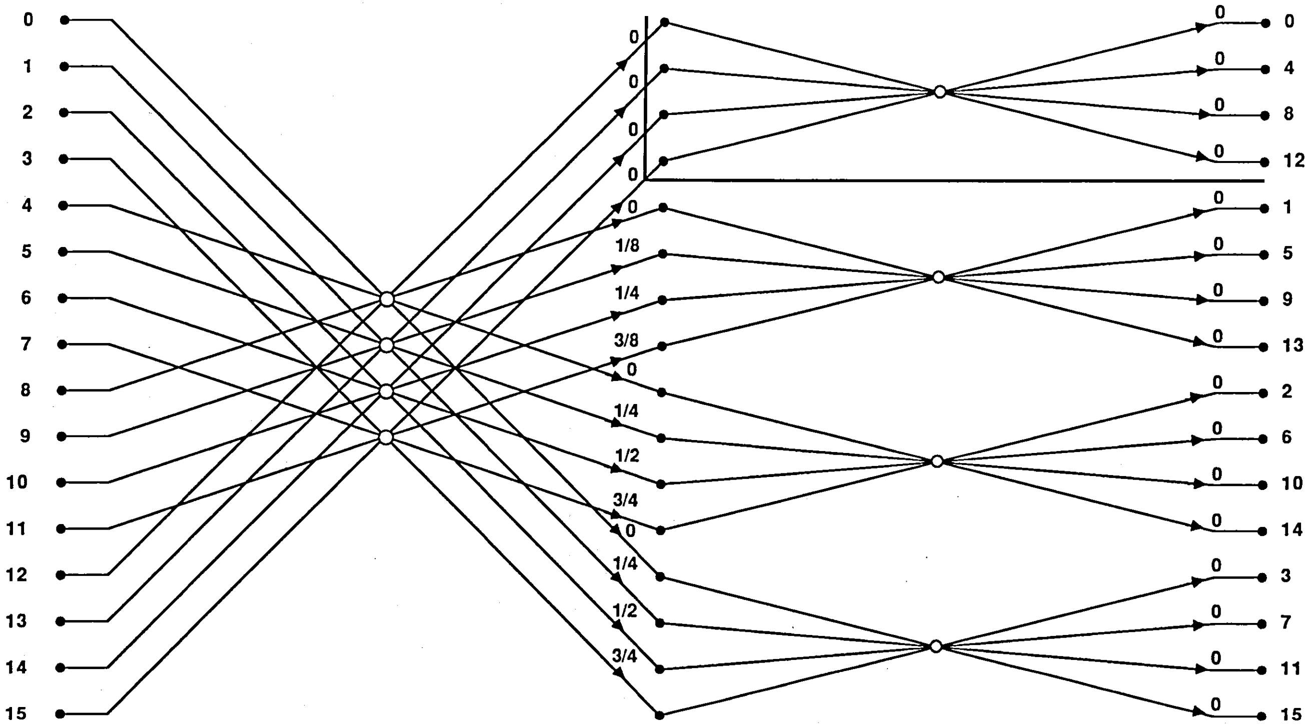

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
0
0
0
0
0
1/8
1/4
3/8
0
1/4
1/2
3/4
0
1/4
1/2
3/4
0
4
8
12
1
5
9
13
2
6
10
14
3
7
11
15

Figure 3-2.8

last frequency filters, which have imaginary parts equal to zero. Although the order appears similar to bit-reversed order, it does not have a simple translation and must be generated algorithmically from the previous sequence. The new sequence algorithm is as follows:

1) Even elements of the new sequence (starting with element 0) are the previous sequence.

2) Odd elements are inserted by subtracting the even element from the length of the new sequence except for element 1 which is equal to 1/2 the length of the sequence.

3.3 MAGNITUDE CALCULATION

An operation that is frequently necessary in signal processing is the computation of the magnitude of complex samples. This is particularly common when performing FFT processing. A magnitude can be calculated directly as the square root of the sum of the squares of the quadrature components. However this algorithm has the disadvantage of requiring double precision arithmetic for the calculations as well as being complicated and time consuming. A machine performing 16-bit operations would require 32-bit arithmetic to sum the products and the square root computation typically requires one iteration per bit in the result.

Since the computation of the square root of the sum of the squares is such a complex task, many alternatives to the magnitude calculation have been developed. The simplest approach to the calculation is to use a linear combination of the quadrature components. This takes the form M = ax+by for x = max(|R|,|I|) and y = min(|R|,|I|), i.e. x = abs(larger) and y = abs(smaller). Systems in which a shift is less expensive than a multiply (in time, hardware or both) typically use coefficients like a =1 and b =1/2. However a system which calculates a product as easily as a shift can obtain a more accurate magnitude with a = 0.960 and b = 0.398. Even more accuracy can be obtained by changing the coefficients as a function of the angle. Although the magnitude is independent of angle, the approximation error has an angle dependency equal to $1-(a*\cos(\Theta)+b*\sin(\Theta))$ for $0<\Theta<\pi/4$, which is the region defined by the larger/smaller equation stated above. The angle dependency of the algorithm can be kept simple by defining only two regions separated at an angle where the tangent is a simple ratio such as where one quadrature component is twice the other. An example of this approximation is to use a = 1 and b = 1/4 when x>2y and a = 3/4 and b = 3/4 for x<2y.

Another approach to calculation of the magnitude is to use Cartesian-to-polar coordinate conversion

Table 3-1

N =	4	8	16	32
	0_R	0_R	0_R	0_R
				16_R
			8_R	8_R
				8_I
		4_R	4_R	4_R
			4_I	4_I
				12_R
				12_I
	2_R	2_R	2_R	2_R
		2_I	2_I	2_I
				14_R
				14_I
			6_R	6_R
			6_I	6_I
				10_R
				10_I
	1_R	1_R	1_R	1_R
	1_I	1_I	1_I	1_I
				15_R
				15_I
			7_R	7_R
			7_I	7_I
				9_R
				9_I
		3_R	3_R	3_R
		3_I	3_I	3_I
				13_R
				13_I
			5_R	5_R
			5_I	5_I
				11_R
				11_I

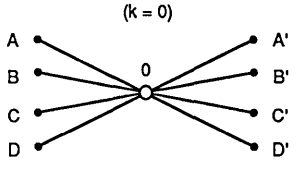
algorithms. The quadrature components are a vector in Cartesian coordinates and conversion to a magnitude-phase pair gives the desired result. A direct calculation of M = y/sin(arctan(y/x)) involves divisions and arctangents which are not attractive alternatives to a square root calculation. However, once the phase is determined, multiplication can be used to rotate the vector onto the x-axis where the value of the real component and magnitude are identical. This reduces the problem to calculation of the arctangent usually by means of a lookup table. Division can be eliminated by concatenating the two components of the vector to form the address of the table, but this double length address results in a prohibitively large table. The simplest approximation is to normalize the

vector by left-justifying the larger component, and to use a limited number of MSBs of each component to address the table, eg. using bits x_{15-10} and y_{15-10} of 16-bit numbers to address a table containing 4096 angles. A more accurate approximation which also removes the necessity for normalization uses the most significant bits of the larger component, the most significant bits of the smaller component and the bit position of the MSBs of each component (or the number of leading zeros). This might use two data bits and a four bit position for each component of the vector to address a table of 4096, e.g., bits x_{12-11} and position 12 for one component and bits y_{8-7} and position 8 for the other (assuming bits x_{15-12} are all the same and y_{15-8} are also the same). Because the arctangent is well behaved for small angles, this algorithm can be iterated for greater accuracy. The previous one can not be iterated since bits y_{15-10} become nonsignificant as the vector is rotated toward the x-axis. Addressing a table with the number of leading zeros of each component and the single bit following the leading one can produce a magnitude with 1.4% accuracy. Additional accuracy can be obtained by using more bits following the leading one or by iteration. Table size can also be traded off by using the difference between the number of leading zeros of each component to address the table rather than the individual counts. Both of the approximations could produce the sine and cosine directly from the table but the angle is sufficient when a sine and cosine table is already available for FFTs.

Another approximation which rotates the vector onto the x-axis can be used if multiplication is too costly. CORDIC (COordinate Rotation DIgital Computer) rotation is one of a class of algorithms developed by Volder in 1956. The key elements of this algorithm are successive approximation and elimination of multiplication. The vector is placed in the first quadrant by taking the absolute values of the components and is then rotated toward the x-axis by adding or subtracting a series of angles. Each angle is added if the phase of the vector is negative or subtracted if the phase is positive with the result that the x component of the vector becomes the magnitude as the phase angle approaches zero. Positive or negative phase is the same as a positive or negative y value which makes the determination of this condition straightforward. The angles are selected so that their tangents are simple binary ratios which allows rotation to take the form

$$x_n = x_{n-1} + y_{n-1}/2^r \text{ and}$$
$$y_n = y_{n-1} + x_{n-1}/2$$

where r increments for each iteration. Accuracy of the algorithm is a function of the number of iterations with four iterations producing an accuracy of 0.772%. A side effect of the algorithm is that there is gain, i.e. after four iterations the x component will be 1.643 times the magnitude of the original vector. However the gain is fixed for a given number of iterations and is usually absorbed in the gain of the system.



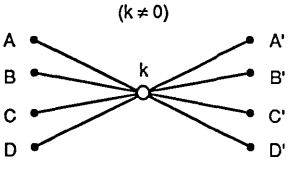
$A' = \text{Re}\,[A + jB + (C + jD)W_N^0]$

$B' = \text{Im}\,[A + jB + (C + jD)W_N^0]$

$C' = \text{Re}\,[A + jB - (C + jD)W_N^0]$

$D' = \text{Im}\,[-A - jB + (C + jD)W_N^0]$

$W_N = e\,(-j2\pi/N)$



$A' = \text{Re}\,[A + jC + (B + jD)W_N^k]$

$B' = \text{Im}\,[A + jC + (B + jD)W_N^k]$

$C' = \text{Re}\,[A + jC - (B + jD)W_N^k]$

$D' = \text{Im}\,[-A - jC + (B + jD)W_N^k]$

$W_N = e\,(-j2\pi/N)$

Figure 3-2.9

A common use of the magnitude calculation is for the detection of signals in noise such as radar target detection or communications carrier detection. The major concern in this application is the signal to noise ratio loss caused by the approximation. An average background value (representing noise) is calculated using an approximation with minimum average error and the signal magnitude is calculated for minimum peak error. Using the simple shifting approximations would mean using larger +1/4 smaller for the average (absolute error = +3, −11%, average error = 0.6%) and larger +1/2 smaller for the signal (maximum absolute error = +11, −0%, average error = 8%). Table 3-2 contains various coefficients and the errors associated with them.

Table 3-2

Approximation	Peak Error	Average Error
L + .5S	11.80	8.68
L + .25S	11.60	0.65
.960L + .398S	3.96	−1.30
L + .25S / .75L + .75S	6.06	−3.01

CHAPTER 4
SYSTEM DESIGN

4.1 ARRAY PROCESSOR DESIGN BASED ON THE Am29500 FAMILY

The board described in this chapter was built and tested in AMD Headquarters Applications in early 1985. This application was developed to show designers the efficiency in using the Am29500 Family for digital signal processing. Doing a paper design is one thing; building a working prototype boosts confidence level. This section of the manual will help the reader to design a system and point out what aspects of the design require consideration.

The goal here is to design an optimum cost/performance board. Table 4-1 compares radix-2 butterfly cycle times for different architectures. The architecture chosen for this design is the 2,2,1 architecture: two buses, two ALUs, and one multiplier. With this architecture, each butterfly takes four cycles. Therefore, 5,120 butterflies in a 1K complex point FFT at 10 MHz takes 2 ms. Figure 4-1.1 shows the basic architecture of the board. The two ALUs are 16 bits each for real and imaginary data. Each ALU is comprised of two Am29501s. The two ALUs can also be combined by using the Am2902 Carry Lookahead Generator to form a single, 32-bit, double-precision ALU. The two 16-bit buses supply real and imaginary data from the data memories to the ALUs, and vice versa, via the bidirectional DIO ports of the Am29501s. Data from the ALUs is transferred to the multiplier via the bidirectional MIO ports of the Am29501s. If 32-bit products are required, 16 of the 32 bits could come in from the multiplier into the MIO ports. The 16X16 multiplier is the Am29517. Its X and Y ports are the two 16-bit input ports, and its P port is the 16-bit output port on which the 32-bit product can be multiplexed. The Y port of the multiplier is a bidirectional port and, for single cycle 32-bit multiplies, this port has to be used in conjunction with the P port to get the complete 32-bit product.

The MIO ports of either the real or imaginary ALUs can be directed to the Y port of the multiplier. The X port data can be selected from one of several sources—the MIO ports of either the real or imaginary ALUs, the real data bus, the imaginary data bus, or the coefficient PROMs. The 1-of-4 MUX provides the path required for data flow to the X port. The P port of the multiplier goes to the MI ports of the ALUs.

The data memory is made up of high-speed (45 ns) RAMs. There are two memory banks. While one is being worked on by the DSP algorithm, the host CPU can unload and reload the other. Each memory bank is 1K deep and 32 bits wide, which means that the maximum FFT size can be 1K complex. Data from the host can be DMA'd over to the board or the CPU can use I/O to transport it. DMA is only provided on the 16-bit host buses. Eight-bit host buses must use programmed I/O. Thirty-two I/O addresses are reserved for this board and the address decode logic decodes these addresses and selects appropriate logic on the board.

The Am29540 and the Am29116 generate addresses for DSP algorithms. The Am29540 is an FFT address generator. The transform length, FFT type, etc. are supplied by the CPU to the part, and

Table 4-1 "Optimum" Cost/Performance (Radix-2 FFT)

For Each Butterfly	Memory Access 8		Add/Subtract 6		Multiply 4	
Resources	**Memory Buses**		**ALUs**		**Multipliers**	
# of Cycles	**#**	**Usage**	**#**	**Usage**	**#**	**Usage**
8	1	8/8	1	6/8	1	4/8
6	2	4/6	1	6/6	1	4/6
4	2	4/4	2	3/4	1	4/4
3	4	2/3	2	3/3	2	2/3
2	4	2/2	4	1.5/2	2	2/2

the part puts out the correct sequence of data and coefficient addresses. The Am29116 provides addresses for the filter and matrix algorithms.

The Am29520 is the address pipeline register for the data memory. The Am29821 is the address pipeline register for the coefficient PROMs for the FFT and filter algorithms.

The last block in the architecture is the microprogram control unit. The microword width is 128 bits. The code can be up to 2K deep. High-speed (35 ns) registered PROMs are used to store the code. The sequencer is the Am2910A. Two Am2922s allow the sequencer to test up to 16 different conditions.

The detailed architecture of the board is explained in the following five sections.

a) Arithmetic
b) Memory
c) Addressing
d) Control
e) I/O

4.1.1 Arithmetic

Figure 4-1.2 shows a detailed diagram of the arithmetic section.

Data scaling must be considered when designing an FFT board. This is necessary in fixed point systems, to ensure that the results do not overflow. The approach taken here is "block scaling," wherein all data is scaled by a certain amount at each pass. This kind of scaling can be done at the input to or the output from the ALU. A shifter at the input to each ALU serves the purpose without restricting the input data. When the shifter is at the output, the input data is restricted so that there is no overflow on the first pass through the pipeline. Overflow can occur during complex mutiplication when summing R*R–I*I or R*I+I*R because of inaccuracies due to truncation. Overflow could also occur when adding or subtracting, so two bits of overflow should be allowed when performing an FFT butterfly. Scaling involves right shifting of the data. Being able to shift up to 3 places is sufficient and thus 4-bit shifters (Am25S10), not barrel shifters, are used. Figure 4-1.3 shows how four Am25S10s are connected to form a 16-bit shifter for each ALU. They are connected so that when shifting a 2's complement number, the sign bit gets copied into the shifted position. The real and imaginary buses bring data into the DIO ports of the ALUs via these shifters. Two microcode bits control the amount of shift. The return paths from the DIO ports of the ALUs to the memories are buffered by the Am29827 10-bit buffers. The Output Enables of the shifters and buffers are controlled by the Write Enable lines from the microword to the data memories to ensure that bus contention does not occur.

The FFT algorithm uses the formula A±WB where A and B are the complex data points and W is the complex coefficient. Multiplication is only between B and W. So for the FFT algorithm, it is sufficient to have the coefficient as one input to the multiplier and the ALUs as the other input. Squaring to form a magnitude, on the other hand, requires multiplication between "data" inputs. No coefficients are involved in matrix multiplication. Thus, for this algorithm, both multiplier inputs must connect to the ALUs. Sum-of-product-type calculations may require that data flow directly from the memory into the multiplier, bypassing the ALU. Finally, complex arithmetic requires that data flow from either ALU to the other. All of these data paths exist in this design. The output of a 16-bit, 1-of-4 MUX, made up of eight 74LS253s, is connected to the X input of the multiplier. The four possible X inputs are: the coefficient PROMs, the real data bus from memory, the imaginary data bus from memory, and either ALU. A transceiver between the two ALUs, appropriately controlled, prevents bus contention and allows either ALU to be connected to either multiplier input. The transceiver also allows bidirectional data flow from one ALU MIO port to the other. The MUX and transceiver are controlled from microcode. The Y input of the multiplier is a bidirectional port. If a 32-bit product is required, 16 of the product bits can be multiplexed on this port and input into the ALUs via their corresponding MIO ports. The other 16 bits of the 32-bit product can go from the P port of the multiplier to the MI ports of the ALUs.

Some algorithms may require double-precision arithmetic. The Am2902 is a Carry Lookahead Generator that allows cascading of all four Am29501s to form a 32-bit ALU. Figure 4-1.4 shows how the ALUs and lookahead generator are interconnected to allow this. Here the Most Significant Byte for double-precision is the Imaginary MSB and the Least Significant Byte is the Real LSB. The RALU0M bit from microword controls whether the configuration is single-precision or double-precision. When this bit is High, the propagate and generate signals from the imaginary MSB are forced High, thus forcing C_{N+Y} from the Am2902 Low and effectively enabling single-precision.

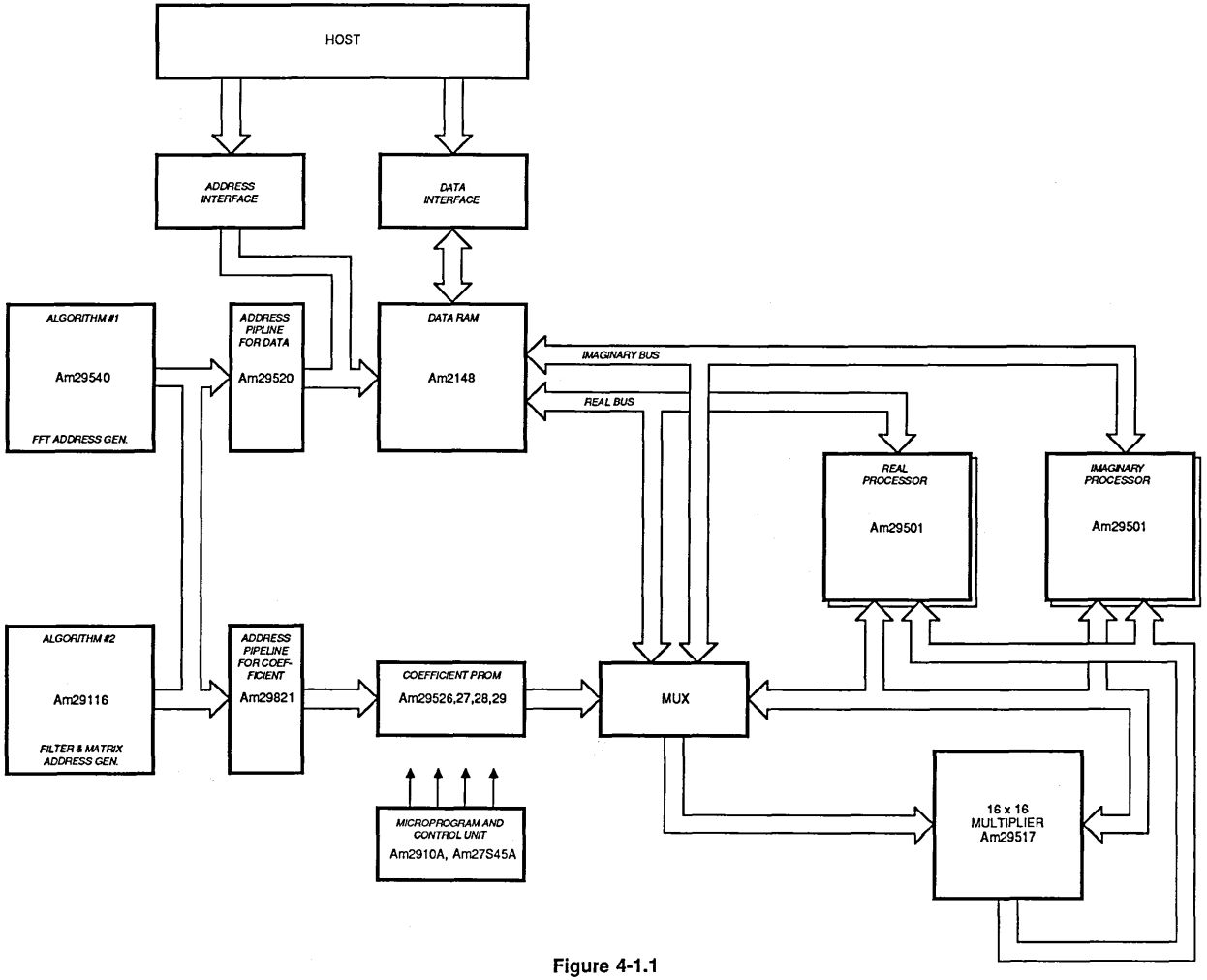

HOST
ADDRESS INTERFACE
DATA INTERFACE
ALGORITHM #1
Am29540
FFT ADDRESS GEN.
ADDRESS PIPLINE FOR DATA
Am29520
DATA RAM
Am2148
IMAGINARY BUS
REAL BUS
REAL PROCESSOR
Am29501
IMAGINARY PROCESSOR
Am29501
ALGORITHM #2
Am29116
FILTER & MATRIX ADDRESS GEN.
ADDRESS PIPELINE FOR COEF-FICIENT
Am29821
COEFFICIENT PROM
Am29526,27,28,29
MUX
16 x 16 MULTIPLIER Am29517
MICROPROGRAM AND CONTROL UNIT
Am2910A, Am27S45A

Figure 4-1.1

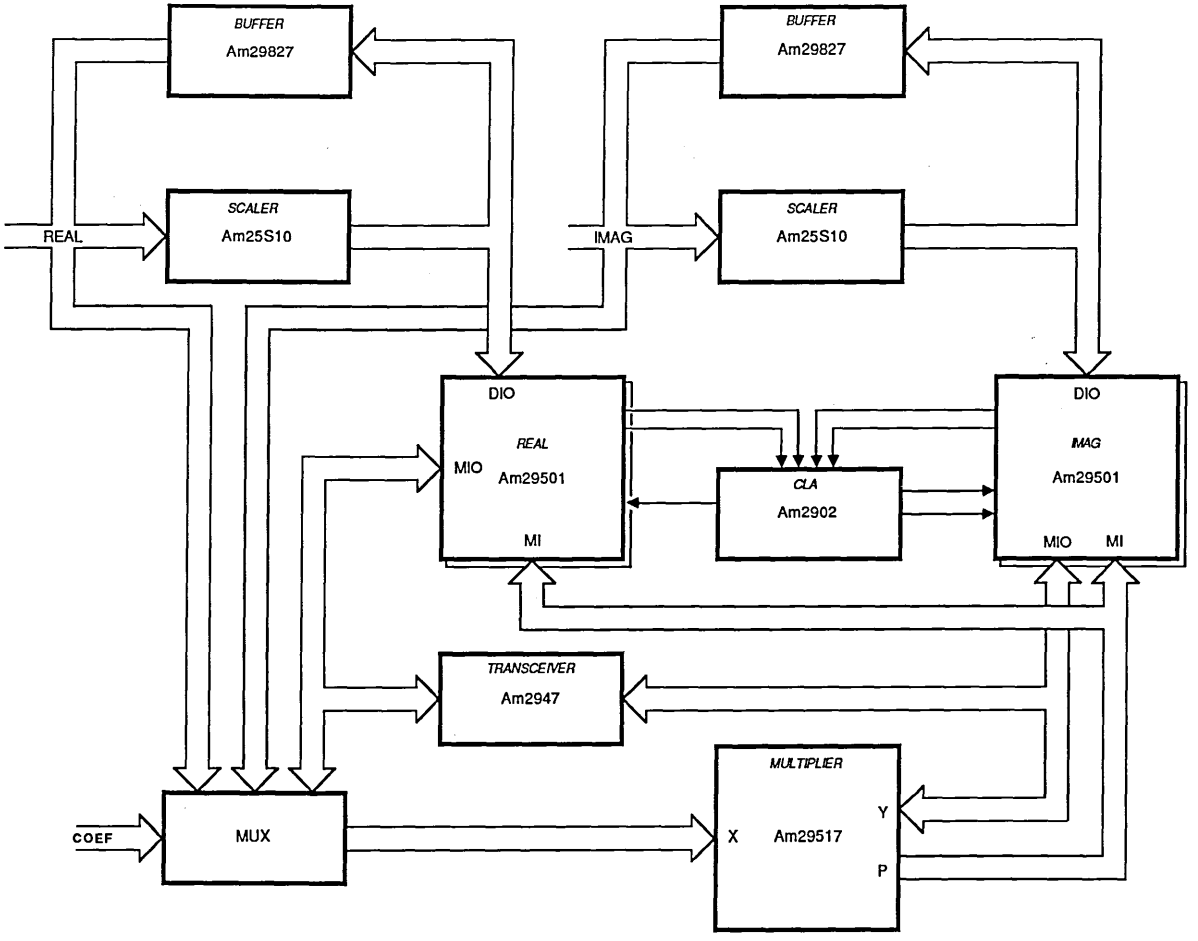

BUFFER
Am29827
BUFFER
Am29827
SCALER
Am25S10
REAL
SCALER
Am25S10
IMAG
DIO
REAL
MIO
Am29501
MI
CLA
Am2902
DIO
IMAG
Am29501
MIO
MI
TRANSCEIVER
Am2947
MULTIPLIER
Y
X
Am29517
P
COEF
MUX

Figure 4-1.2

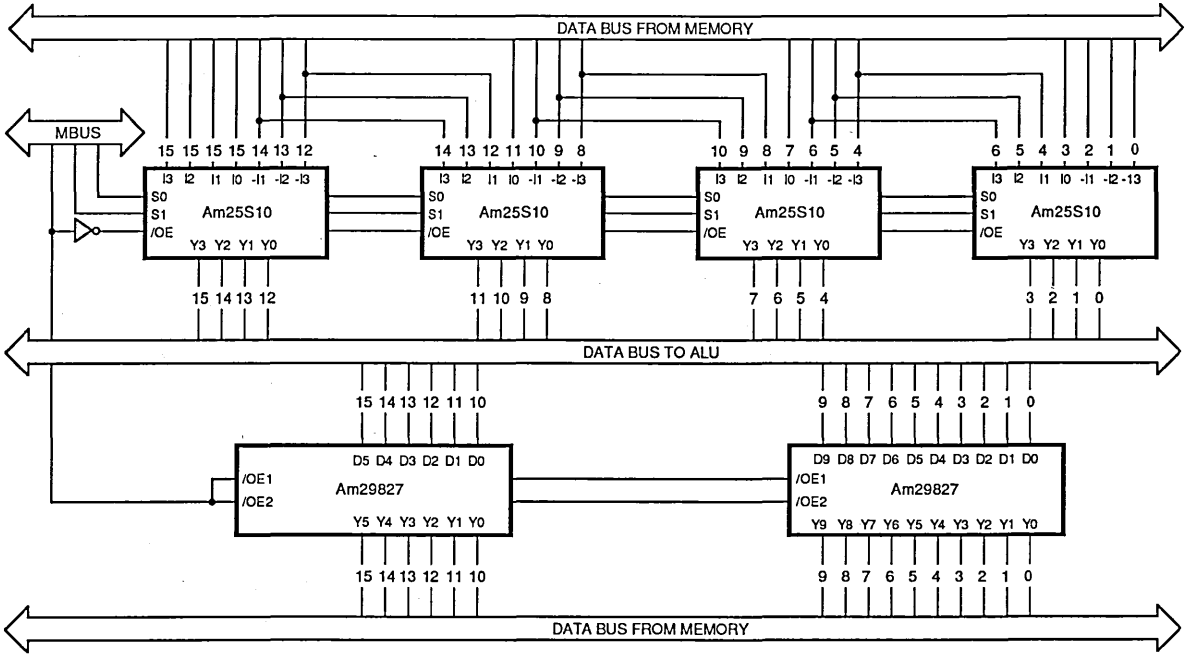

DATA BUS FROM MEMORY
MBUS
15 15 15 15 14 13 12
14 13 12 11 10 9 8
10 9 8 7 6 5 4
6 5 4 3 2 1 0
I3 I2 I1 I0 -I1 -I2 -I3
S0
S1
/OE
Am25S10
Y3 Y2 Y1 Y0
15 14 13 12
11 10 9 8
7 6 5 4
3 2 1 0
DATA BUS TO ALU
15 14 13 12 11 10
9 8 7 6 5 4 3 2 1 0
D5 D4 D3 D2 D1 D0
/OE1
/OE2
Am29827
Y5 Y4 Y3 Y2 Y1 Y0
D9 D8 D7 D6 D5 D4 D3 D2 D1 D0
Y9 Y8 Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0
DATA BUS FROM MEMORY

Figure 4-1.3

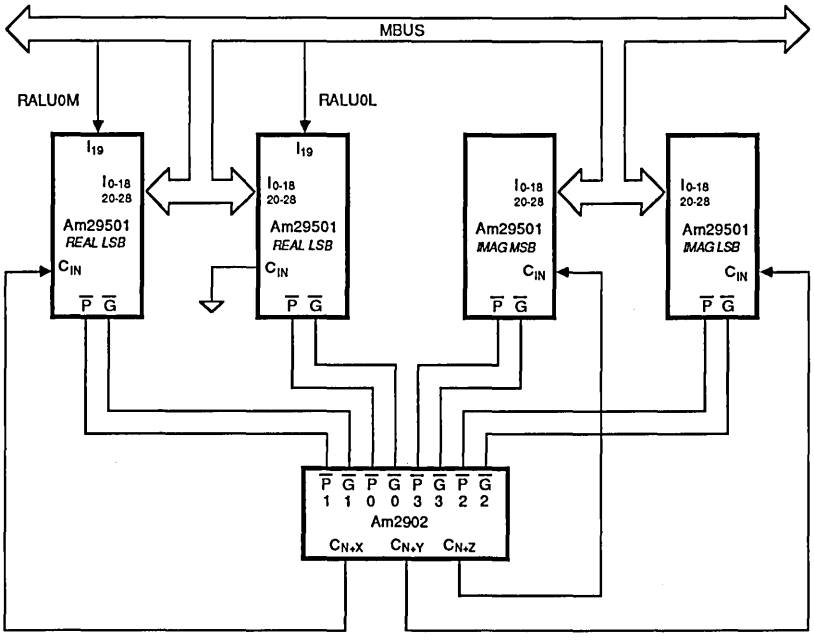


Figure 4-1.4

4.1.2 Memory

The data memory is an important consideration in the design. It must be designed to support the high-speed architecture. The minimum requirement is one memory bank, to be toggled between the host system and the DSP Processor. But this means that the DSP Processor is idle during the time data is unloaded and new data reloaded. Real-time applications would typically sample data continuously at a fixed rate. Results of the process would be read continuously from the data memory. Thus, it is necessary to have two memory banks so that, while one bank is being unloaded and reloaded, the other bank is processed by the DSP. This restricts this design to in-place FFTs.

Figure 4-1.5 shows the memory section architecture of this design. High-speed (45 ns) Am2148 memories make up the two memory banks designated L (Left) and R (Right). Each memory bank is 1K deep and 32-bits wide. Since FFTs operate on complex data, each memory bank is further divided into 16-bit real and imaginary parts designated by R and I respectively.

One line, Q (described in the Control section), controls memory bank switching. To allow parallel operations, two address buses (designated Left and Right) supply addresses simultaneously to the two memory banks. These address buses switch when Q switches so that DSP data addresses are provided to the memory on which the DSP is processing and host data addresses are provided to the other memory bank for unloading and reloading of data.

The DSP data buses are 16 bits wide. Transceivers (Am2947s) isolate one memory bank from the other on the DSP side. Q controls the Chip Enables of these transceivers so that only the appropriate set of transceivers are enabled at any time. The direction of these transceivers is controlled by the real and imaginary Write Enable bits from microcode.

Host data buses can be 8 bits or 16 bits wide. The three inverting transceivers (Am2946) at the top of Figure 4-1.5 are connected to accommodate both sizes. If the host bus is 16 bits wide, transceivers 1 and 3 are enabled by CD3 so that data is transferred directly in 16-bit words. If the host bus

is 8 bits wide, data must be transferred in bytes. First the low byte is transferred through transceiver 3 and then the high byte is transferred via transceiver 2. Notice that when the low byte is written, transceiver 1 is also enabled and invalid data enabled on the high byte. This is followed immediately by the true high byte. When data is read back by the host, there is no problem because the host bus is only 8 bits wide. Control signals CD2 and CD3 are generated from a PAL device.

The direction of these transceivers is controlled by the system Read signal, IOR, such that the transceivers are directed into the board by default.

The Am2947 transceivers on the host side of the section are to isolate the memory banks on that side. Their Chip Enables (CELR, CELI, CERR, CERI) are generated in a PAL device. Control signal Q is used to distinguish between the left and right memory banks. The system Read signal, IOR, also controls the direction of these transceivers.

Eight Write Enable lines, WE1 through WE8, are produced in a PAL device. Data Writes from the DSP side are in 16-bit words only. Two microcode bits, one for real data and the other for imaginary data, are used to generate the Write signals from the DSP side. Data Writes from the host side can be accomplished by DMA or I/O. DMA is allowed only for 16-bit transfers. The Am9517A is the DMA controller being used in the design. Since I/O Writes can be in either byte or word mode, two I/O addresses are reserved for this. When a decode of these two addresses occurs, a PAL device produces two signals, $\overline{\text{BYTEH}}$ for the high byte and $\overline{\text{BYTEL}}$ for the low byte. When the bus is used in 16-bit mode, the MULTIBUS* produces a control signal, $\overline{\text{BHEN}}$. A combination of these three signals is used to produce the eight Write Enables during data loading via I/O.

4.1.3 Addressing

To achieve parallel operation, both memory banks are addressed simultaneously, one by the host for unloading and reloading of data and the other by the DSP address generator. Figure 4-1.6 shows the architecture of this section.

Addressing from the host processor must accommodate DMA or programmed I/O. For this design, only 16-bit DMA transfers are allowed. Programmed I/O transfers, on the other hand, can be 8 bits or 16 bits. Host addressing is done by a "fly-by" counter. The counter is preloaded with the starting address from the host processor. The counter is clocked by the Read or Write signal produced by the Am9517 DMA Controller, if DMA is being used, or by the Read or Write signal from the host system, if programmed I/O is being used. A PAL device produces these clock signals for the fly-by counter. The 12-bit Counter is made up of three Am74161s (4-bit presettable counters).

The Least Significant Bit of the fly-by counter goes into the PAL device that produces the eight Write Enable signals for the memory. It distinguishes the real data from the imaginary data. The next 10 bits of the counter address the 1K deep memory. The address for a complex word of memory is the same. Thus in the 16-bit mode, the address from the fly-by counter to the memory remains unchanged for two consecutive clocks and the Least Significant Bit of the fly-by counter helps to generate a Write signal for either the real or imaginary part of the complex word. For 8-bit I/O transfers, the fly-by counter is clocked once every two 8-bit Writes so that the Least Significant Bit of the fly-by counter still distinguishes between real and imaginary data.

This design supports the following DSP processes: 1)Fast Fourier Transforms, 2) Filters, and 3) Matrix Multiplication. Addressing for the FFT is quite complex but the Am29540 provides a hardware solution. Addresses for data source, data destination and coefficients are generated by the Am29540 FFT Address Sequencer. The microcode indicates to the address sequencer the FFT type (radix 4/2; inplace, non-inplace; DIT/DIF). Four bits from the Instruction Register (described in the Control Section) indicate the transform length to the sequencer. The transform length is latched into the part at the start of the process. That's all that is required for initialization. The sequencer now produces data and coefficient address in the required order for the entire transform.

The Am29116 is programmed to produce the address sequence for Filters and Matrix Multiplication. Since the board runs just one process at a time, the Am29540 and the Am29116 are never used simultaneously. Therefore the microcode bits for the two parts are overlayed. The FFT transform length four bits from the IR indicate to the Am29116 the type of filter or the matrix size.

The data addresses must be saved in a pipeline register for efficient microcoding. The Am29520 serves as a dual, two-level pipeline register. The source addresses for the two complex inputs are saved in one level. These are moved into the second level to become the destination addresses for the results and new source addresses get put into the first level. Four bits of microcode control the two Am29520s that are connected in parallel to form the 12-bit-wide pipeline register for the data addresses.

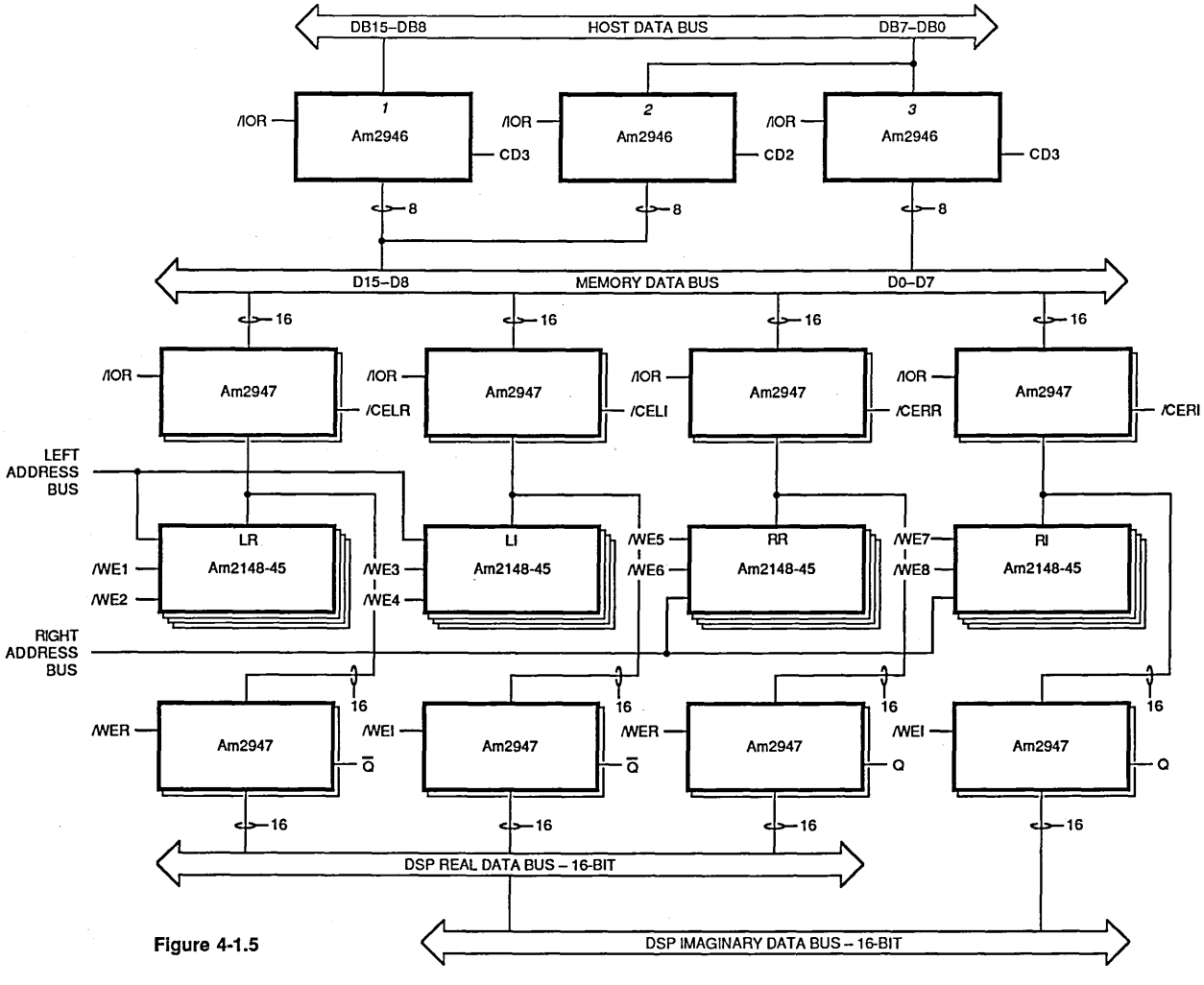

DB15–DB8
HOST DATA BUS
DB7–DB0
/IOR
1
Am2946
CD3
/IOR
2
Am2946
CD2
/IOR
3
Am2946
CD3
8
8
8
D15–D8
MEMORY DATA BUS
D0–D7
16
16
16
16
/IOR
Am2947
/CELR
/IOR
Am2947
/CELI
/IOR
Am2947
/CERR
/IOR
Am2947
/CERI
LEFT ADDRESS BUS
LR
Am2148-45
LI
Am2148-45
RR
Am2148-45
RI
Am2148-45
/WE1
/WE2
/WE3
/WE4
/WE5
/WE6
/WE7
/WE8
RIGHT ADDRESS BUS
16
16
16
16
/WER
Am2947
Q
/WEI
Am2947
Q
/WER
Am2947
Q
/WEI
Am2947
Q
16
16
16
16
DSP REAL DATA BUS – 16-BIT
DSP IMAGINARY DATA BUS – 16-BIT

Figure 4-1.5

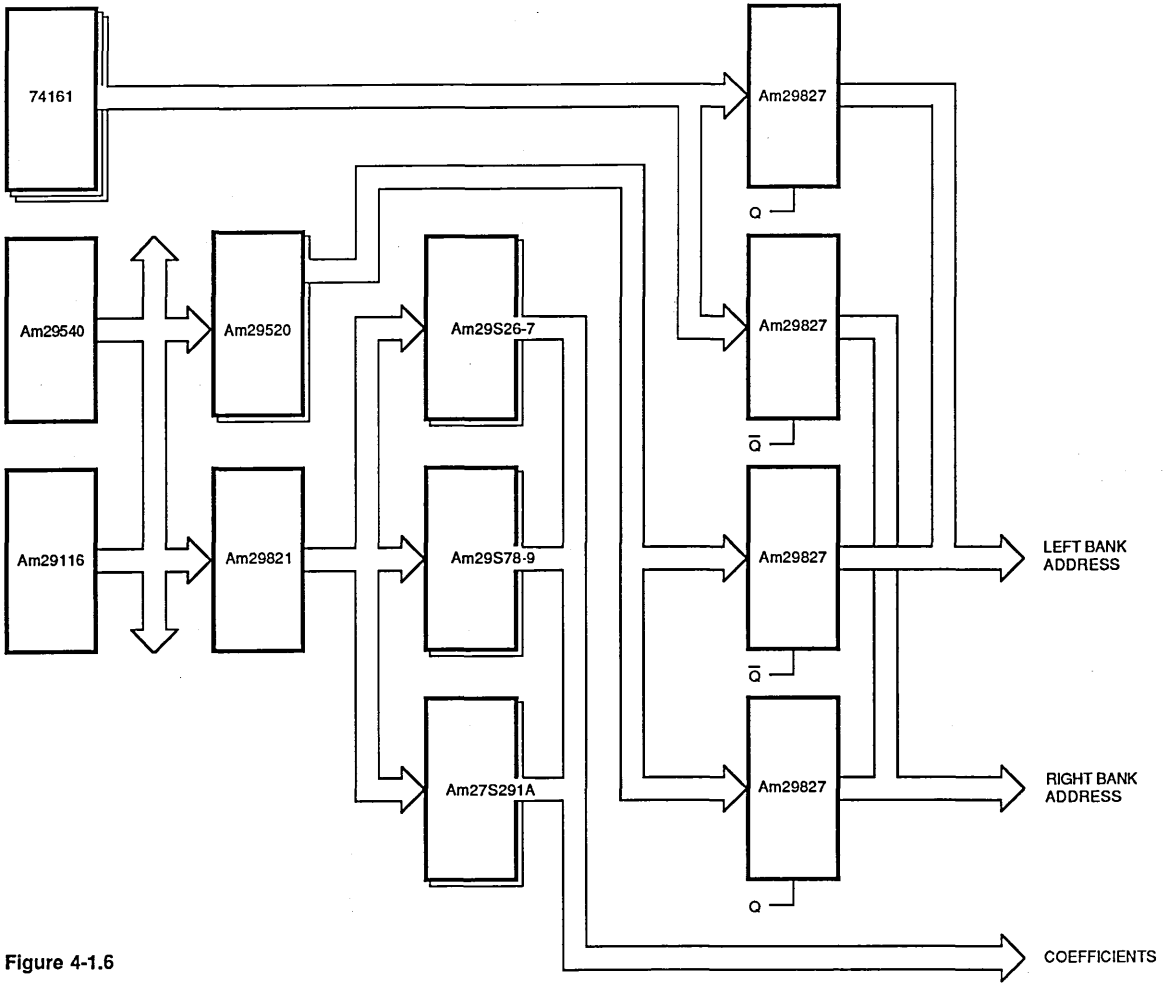

74161
Am29827
Q
Am29540
Am29520
Am29S26-7
Am29827
Q
Am29116
Am29821
Am29S78-9
Am29827
LEFT BANK ADDRESS
Q
Am27S291A
Am29827
RIGHT BANK ADDRESS
Q
COEFFICIENTS

Figure 4-1.6

The coefficient addresses need a simple, one-level pipeline register and the Am29821 10-bit register serves the purpose. For the FFT process, there are 16-bit sine and cosine coefficient PROMs (Am29526–Am29529). Two 8-bit PROMs, Am27S291s, form a third 16-bit coefficient PROM for the filter algorithm. Addresses for the sine and cosine filter coefficient PROMs are generated by the Am29540. Coefficient addresses from the Am29540 are left-justified and for radix-2 operations, the MSB is always a "0." The Am29116 is programmed so that its MSB is always a "1" when generating filter coefficient addresses. This MSB is therefore used to "Chip Select" between the FFT coefficient PROMs and the filter coefficient PROMs. A microcode bit selects between the sine and cosine during the FFT process.

The two memory banks are addressed simultaneously, one by the DSP addressing and one by the host addressing. Two address buses therefore supply these two addresses to the two memory banks. At the end of a process, the two buses must be switched, under control of the flag Q. This is achieved by the four 10-bit buffers, Am29827s. At any one time, two of these buffers are enabled by Q, one supplying host addresses to one bank of memory and the other supplying DSP addresses to the other bank. At the end of a process, these two buffers are turned off and the other two turned on so that the buses switch, effectively switching memory banks.

4.1.4 Control

This section addresses the "heart" of the design because the microcode controls the rest of the system. The microcode width must be decided during this phase of the design. A Microprogram

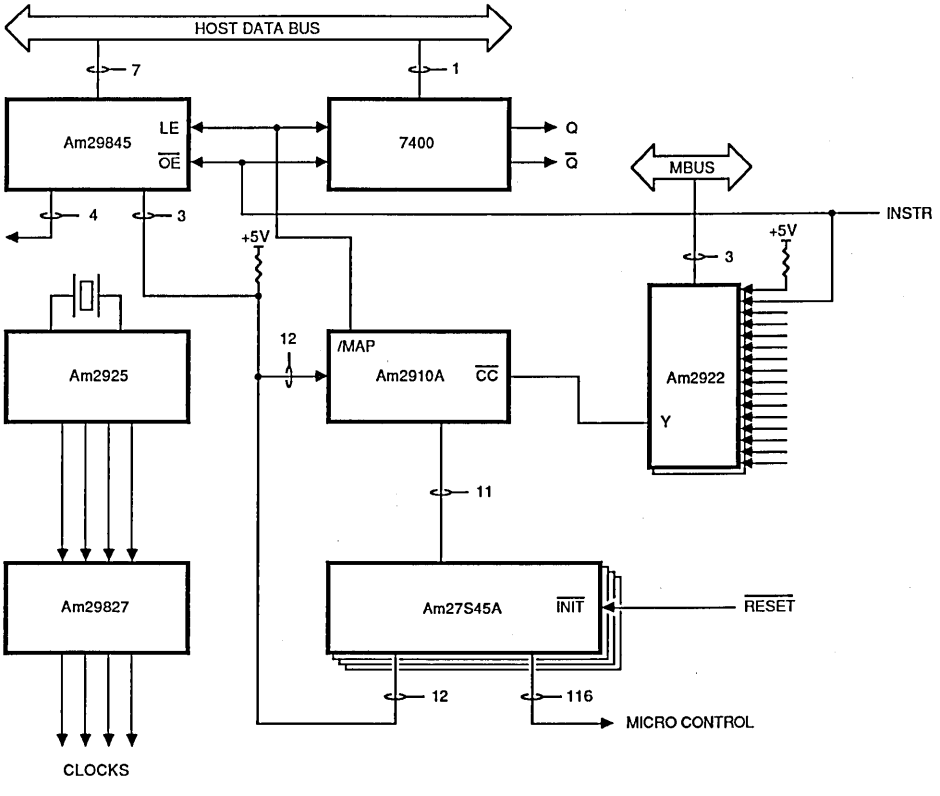


Figure 4-1.7

Sequencer is used to sequence through the microcode. A Pipeline Register is required for the microcode bits so that the sequencer can fetch the next microinstruction while the present one is executing. A Condition Code MUX is needed to test conditions. These are the basic necessities of the Control section.

Figure 4-1.7 shows the architecture for this section. The microcode width required for this design is 128 bits: 58 bits for the Real and Imaginary ALUs, 21 overlayed bits for the Addressing section, 6 bits for the Multiplier, 4 bits for the Shift Register, 4 bits for the Address Pipeline Register, 16 bits for the Microprogram Sequencer, 5 bits for the Condition Code MUX, and some other miscellaneous bits. The PROMs used are Am27S45As; they are high speed (35 ns) 2K x 8 registered PROMs. Using registered PROMs decreases part count and saves board space because the Pipeline Register is built into the PROM. The Condition Code MUX has built-in registers and so, to avoid having two registers in its path, a non-registered microcode PROM is used for it. A registered PROM could be used but the microcode for the CCMUX would be shifted by 1 line with respect to the rest of the microcode.

These PROMs have a 2049th location which can be programmed as any value. This value appears at the output of the PROM when an INIT signal is applied to the PROMs. This is a useful feature for initialization on reset. The op-code for a Jump to Zero (JZ) instruction for the sequencer is programmed into this location for initialization on reset. The reset line from the CPU is connected to the INIT input of the PROM.

The sequencer used is the Am2910A, capable of addressing up to 4K of microcode memory. We have 2K deep microcode memory in this design which is enough for the processes desired here.

Two Condition Code MUXs (Am2922s) enable testing of up to 16 inputs, of which one is used for the "forced pass" condition. The outputs of the Condition Code MUX is fed into the CC input of the sequencer for condition testing.

The control section also contains an 8-bit Instruction Register (Am29845). Four bits go to the Am29540 or Am29116 to indicate FFT transform length, filter type, or matrix size. Three bits go into the sequencer to indicate the process to be run. The eighth bit from the host goes into a latch. This is the latch for Control Signal Q that indicates to the entire system which memory bank to process. The state of this eighth bit from the host either sets or resets the Q latch.

Also included in the Control Section of the design is the clock circuit. The Am2925 is a clock generator and microcycle length controller. It produces clocks of varying duty cycles.

The following is a sequence of events that would occur from start to finish of any process. On power-up, the reset line from the CPU is activated and the op-code for a JZ instruction is put out from the microcode PROM to the sequencer. This makes the sequencer jump to the start of microcode. The sequencer now waits for an instruction to be loaded into the IR by the CPU.

On receiving an instruction, the sequencer jumps to one of eight locations at the end of the microcode. The exact location is decided by the three bits from the IR to the branch address field of the sequencer. The value of the three bits depends on what process must be run (one of eight). The sequencer jumps to one of eight locations and gets a branch address from there which would be the starting address for the process. At the end of the process, the sequencer would execute a JZ instruction and set a flag. This flag can be read by the CPU via software. The CPU can now load the IR with another instruction. The CPU should not load the IR if the flag from the microcode is inactive, as the sequencer would miss the instruction.

4.1.5 Input/Output

The I/O section is the interface between the board and the host system. Decoding logic is required to decode I/O addresses reserved for the board. Also required is the DMA controller and some registers and buffers to interface with the host address and data buses. Figure 4-1.8 is a diagram of the architecture.

The DMA Controller used in this design is the Am9517. The Am9517 is chosen primarily because of its capability to interface easily with an 8-bit CPU. If a 16-bit CPU were being used, the Am9516 would have been selected. Although an 8-bit CPU and a 16-bit DMA seem contradictory, the Multibus allows bus masters with different bus widths to exist in the same system.

I/O addresses must be reserved for this board. The DMA Controller must have 16 addresses reserved for it. Other I/O addresses needed are for I/O memory writes, fly-by counter loading, checking process complete status flag, loading three MSBs of address for DMA (described later in this section), loading the instructions into the IR, and initialization of the board. Thirty-two I/O addresses are reserved for this board.

The Am29809, an 8-Bit Comparator, produces a board select when one of these 32 I/O addresses is put on the bus by the CPU. When using DMA, the controller must be programmed. This is done via Buffer 1 (Am29828). When an address decode occurs for the DMA Controller, Buffer 1 is "output enabled" and the Am9517 is "chip selected" and programmed. The MULTIBUS address bus is 20 bits wide. Since DMA is allowed only in 16-bit mode and this design is for an 8-bit CPU, the LSB of the address bus is grounded. The Am9517 operates in 256-byte pages. The low order 8 bits go from the Am9517 to the host address bus via Buffer 2. The output of this buffer is enabled by the acknowledge line from the controller. The high order 8 bits of the address must be latched into a register. Device #3 on the diagram is an 8-bit register into which these 8 bits are latched. The MULTIBUS address bus being 20 bits wide, the remaining 3 bits must be loaded into a second register (Device #4 on the diagram) by the CPU. An I/O address (signal name ADDRL) is reserved for this function.

An I/O address is also reserved for the process complete status flag. This flag from microcode goes into Buffer 5 and this buffer's output is enabled when the address decode for this flag occurs. The CPU reads the value of the buffer and decides if the DSP process is complete.

Appendix 1 lists the equations for the three PAL devices used in the I/O section.

4.1.6 Timing Considerations

This is probably the most critical phase of the design. "Timing Considerations" refers to the speed at which the design will actually run. This should be done before the board is built. The designer starts out with a certain set of goals which the design has to meet. These goals can be divided into two categories. Category 1 lists the different algorithms that the design has to process. Category 2 lists how fast these processes must be run. The designer starts with the goals in Category 1. He designs his architecture so that all algorithms can be run by the design. He then writes microcode for the processes. Knowing the architecture and having written the microcode, the designer can now evaluate worst case data paths and can compute process times and compare them with the goals in Category 2. All is well if the goals in both categories are met. If not, trade-offs must be made. First, the designer should try to make the microcode more efficient. If this is not sufficiently effective, the architecture needs to be changed or some of the goals need to be relaxed. Changing the architecture usually implies adding more hardware so that the design has more processing power. This is not always possible due to board space and cost limitations. The alternative solution, relaxing some of the goals, could mean either disallowing some of the algorithms thus getting rid of some hardware and thereby eliminating propagation delays and increasing speed, or finally, deciding that the slower speeds are acceptable.

In this design, the Category 1 goals were:
a) 1K Complex FFT;
b) Filters;
c) Matrix Multiplication.

The Category 2 goal was:
a) 1K Complex FFT in 2 ms.

The hardware is designed and microcode for the FFT has been written. Each butterfly takes 4 cycles. Now the worst case data path needs to be computed which would decide the minimum cycle time. To achieve the 2 ms goal, the cycle time should be no greater than 100 ns.

All data paths should be and have been considered. Three of the worst paths are:

1) Clock to output of microprogram pipeline register
   + select to output of Am29520
   + prop delay of bus switching transceivers
   + data memory access time
   + prop delay of transceivers separating memory banks on DSP side
   + shifters
   + data set up time of Am29501
   = 141 ns.

2) Clock to output of microprogram pipeline reg
   + INST to output of Am29116
   + data setup time of Am29520
   = 104 ns.

3) Clock to output of microprogram pipeline reg
   + 2 slice delay of Am29501
   + CCMUX prop delay
   + Am2910 setup and prop delay
   + microprogram memory access time
   = 178 ns.

The worst case path is 178 ns which means that the maximum clock frequency can be 5.62 MHz; less than the goal of 10 MHz. Reducing microcode is not possible. With the architecture chosen, doing a butterfly in 4 cycles is the best one can do; it's time to make a trade-off. As this design is for an evaluation board, the goals of Category 1 cannot be relaxed. The two possible paths left are to add more computing power to the design or relax the goals of Category 2. Again, because this design is

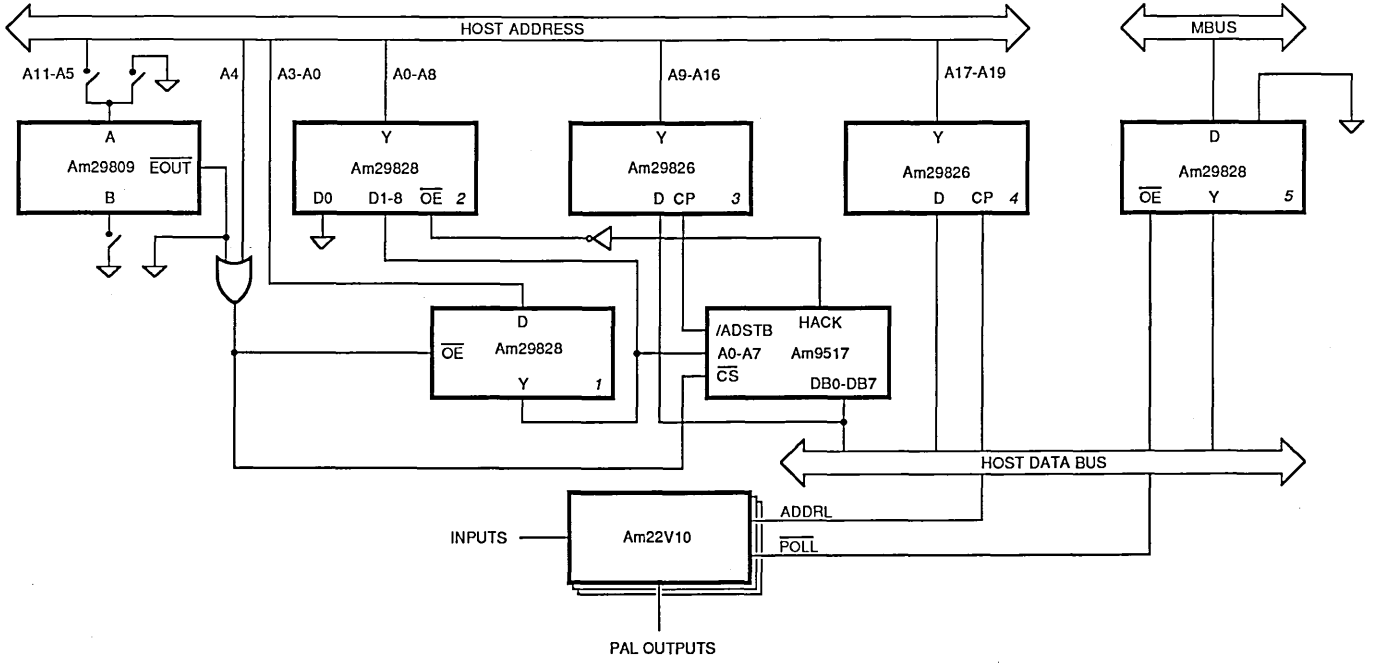

HOST ADDRESS
MBUS
A11-A5
A4
A3-A0
A0-A8
A9-A16
A17-A19
A
Am29809
EOUT
B
Y
Am29828
D0
D1-8
OE
2
Y
Am29826
D CP
3
Y
Am29826
D CP
4
D
Am29828
OE
Y
5
D
OE
Am29828
Y
1
/ADSTB
A0-A7
CS
HACK
Am9517
DB0-DB7
HOST DATA BUS
INPUTS
Am22V10
ADDRL
POLL
PAL OUTPUTS

Figure 4-1.8

for an evaluation board, the latter of the two paths is chosen. The worst case path is not likely to be executed very often, so it is supported by changing the clock period with the Am2925, and the 141 ns path is considered worst case making the clock frequency 7.09 MHz. This path can be improved by putting in faster transceivers and faster data memories like the Am2148-35 or the Am9150-25. Using the latter brings the speed up to 8.26 MHz. Adding more computing power would mean providing more ALU's and/or multipliers. The designer would have to evaluate by how much the microcode would reduce if there was more arithmetic processing power. Timing paths would have to be recalculated and a new estimate made for the maximum clock frequency.

4.1.7 Microcode

This section deals with the software aspect of the design. Microprogramming involves writing a coherent sequence of microinstructions used to execute the various steps required by the process. A microinstruction usually has two primary parts: 1) the definition and control of all elemental micro-operations to be carried out; and 2) the definition and control of the address of the next microinstruction to be executed.

For our design, the definition of the various micro-operations to be carried out includes the Real and Imaginary ALU's, multiplier, data address generation, data memory control, address pipeline registers, shifters, clock controller and condition code MUX select. The definition of the next microinstruction function includes identifying the source selection of the next microinstruction address and supplying the starting address for any process.

Two basic principles should be remembered when writing microcode: 1) parallel execution of different operations, and 2) maximum utilization of resources due to the pipelined architecture.

The microcode for the FFT is described here in some detail.

The FFT algorithm is highly repetitive. The same butterfly operations are performed on different sets of numbers. Each radix 2 butterfly consists of four multiplies and six adds/subs. The code developed here is for a radix 2 DIT FFT. The equations for a radix 2 DIT butterfly are:

$$A1 = A + BWk \qquad B1 = A - BWk$$

where A and B are the complex input points, A1 and B1 are the complex results and Wk is the complex coefficient.

The following rules should be kept in mind when developing microcode:

a) Determine the program repetition rate. We have determined that our repetition rate is R = 4, i.e., each butterfly will take 4 cycles.

b) Start programming at line R+1=5.

c) For every program entry, enter an 'X' R cycles above and below (Table 4-3).

The arithmetic section will be programmed first. Table 4-2 is a programming work sheet for this section. Figures 4-1.9 through 4-1.18 and Tables 4-3 through 4-7 show the development of the code for the arithmetic section. Each block diagram showing data movement for a particular line of code is followed by a coding sheet showing the corresponding line. If a conflict of resources occurs, another resource must be used or the function re-scheduled for execution.

Note in Table 4-9 that code is repeated every 4th line. Also note that the ALU's and multiplier are utilized 100% of the time. Also note in Table 4-9 that a new butterfly starts every 4th cycle. It is the pipelined process that makes the butterfly time equal to 4 cycles. At any given time, computation for 3 butterflies is in progress as illustrated in Figure 4-1.19.

Next the code for the address generating section needs to be written. Obviously it needs to be mapped into the code already written for the arithmetic section. Tables 4-10 and 4-11. show the coding for the coefficient PROM select and for the address pipeline register for the coefficient PROMs. Tables 4-12 through 4-14 show the coding for the FFT address generator and the pipeline register. Figures 4-1.20 through 4-1.30 show the data flow from the address generator through the data and coefficient address pipeline register to the memories.

Having decided upon the code, it is now necessary to actually write it. A symbolic language would no doubt be of great help. This is possible with the AMDASM meta-assembler which is used in two phases. The first phase consists of defining the microinstructions and the language. This is done by creating a file of the type XXXX.DEF (see Appendix 2, DSP.DEF) which contains:

a) The microword width (WORD 128).

b) The list of language mnemonics. Each mnemonic would be associated with an instruction field defining the bits controlling each resource.

Table 4-2

Step	DIO	Real							Imaginary							Multiplier	
		ALU	A_1	A_2	A_3	B_1	B_2	B_3	ALU	A_1	A_2	A_3	B_1	B_2	B_3	MIO	MULT
1																	
2																	
3																	
4																	
5																	
6																	
7																	
8																	
9																	
10																	
11																	
12																	
13																	
14																	
15																	
16																	
17																	
18																	

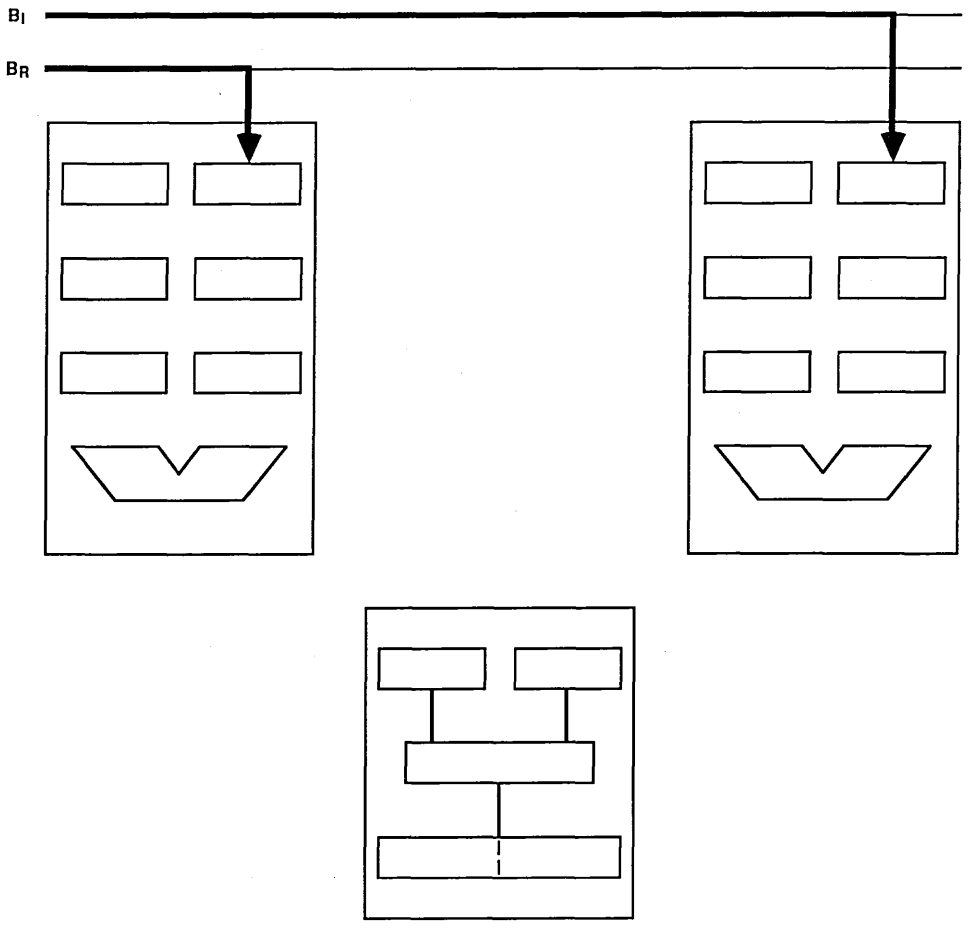


Figure 4-1.9 Line 5: Read B.

Table 4-3

Step	DIO	Real							Imaginary							Multiplier	
		ALU	A_1	A_2	A_3	B_1	B_2	B_3	ALU	A_1	A_2	A_3	B_1	B_2	B_3	MIO	MULT
1	X					X							X				
2																	
3																	
4																	
→ 5	Read B					DI							DI				
6																	
7																	
8																	
9	X					X							X				
10																	
11																	
12																	
13																	
14																	
15																	
16																	
17																	
18																	

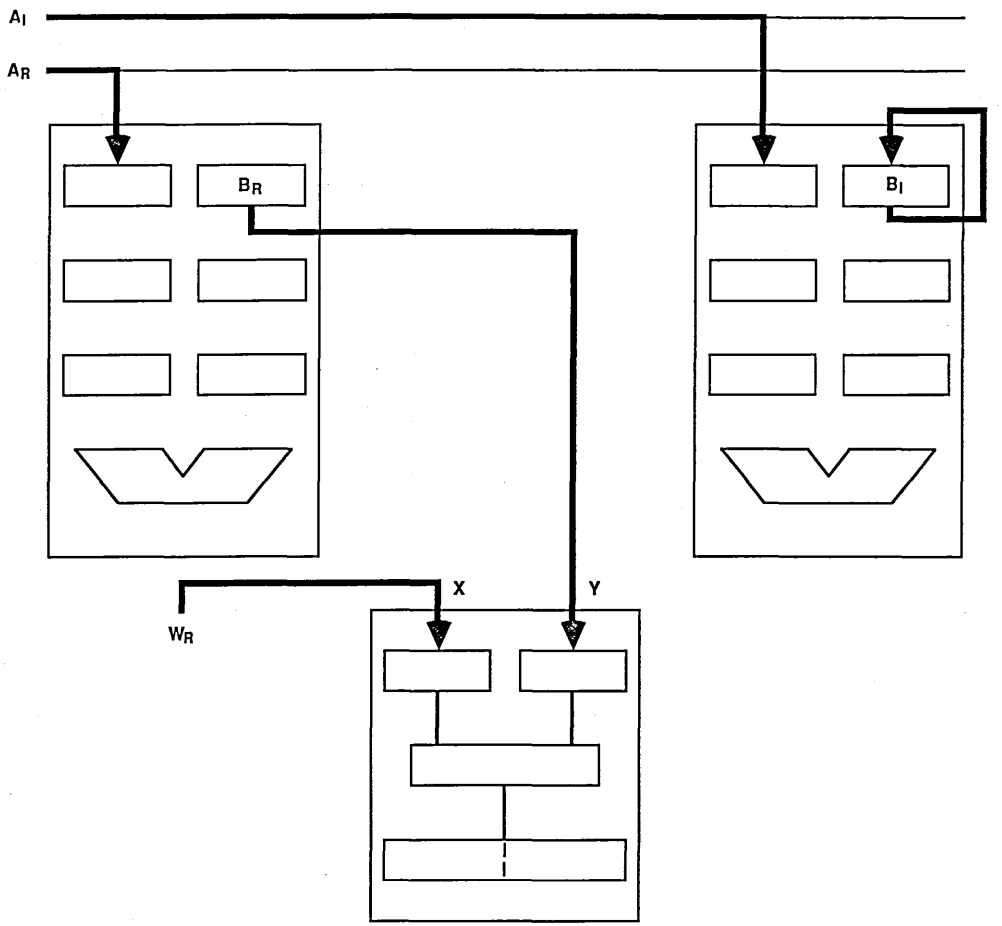


Figure 4-1.10 Line 6: Read A, present B_R to Mult Y; hold B_I, present W_R to Mult X.

Table 4-4

Step	DIO	Real							Imaginary							Multiplier	
		ALU	A_1	A_2	A_3	B_1	B_2	B_3	ALU	A_1	A_2	A_3	B_1	B_2	B_3	MIO	MULT
1	X					X							X				
2	X		X							X			X			X	
3																	
4																	
5	Read B					DI							DI				
→ 6	Read A		DI							DI			H			B_R	
7																	
8																	
9	X					X							X				
10	X		X							X			X			X	
11																	
12																	
13																	
14																	
15																	
16																	
17																	
18																	

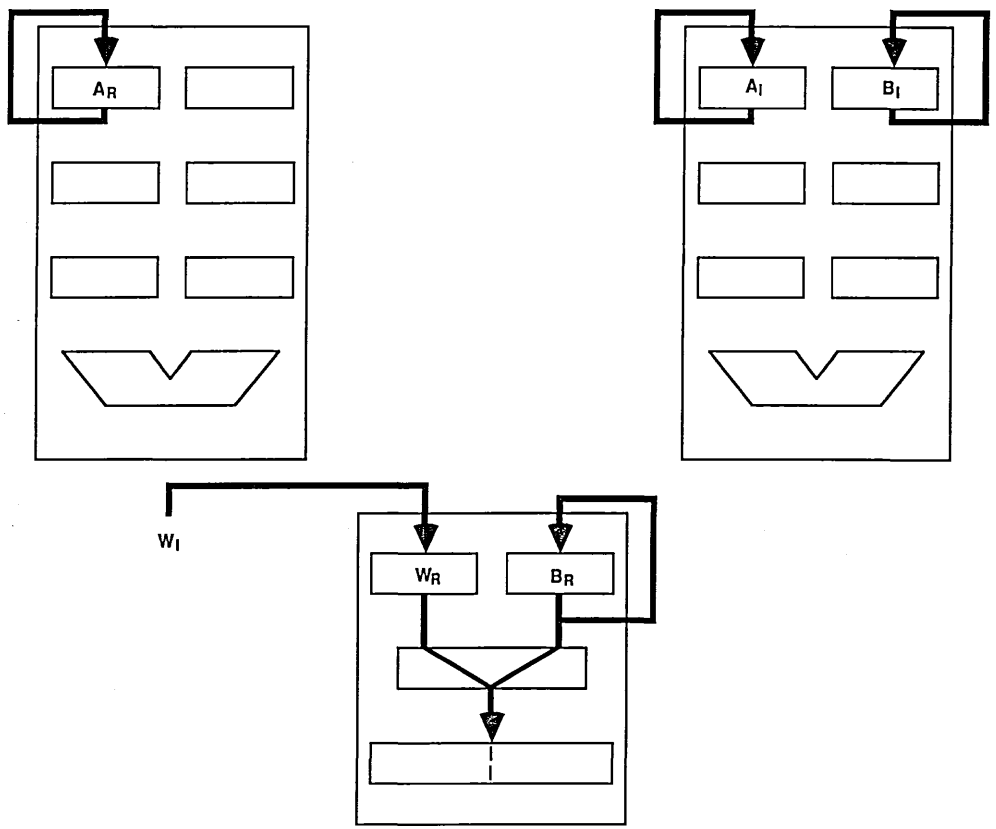


Figure 4-1.11 Line 7: multiply B_RW_R; hold B_R in Mult Y-Reg

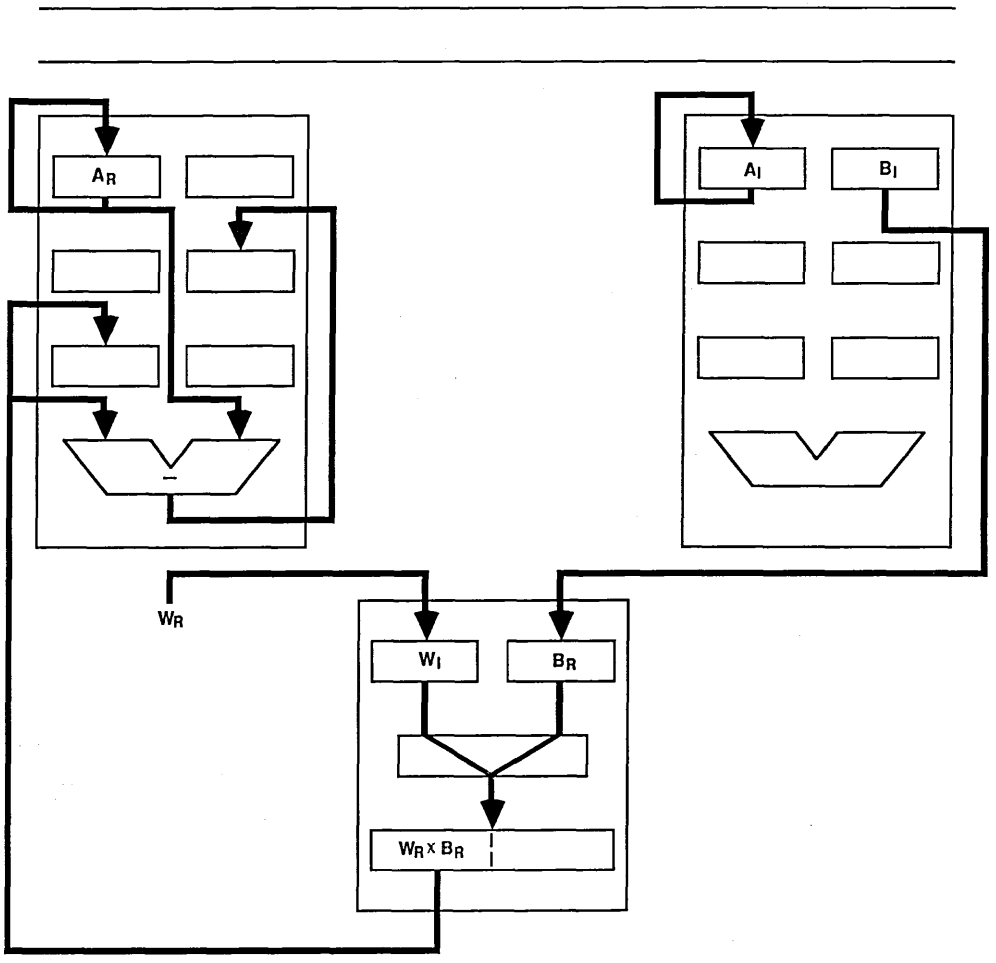


Figure 4-1.12 Line 8: The product $W_R B_R$ is used by Re ALU to start to form B'. Save product in A_3 Re A'.

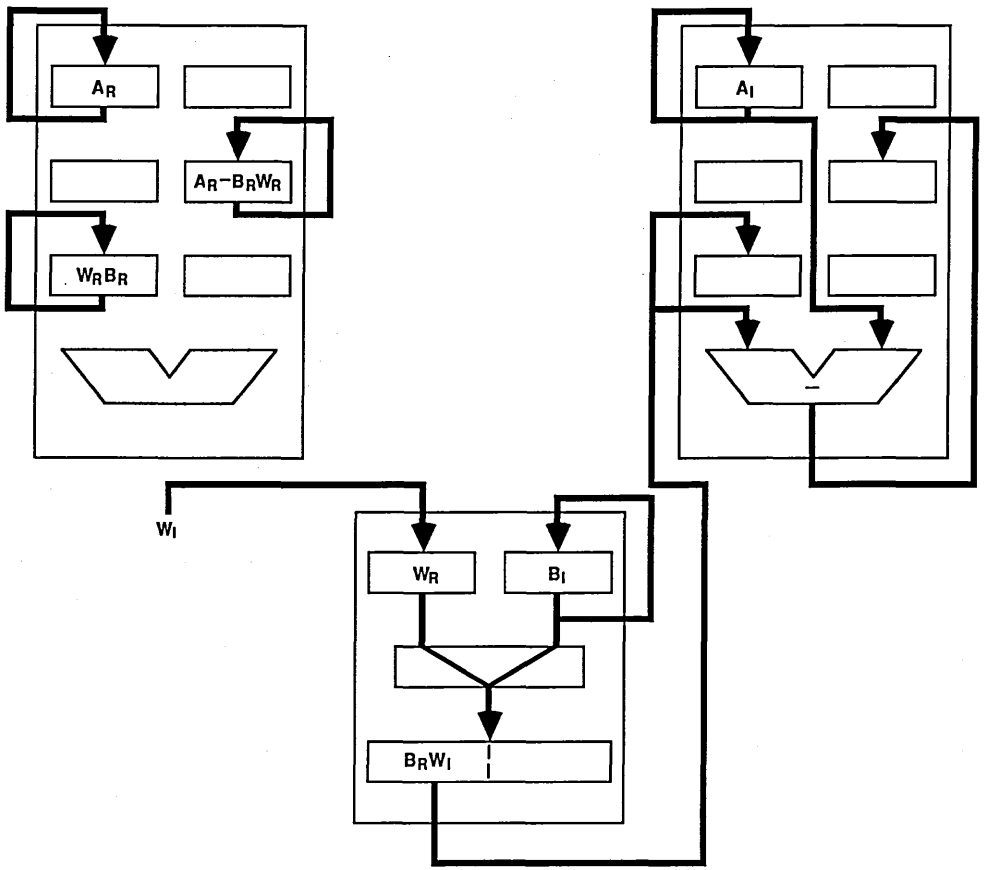


Figure 4-1.13 Line 9: Product $B_R W_I$ used by Im ALU to start B'. Save this product in A_3 for A' later.

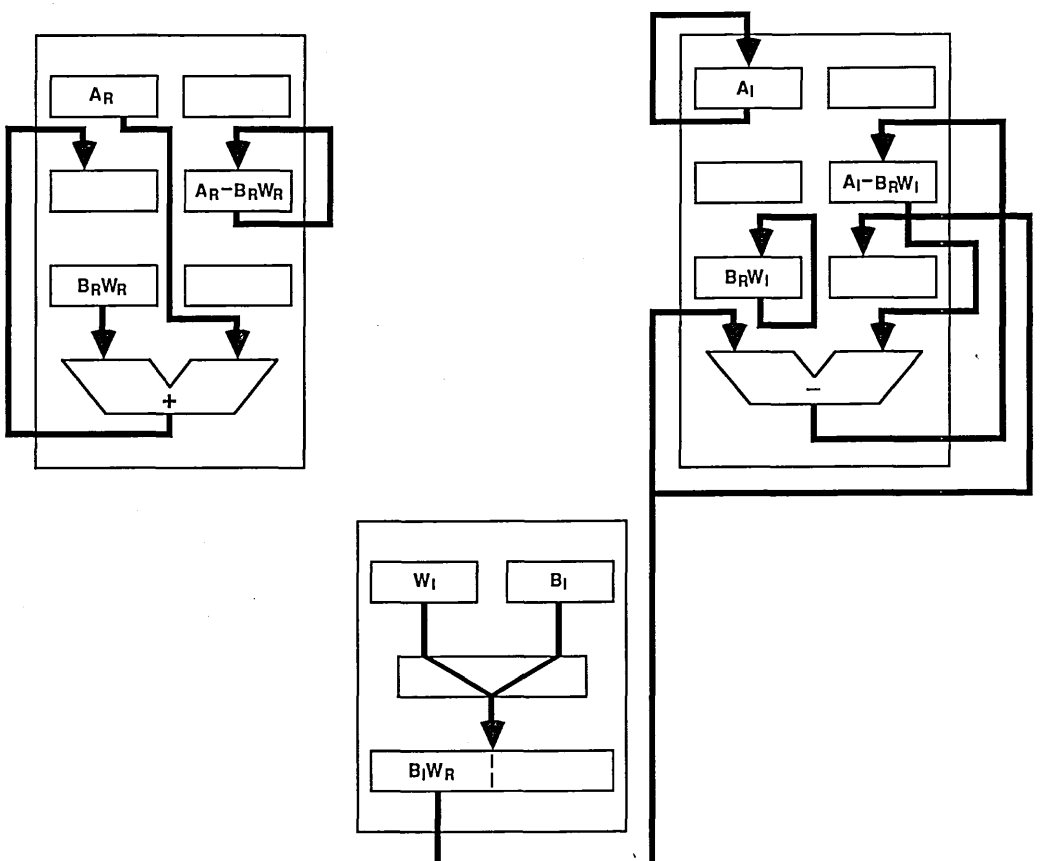


Figure 4-1.14 Line 19: $B_I W_R$ used by Im ALU to complete B'. Also save this product for A' later. Re ALU starts on A'.

Table 4-5 Line 10: a conflict has developed in Im A_1

		Real							Imaginary							Multiplier	
Step	DIO	ALU	A_1	A_2	A_3	B_1	B_2	B_3	ALU	A_1	A_2	A_3	B_I	B_2	B_3	MIO	MULT
1	X					X							X				
2	X		X							X			X			X	
3			X							X			X				X
4		X	X		X		X			X						X	X
5	Read B		X		X	DI	X		X	X		X	DI	X			X
6	Read A	X	DI	X			X		X	X/DI		X	H	X	X	B_R	X
7			H							H			H				$B_R W_R$
8		A1-MSP	H		MSP		ALU			H						B_I	$B_R W_I$
9	X		H		H	X	H		A1-MSP	H		MSP	X	ALU			$B_I W_R$
→ 10	X	A1 + A3	X	ALU			H		B2-MSP	H/X		H	X	ALU	MSP	X	$B_I W_I$
11			X							X			X				X
12		X	X		X		X			X						X	X
13			X		X		X		X	X		X		X			X
14		X		X			X		X	X		X		X	X		X
15																	
16																	
17																	
18																	

↑

Table 4-6 Line 10: conflict in Im A_1 resolved by delaying Read A one cycle to Line 7 (from 6).

		Real							Imaginary							Multiplier	
Step	DIO	ALU	A_1	A_2	A_3	B_1	B_2	B_3	ALU	A_1	A_2	A_3	B_1	B_2	B_3	MIO	MULT
1	X					X							X				
2													X			X	
3	X		X							X			X				X
4		X	X		X		X			X						X	X
5	Read B		X		X	DI	X		X	X		X	DI	X			X
6		X		X			X		X	X		X	H	X	X	B_R	X
→ 7	Read A		DI			X				DI			H				$B_R W_R$
8		A1-MSP	H		MSP		ALU			H			X			B_I	$B_R W_I$
9	X		H		H		H		A1-MSP	H		MSP	X	ALU			$B_I W_R$
→ 10		A1 + A3		ALU			H		B2-MSP	H		H	X	ALU	MSP	X	$B_I W_I$
11	X		X							X							X
12		X	X		X		X			X						X	X
13			X		X		X		X	X		X		X			X
14		X		X			X		X	X		X		X	X		X
15																	
16																	
17																	
18																	

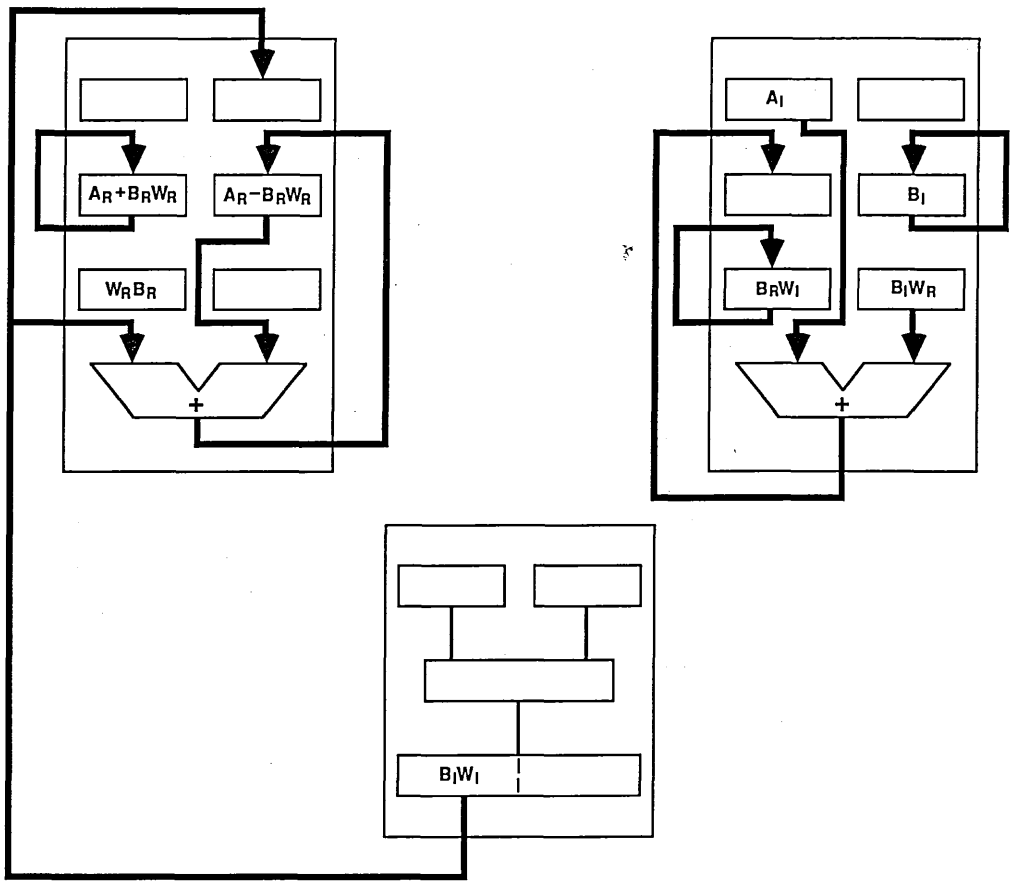


Figure 4-1.15 Line 11: B_IW_I used by Re ALU to complete B′. Hold this product Re A′ later. Im ALU starts on A′.

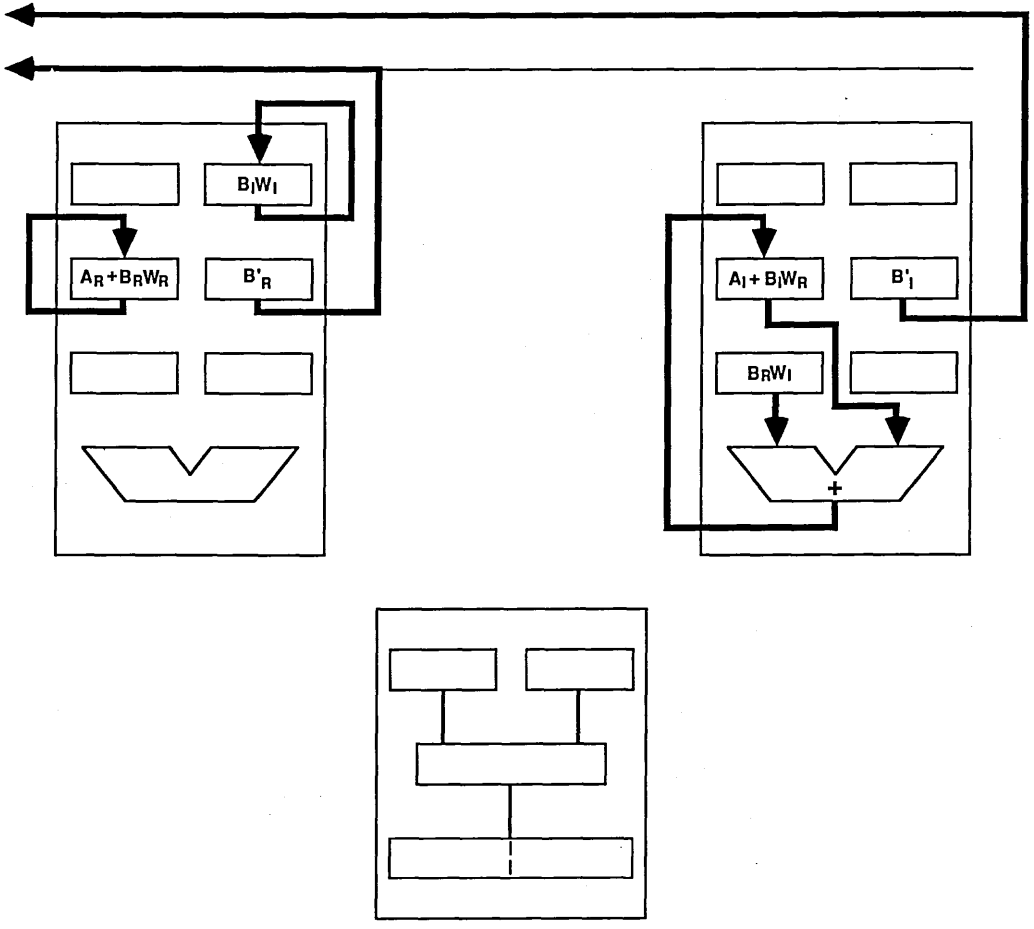


Figure 4-1.16 Line 12: Write B' back to memory from B2's. Im ALU completes A'.

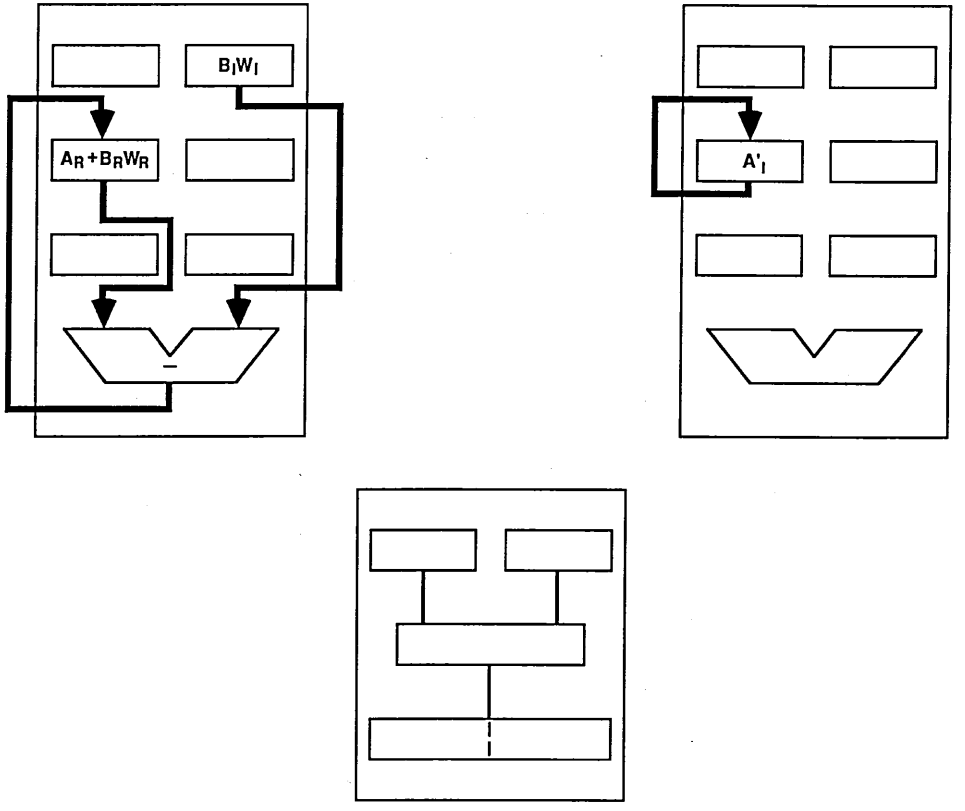


Figure 4-1.17 Line 13: Re ALU completes A'. Im ALU holds A_I as completed in previous line.

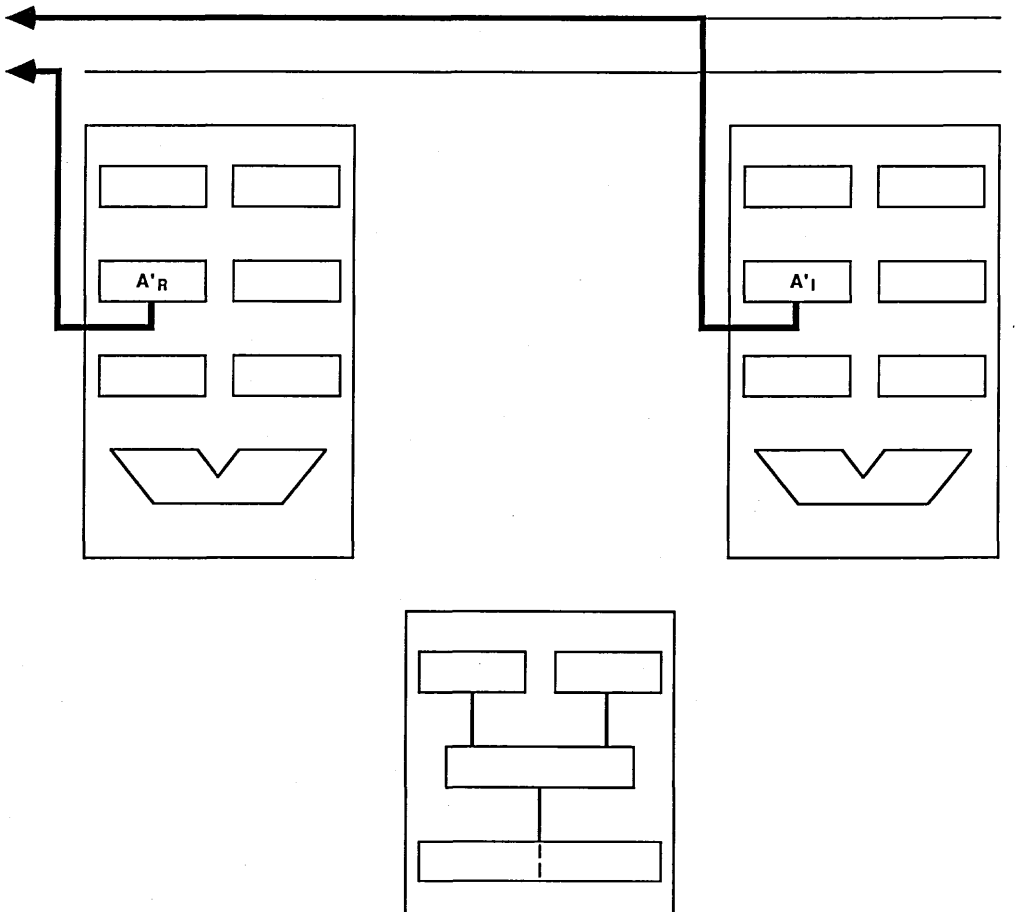


Figure 4-1.18 Line 14: Write A′ back to memory from A2's. This completes the butterfly started in Line 5.

Table 4-7 One complete butterly: lines 5–14 (10 lines).

Step	DIO	Real							Imaginary							Multiplier	
		ALU	A_1	A_2	A_3	B_1	B_2	B_3	ALU	A_1	A_2	A_3	B_1	B_2	B_3	MIO	MULT
1	X					X							X				
2													X			X	
3	X		X							X			X				X
4		X	X		X		X			X						X	X
→ 5	Read B		X		X	DI	X		X	X		X	DI	X			X
6		X		X			X		X	X		X	H	X	X	B_R	X
7	Read A	X	DI	X		X	X		X	DI	X	X	H	X			$B_R W_R$
8	X	A1-MSP	H	X	MSP	X	ALU		X	H	X					B_I	$B_R W_I$
9	X	X	H	X	H	X	H		A1-MSP	H	X	MSP	X	ALU			$B_I W_R$
10	X	A1 + A3		ALU			H		B2-MSP	H		H	X	ALU	MSP	X	$B_I W_I$
11	X	B2+MSP	X	H		MSP	ALU		A1 + B3	X	ALU	H	X	H			X
12	Write B2	X	X	H	X	H	X		A2 + A3	X	ALU					X	X
13		A2 - B1	X	ALU	X		X		X	X	H	X		X			X
→ 14	Write A2	X		X			X		X	X		X		X	X		X
15		X		X		X	X		X		X	X		X			
16	X			X		X			X		X						
17		X		X							X						
18	X																

Table 4-8 The code repeats every 4 lines. Thus a new butterfly is completed every 4 cycles.

		Real							Imaginary							Multiplier	
Step	DIO	ALU	A_1	A_2	A_3	B_1	B_2	B_3	ALU	A_1	A_2	A_3	B_1	B_2	B_3	MIO	MULT
1	Read B	A2 - B1	H	ALU	H	DI	H		A1-MSP	H	H	MSP	DI	ALU			$B_I W_R$
2	Write A2	A1 + A3		ALU			H		B2-MSP	H		H	H	ALU	MSP	B_R	$B_I W_I$
3	Read A	B2+MSP	DI	H		MSP	ALU		A1 + B3	DI	ALU	H	H	H			$B_R W_R$
4	Write B2	A1-MSP	H	H	MSP	H	ALU		A2 + A3	H	ALU					B_I	$B_R W_I$
5	Read B	A2 - B1	H	ALU	H	DI	H		A1-MSP	H	H	MSP	DI	ALU			$B_I W_R$
6	Write A2	A1 + A3		ALU			H		B2-MSP	H		H	H	ALU	MSP	B_R	$B_I W_I$
7	Read A	B2+MSP	DI	H		MSP	ALU		A1 + B3	DI	ALU	H	H	H			$B_R W_R$
8	Write B2	A1-MSP	H	H	MSP	H	ALU		A2 + A3	H	ALU					B_I	$B_R W_I$
9	Read B	A2 - B1	H	ALU	H	DI	H		A1-MSP	H	H	MSP	DI	ALU			$B_I W_R$
10	Write A2	A1 + A3		ALU			H		B2-MSP	H		H	H	ALU	MSP	B_R	$B_I W_I$
11	Read A	B2+MSP	DI	H		MSP	ALU		A1 + B3	DI	ALU	H	H	H			$B_R W_R$
12	Write B2	A1-MSP	H	H	MSP	H	ALU		A2 + A3	H	ALU					B_I	$B_R W_I$
13	Read B	A2 - B1	H	ALU	H	DI	H		A1-MSP	H	H	MSP	DI	ALU			$B_I W_R$
14	Write A2	A1 + A3		ALU			H		B2-MSP	H		H	H	ALU	MSP	B_R	$B_I W_I$
15	Read A	B2+MSP	DI	H		MSP	ALU		A1 + B3	DI	ALU	H	H	H			$B_R W_R$
16	Write B2	A1-MSP	H	H	MSP	H	ALU		A2 + A3	H	ALU					B_I	$B_R W_I$
17	Read B	A2 - B1	H	ALU	H	DI	H		A1-MSP	H	H	MSP	DI	ALU			$B_I W_R$
18	Write A2	A1 + A3		ALU			H		B2-MSP	H		H	H	ALU	MSP	B_R	$B_I W_I$

Table 4-9 A new butterfly must also start every 4 cycles.

		Real							Imaginary							Multiplier	
Step	DIO	ALU	A_1	A_2	A_3	B_1	B_2	B_3	ALU	A_1	A_2	A_3	B_1	B_2	B_3	MIO	MULT
1	Read B	A2 - B1	H	ALU	H	DI	H		A1-MSP	H	H	MSP	DI	ALU			$B_I W_R$
2	Write A2	A1 + A3		ALU			H		B2-MSP	H		H	H	ALU	MSP	B_R	$B_I W_I$
3	Read A	B2+MSP	DI	H		MSP	ALU		A1 + B3	DI	ALU	H	H	H			$B_R W_R$
4	Write B2	A1-MSP	H	H	MSP	H	ALU		A2 + A3	H	ALU					B_I	$B_R W_I$
5	Read B	A2 - B1	H	ALU	H	DI	H		A1-MSP	H	H	MSP	DI	ALU			$B_I W_R$
6	Write A2	A1 + A3		ALU			H		B2-MSP	H		H	H	ALU	MSP	B_R	$B_I W_I$
7	Read A	B2+MSP	DI	H		MSP	ALU		A1 + B3	DI	ALU	H	H	H			$B_R W_R$
8	Write B2	A1-MSP	H	H	MSP	H	ALU		A2 + A3	H	ALU					B_I	$B_R W_I$
9	Read B	A2 - B1	H	ALU	H	DI	H		A1-MSP	H	H	MSP	DI	ALU			$B_I W_R$
10	Write A2	A1 + A3		ALU			H		B2-MSP	H		H	H	ALU	MSP	B_R	$B_I W_I$
11	Read A	B2+MSP	DI	H		MSP	ALU		A1 + B3	DI	ALU	H	H	H			$B_R W_R$
12	Write B2	A1-MSP	H	H	MSP	H	ALU		A2 + A3	H	ALU					B_I	$B_R W_I$
13	Read B	A2 - B1	H	ALU	H	DI	H		A1-MSP	H	H	MSP	DI	ALU			$B_I W_R$
14	Write A2	A1 + A3		ALU			H		B2-MSP	H		H	H	ALU	MSP	B_R	$B_I W_I$
15	Read A	B2+MSP	DI	H		MSP	ALU		A1 + B3	DI	ALU	H	H	H			$B_R W_R$
16	Write B2	A1-MSP	H	H	MSP	H	ALU		A2 + A3	H	ALU					B_I	$B_R W_I$
17	Read B	A2 - B1	H	ALU	H	DI	H		A1-MSP	H	H	MSP	DI	ALU			$B_I W_R$
18	Write A2	A1 + A3		ALU			H		B2-MSP	H		H	H	ALU	MSP	B_R	$B_I W_I$

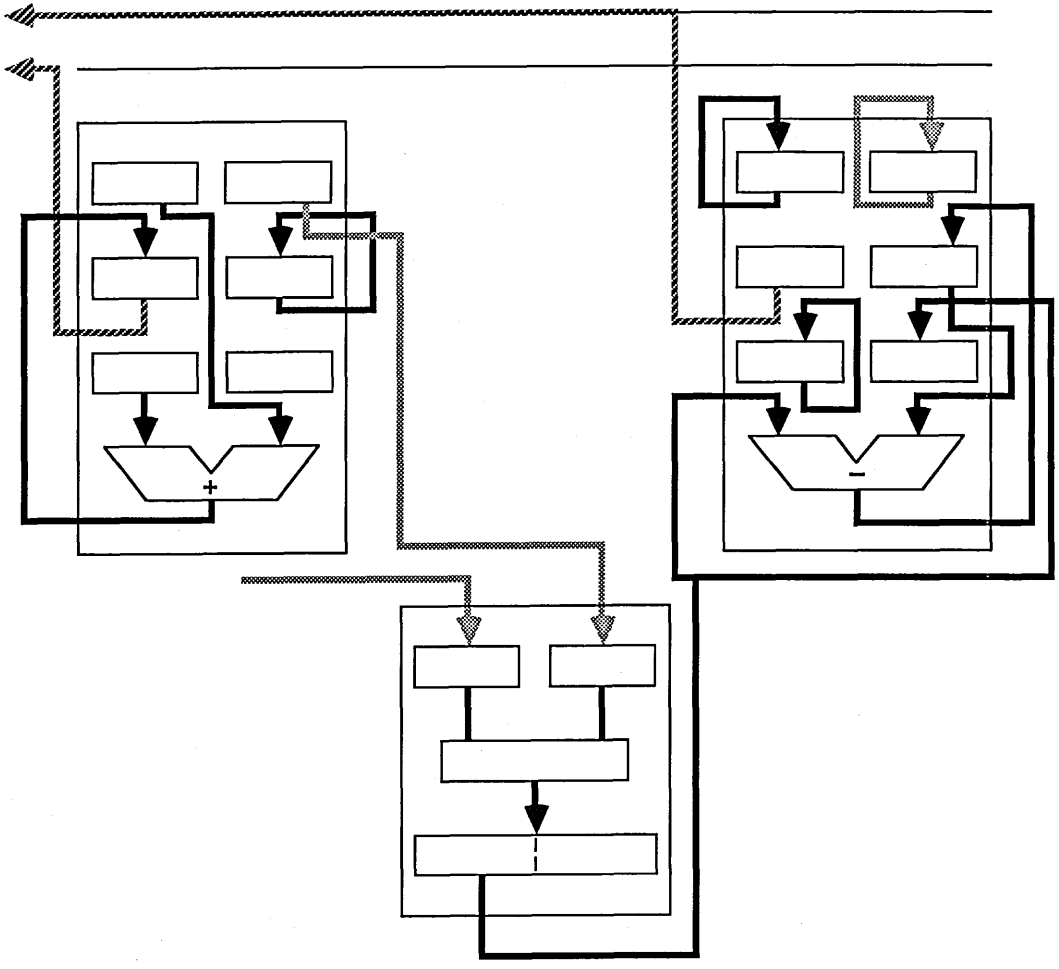


Figure 4-1.19 Three butterflies in progress at the same time: A′ for one BF is being written back. Arithmetic for next BF is in progress. The first product for the BF that is next again is being set up.

Table 4-10 Coefficient selection: work back from required products.

Step	FFT Addr Gen		D Addr Pipeline		RAM R/W	Register Enable	PROM SEL	DIO	Multiplier
	Instr	A SEL	Instr	SEL					Mult
1								X	
2							X		
3							X	X	X
4							X		X
5							X	Read B	X
6							Re		X
7							Im	Read A	B_RW_R
8							Re	X	B_RW_I
9							Im	X	B_IW_R
10							X	X	B_IW_I
11							X	X	X
12							X	Write B2	X
13							X		X
14								Write A2	X
15									
16								X	
17									
18								X	

Table 4-11 Coefficient pipeline: pick up 'k' from 29540 and hold for all four products.

	FFT Addr Gen		D Addr Pipeline						Multiplier	
Step	Instr	A SEL	Instr	SEL	RAM R/W	Register Enable	PROM SEL	DIO		Mult
1		X				X		X		
2						X	X			
3						X	X	X		X
4						X	X			X
5		8				En	X	Read B		X
6						H	Re			X
7						H	Im	Read A		B_RW_R
8						H	Re	X		B_RW_I
9		X				X	Im	X		B_IW_R
10						X	X	X		B_IW_I
11						X	X	X		X
12						X	X	Write B2		X
13							X			X
14								Write A2		X
15										
16								X		
17										
18								X		

Table 4-12 Data–address pipelining: needs 2-level push-down-only stack in an Am29520/21.

	FFT Addr Gen		D Addr Pipeline						Multiplier	
Step	Instr	A SEL	Instr	SEL	RAM R/W	Register Enable	PROM SEL	DIO		Mult
1		X		X	X	X		X		
2						X	X			
3				X	X	X	X	X		X
4						X	X			X
5		8		B1	R	En	X	Read B		X
6						H	Re			X
7				A1	R	H	Im	Read A		B_RW_R
8				X	X	H	Re	X		B_RW_I
9		X		X	X	X	Im	X		B_IW_R
10				X	X	X	X	X		B_IW_I
11				X	X	X	X	X		X
12				B2	W	X	X	Write B2		X
13							X			X
14				A2	W			Write A2		X
15										
16				X	X			X		
17										
18				X	X			X		

Table 4-13 Data-address pipeline: pipeline register instructions; 29540 address selection.

Step	FFT Addr Gen		D Addr Pipeline		RAM R/W	Register Enable	PROM SEL	DIO	Multiplier
	Instr	A SEL	Instr	SEL					Mult
1		X	X	X	X	X		X	
2		X	X			X	X		
3			X	X	X	X	X	X	X
4		1	Push B			X	X		X
5		8	H	B1	R	En	X	Read B	X
6		0	Push A			H	Re		X
7			H	A1	R	H	Im	Read A	B_RW_R
8		X	X	X	X	H	Re	X	B_RW_I
9		X	X	X	X	X	Im	X	B_IW_R
10		X	X	X	X	X	X	X	B_IW_I
11			X	X	X	X	X	X	X
12		X	X	B2	W	X	X	Write B2	X
13		X	X	X	X	X	X		X
14		X	X	A2	W	X	X	Write A2	X
15			X	X	X	X	X		
16				X	X	X	X	X	
17							X		
18				X	X			X	

Table 4-14 Full address generation code: for one butterfly; still to be overlapped.

	FFT Addr Gen		D Addr Pipeline						Multiplier	
Step	Instr	A SEL	Instr	SEL	RAM R/W	Register Enable	PROM SEL	DIO		Mult
1	X	X		X	X	X		X		
2	X	X	X			X	X			
3	COUNT			X	X	X	X	X		X
4	H	1	Push B			X	X			X
5	H	8		B1	R	En	X	Read B		X
6	H	0	Push A			H	Re			X
7	X			A1	R	H	Im	Read A		B_RW_R
8	X	X	X	X	X	H	Re	X		B_RW_I
9	X	X		X	X	X	Im	X		B_IW_R
10	X	X	X	X	X	X	X	X		B_IW_I
11	X			X	X	X	X	X		X
12	X	X	X	B2	W	X	X	Write B2		X
13	X	X		X	X	X	X			X
14	X	X	X	A2	W	X	X	Write A2		X
15				X	X	X	X			
16				X	X	X	X	X		
17							X			
18				X	X			X		

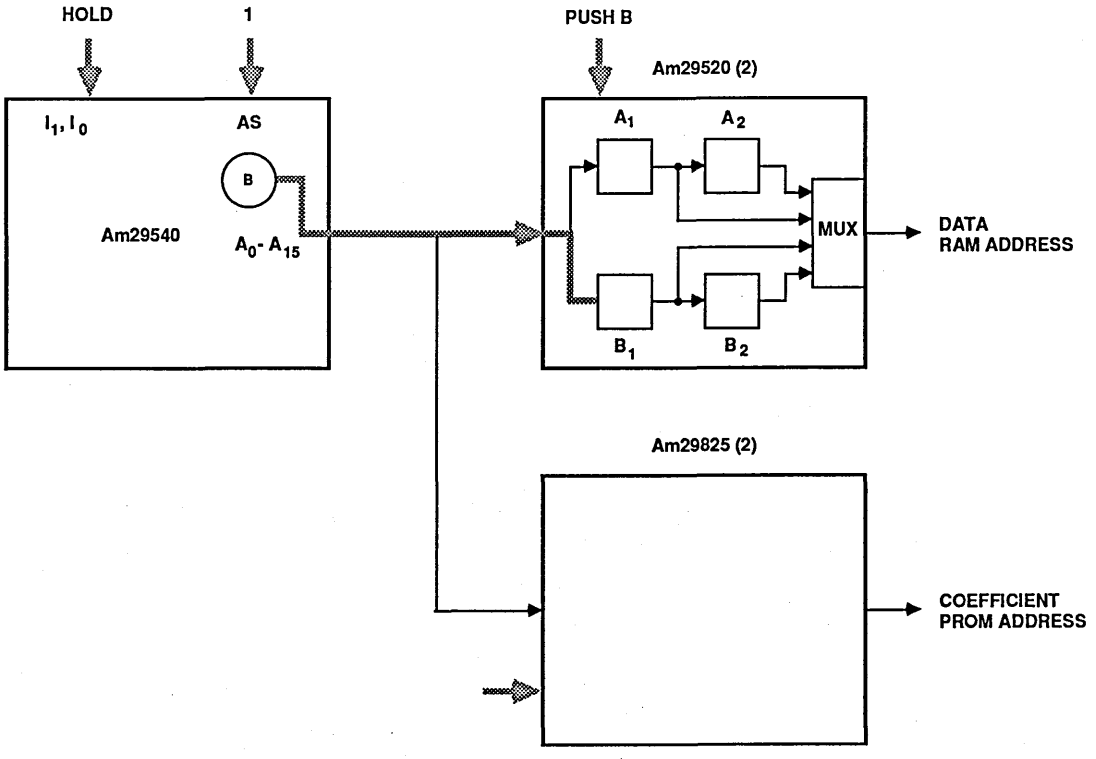

HOLD
1
PUSH B
Am29520 (2)
I_1, I_0
AS
A_1
A_2
B
MUX
DATA
RAM ADDRESS
Am29540
A_0- A_{15}
B_1
B_2
Am29825 (2)
COEFFICIENT
PROM ADDRESS

Figure 4-1.20

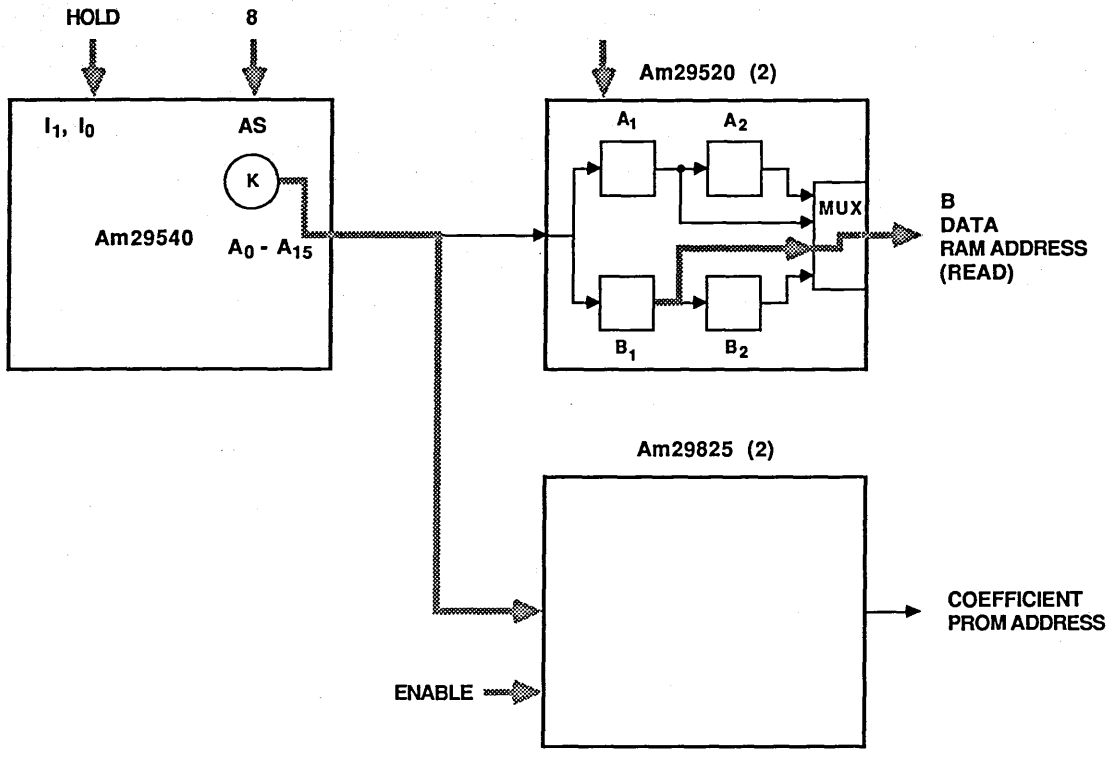

HOLD
8
Am29520 (2)
I_1, I_0
AS
K
Am29540
A_0 - A_{15}
A_1
A_2
MUX
B
DATA
RAM ADDRESS
(READ)
B_1
B_2
Am29825 (2)
COEFFICIENT
PROM ADDRESS
ENABLE

Figure 4-1.21

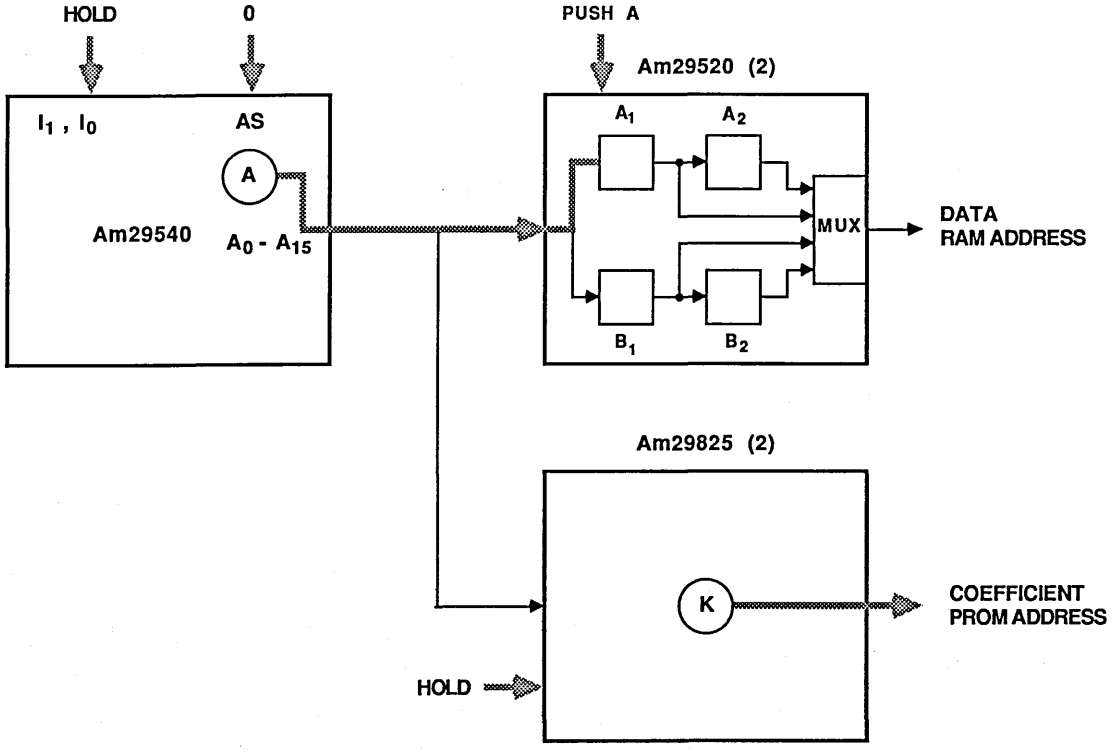

HOLD
0
PUSH A
Am29520 (2)
I_1 , I_0
AS
A
Am29540
A_0 - A_{15}
A_1
A_2
MUX
DATA
RAM ADDRESS
B_1
B_2
Am29825 (2)
K
COEFFICIENT
PROM ADDRESS
HOLD

Figure 4-1.22

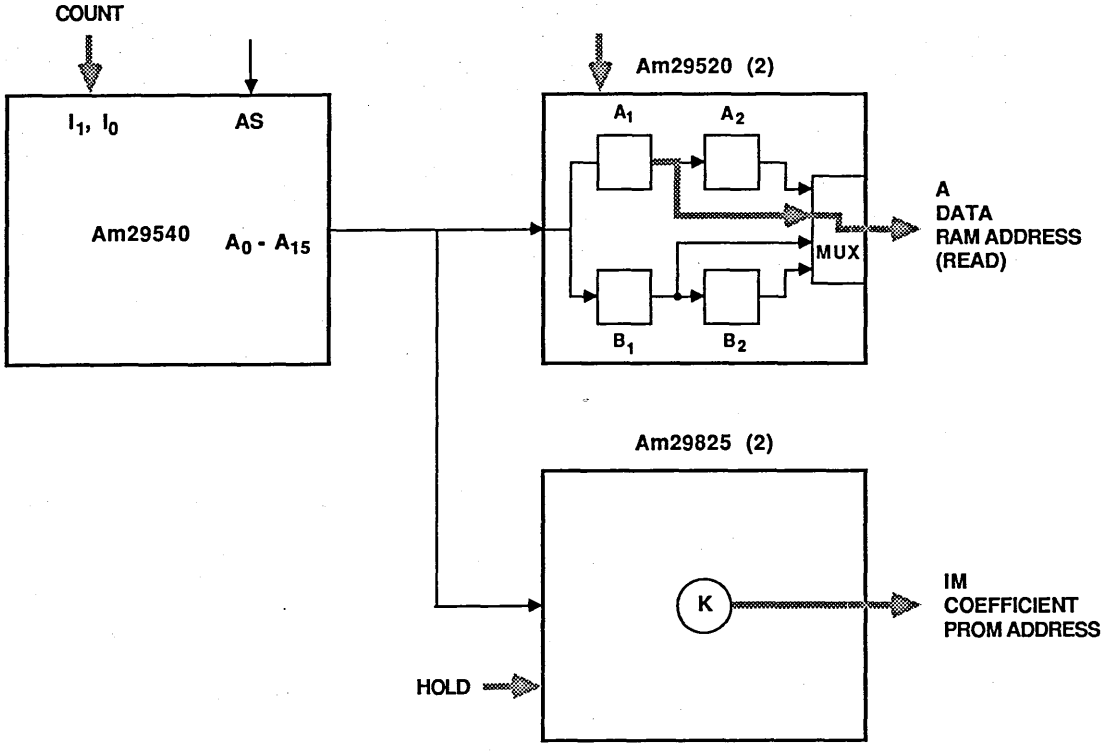

COUNT
I_1, I_0
AS
Am29540
A_0 - A_{15}
Am29520 (2)
A_1
A_2
MUX
B_1
B_2
A
DATA
RAM ADDRESS
(READ)
Am29825 (2)
K
IM
COEFFICIENT
PROM ADDRESS
HOLD

Figure 4-1.23

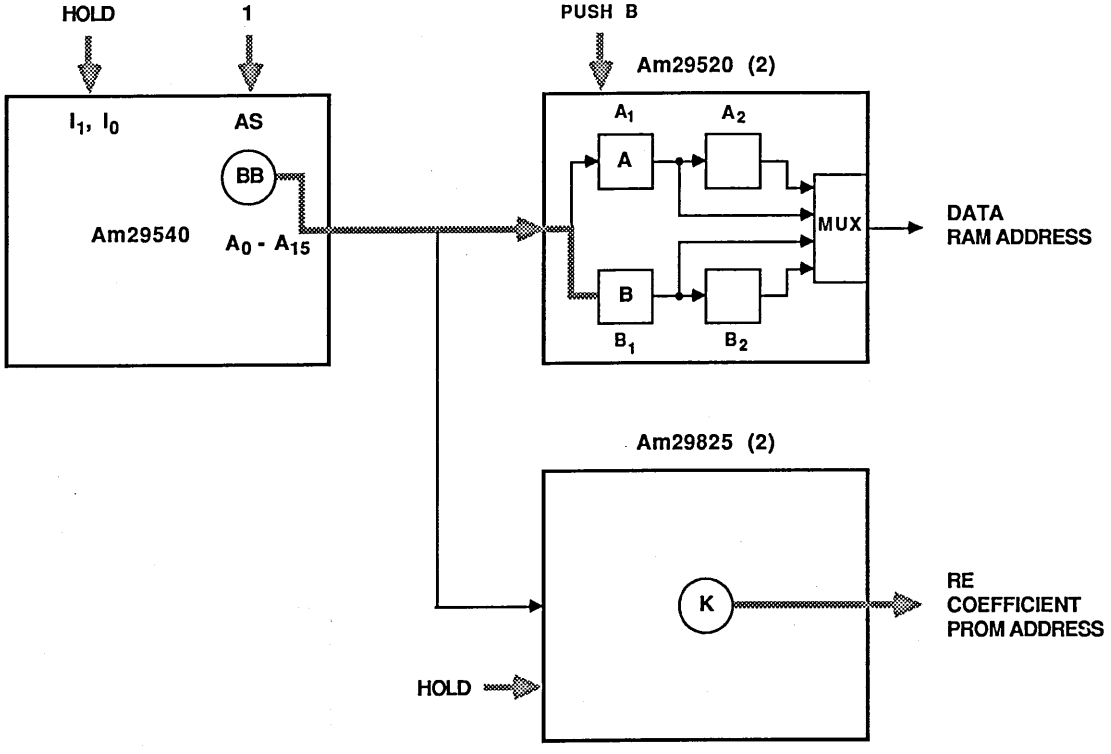

HOLD
1
PUSH B
Am29520 (2)
I_1, I_0
AS
BB
Am29540
A_0 - A_{15}
A_1
A_2
A
B
MUX
B_1
B_2
DATA
RAM ADDRESS
Am29825 (2)
K
RE
COEFFICIENT
PROM ADDRESS
HOLD

Figure 4-1.24

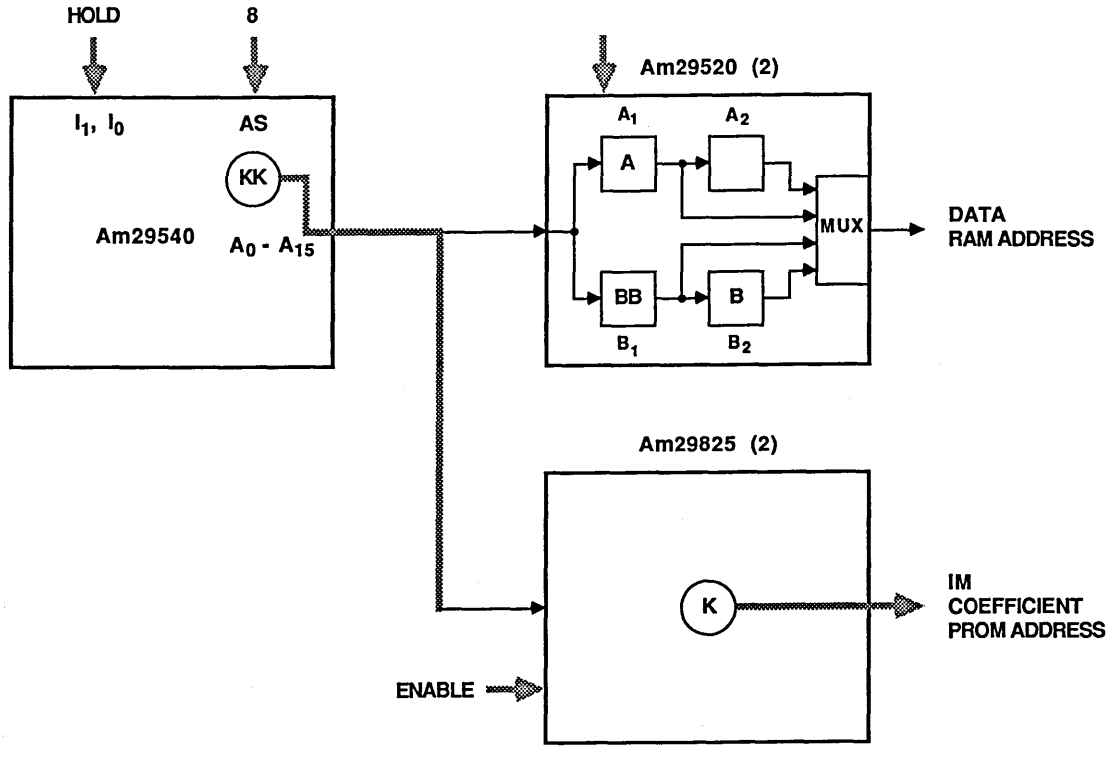

HOLD
8
Am29520 (2)
I_1, I_0
AS
A_1
A_2
KK
A
MUX
DATA
RAM ADDRESS
Am29540
A_0 - A_{15}
BB
B
B_1
B_2
Am29825 (2)
K
IM
COEFFICIENT
PROM ADDRESS
ENABLE

Figure 4-1.25

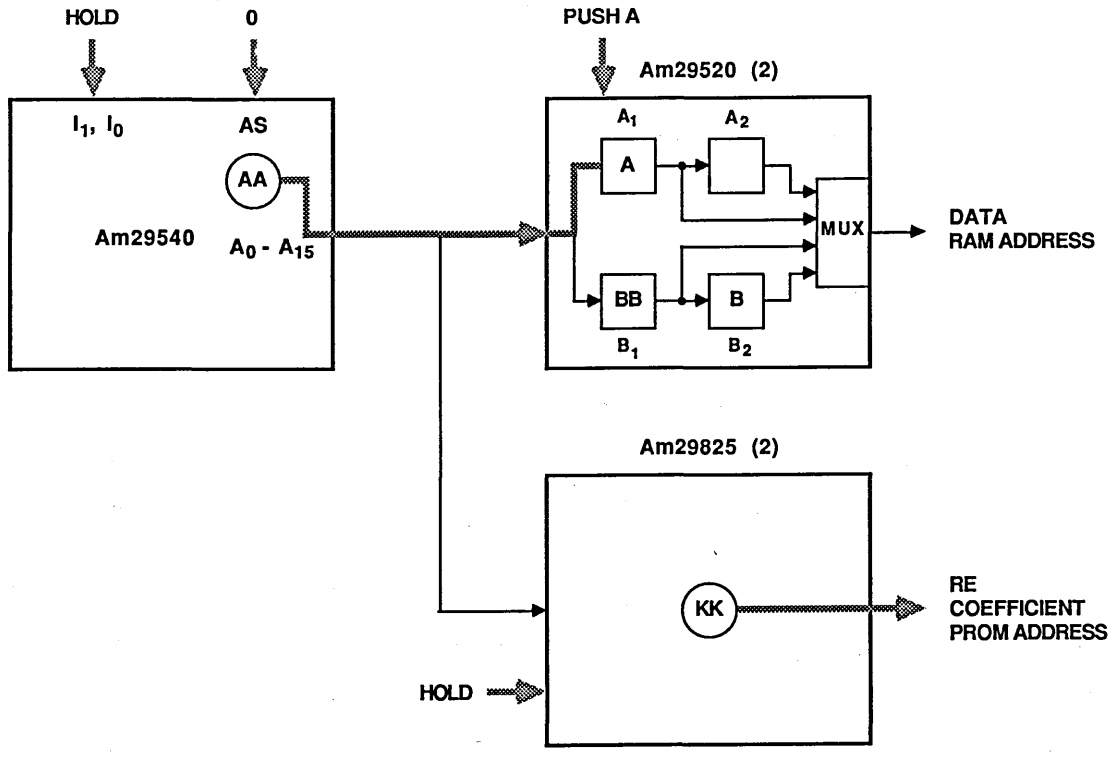

HOLD
0
PUSH A
Am29520 (2)
I_1, I_0
AS
AA
Am29540
A_0 - A_{15}
A_1
A_2
A
MUX
DATA
RAM ADDRESS
BB
B
B_1
B_2
Am29825 (2)
KK
RE
COEFFICIENT
PROM ADDRESS
HOLD

Figure 4-1.26

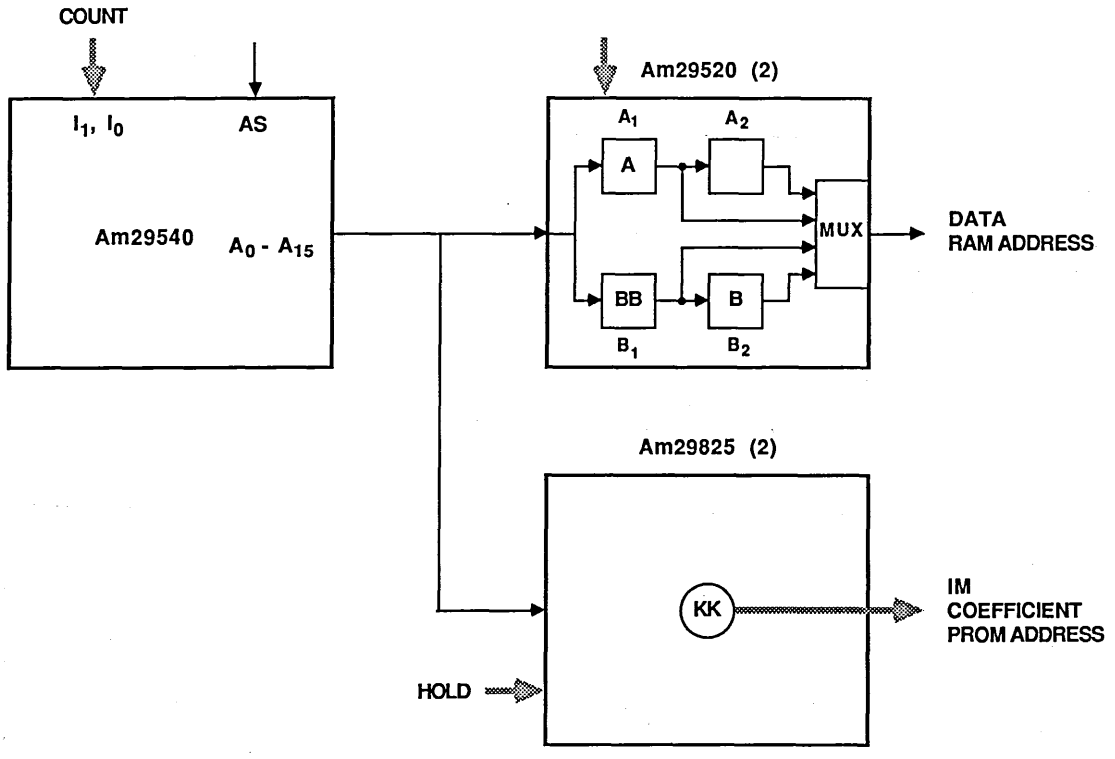

COUNT
I1, I0
AS
Am29540
A0 - A15
Am29520 (2)
A1
A2
A
MUX
DATA
RAM ADDRESS
BB
B
B1
B2
Am29825 (2)
KK
IM
COEFFICIENT
PROM ADDRESS
HOLD

Figure 4-1.27

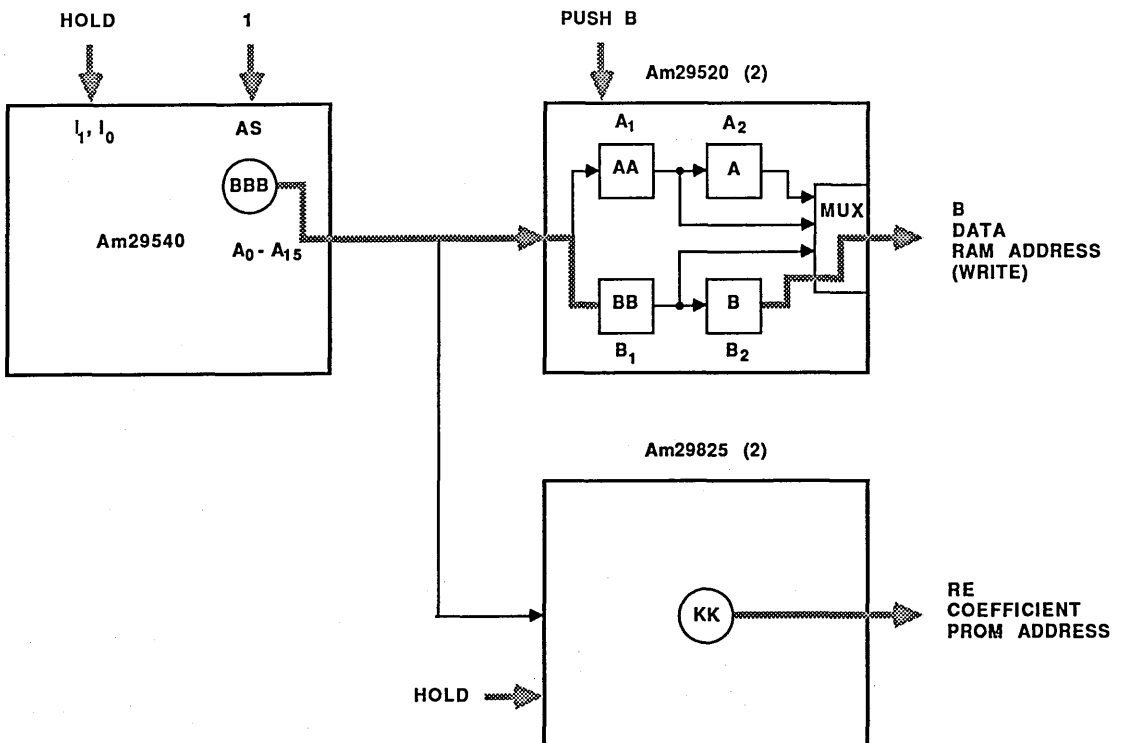

HOLD
1
PUSH B
Am29520 (2)
I_1, I_0
AS
A_1
A_2
AA
A
BBB
MUX
Am29540
$A_0 - A_{15}$
B
DATA
RAM ADDRESS
(WRITE)
BB
B
B_1
B_2
Am29825 (2)
KK
RE
COEFFICIENT
PROM ADDRESS
HOLD

Figure 4-1.28

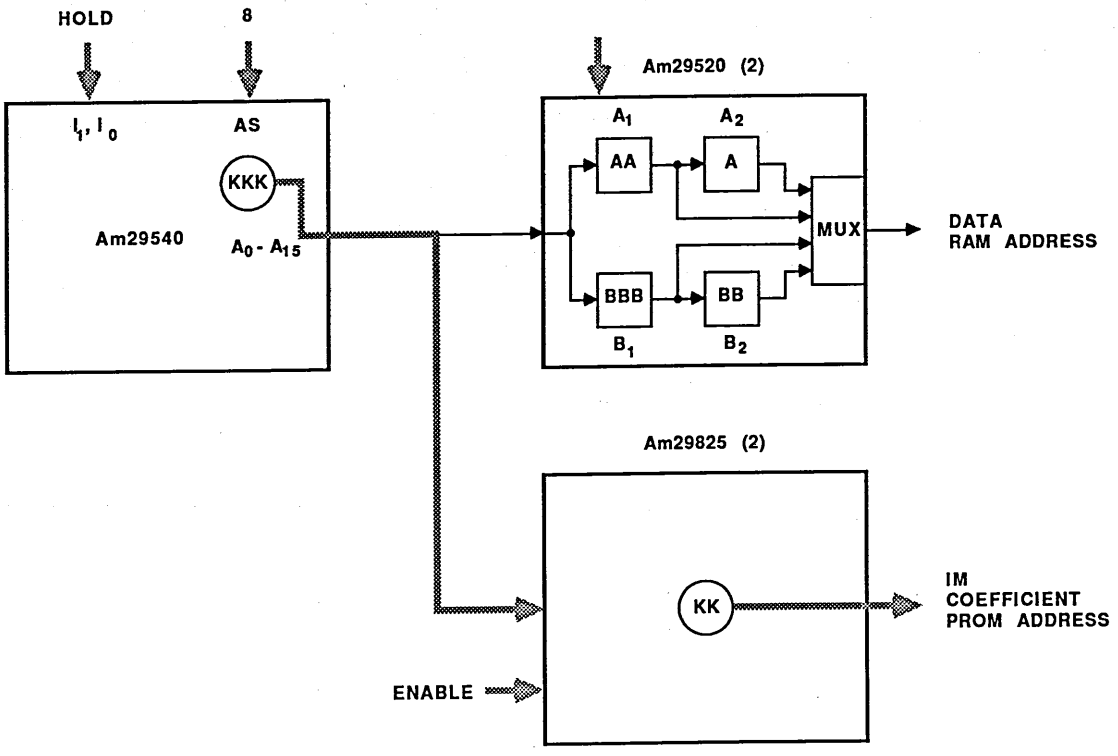

HOLD
8
Am29520 (2)
I_1, I_0
AS
A_1
A_2
AA
A
KKK
MUX
DATA
RAM ADDRESS
Am29540
$A_0 - A_{15}$
BBB
BB
B_1
B_2
Am29825 (2)
KK
IM
COEFFICIENT
PROM ADDRESS
ENABLE

Figure 4-1.29

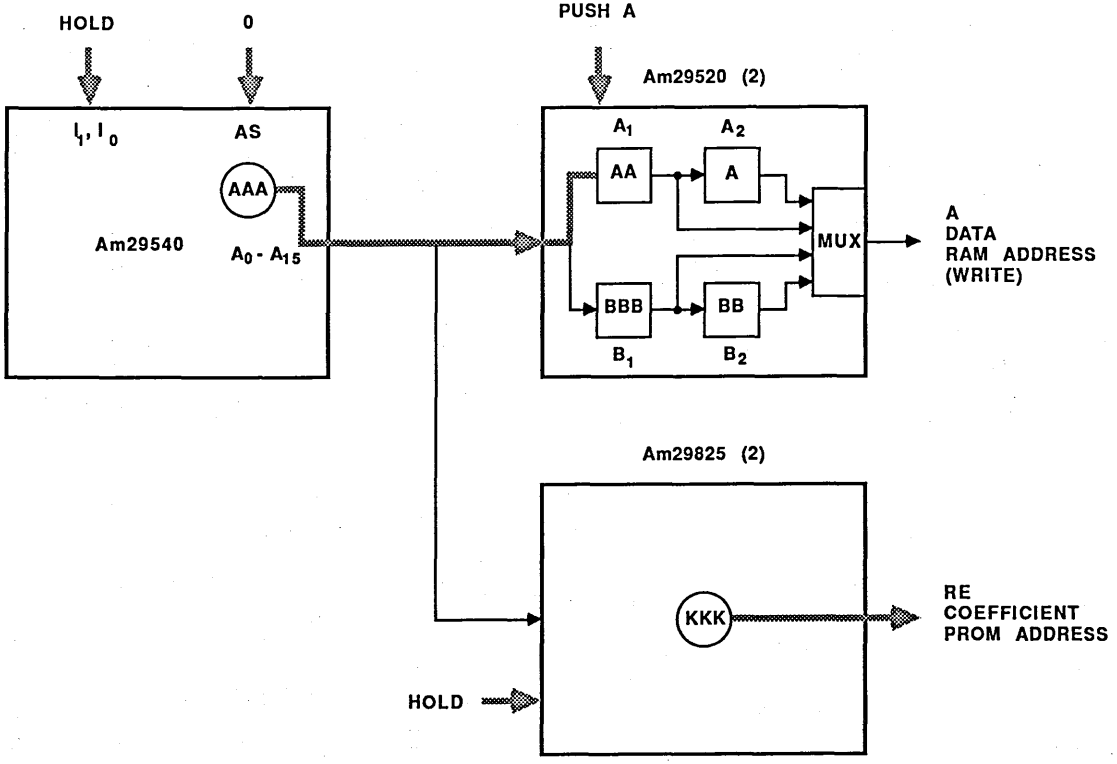

HOLD
0
PUSH A
Am29520 (2)
I_1, I_0
AS
A_1
A_2
AAA
AA
A
MUX
A
DATA
RAM ADDRESS
(WRITE)
Am29540
A_0 - A_{15}
BBB
BB
B_1
B_2
Am29825 (2)
KKK
RE
COEFFICIENT
PROM ADDRESS
HOLD

Figure 4-1.30

c) The format definition for each instruction. It consists of defining the order in which the value of each field is given while writing the microprogram.

d) The second phase consists of writing the microprogram using the defined language to create a source file (Appendix 3, FFT.SRC).

Another meta-assembler is MACASM by Microtec. Further queries can be directed to: Microtec Research, Inc., 3930 Freedom Circle, #101, Santa Clara, CA 95054, (408) 733-2919.

Table 4-9 shows that the four lines of code for the butterfly are repeated. These are put in a loop and the condition for exiting from the loop is the FFT complete flag from the address sequencer. Some code is required to fill the pipeline and some to flush the pipeline at the end of the process. The following is the sequence of events that would occur from power up.

On power up when the system is reset, the opcode for the JZ instruction is applied to the sequencer as described in the control section. This causes the sequencer to branch to location 0 of the microcode. The op-code for the sequencer at this location is a jump to location 1. This is done to enable a development system to jam address zero on the address bus to emulate the JZ instruction. The PC in the sequencer is updated to the externally produced address by the jump instruction. At location 1, the sequencer waits for the load IR signal to go active. When this happens, the sequencer goes to location 2 and waits for the signal to go inactive. When this happens, the sequencer goes to location 3 and jumps to 1 of 8 locations at the end of the microcode memory. At this location, the start address for the process is obtained and the sequencer jumps to this location. The process is completed and the process complete flag is set. When the CPU acknowledges this flag, the sequencer returns to location 1 and waits again.

The schematics for this design are included in Appendix 6.

4.2 DIGITAL FILTERS USING MULTIPLY-ACCUMULATOR

A high speed stand-alone digital filter is frequently implemented with a multiplier/accumulator, temporary storage and a state machine which together perform the following calculation

$$y_n = \sum_{k=0}^{M} a_k * x_{n-k} - \sum_{k=1}^{L} b_k * y_{n-k}$$

where xk is a digitized time sample and yk is the output of the filter. This arithmetic can be performed in real time by a microprocessor or single chip signal processor for low frequency applications but a specialized design is required when the input signal frequencies are above 100 kHz (requiring data sampling at >200 kHz). The Am29PL141 Fuse Programmable Controller (FPC) simplifies the design of microcode controlled filters by incorporating all the control functions in a single chip. It contains a microprogram address sequencer and 16 outputs for control of the other circuits in the system. In a digital filter design these outputs manage the temporary storage of previous inputs and outputs and steer the operands to a multiplier/ accumulator.

Figure 4.2.1 is an example of a second order IIR filter (also called a biquadratic filter) using an Am29PL141 FPC, an Am29510 multiplier/accumulator (MAC), 2–Am29520 multilevel pipeline registers, 2–Am27LS19 PROMs and 2–Am29827 buffers. The Am29510 consists of a 16x16-bit parallel multiplier with a 35-bit accumulator. The Am29520 is a set of four 8-bit pipeline registers which can be configured as two shift registers for this design. Any one of the four registers can be selected at the output of the Am29520. The Am27LS19 PROMs are used to hold the coefficients which determine the characteristics of the filter. The negated coefficients are stored for W3 and W4 to simplify the accumulation and one of four sets of coefficients can selected by strapping two address inputs to the PROMs.

The design implements the equation shown above directly although the sequence of calculations is done using the oldest data first for purposes of data management. Appendix 4 is a meta-assembler definition file for the Am29PL141. It defines a set of custom program flow instructions for the filter and also defines the control functions for the output pins. Appendix 5 is the assembly source file for the program which uses the instructions defined in the previous listing. Another way to use a meta-assembler is to define a standard set of device instructions (FPC instructions for this design) instead of IIR filter instructions in the definition phase and to keep the customization in the assembly program. This has advantages in a complex design since it eliminates one pass of the meta-assembler when changes or corrections are made and the program is reassembled.

The operation of the filter consists of receiving data from an A/D or a similar filter section, performing the sum of products required for the filter and sending the output to a D/A converter or another filter section. There is signalling for input data taken and output data ready but synchronous operation is assumed and there is no "handshaking" where the device waits until data is ready or taken. This was done because several sections of this type would normally be cascaded as shown in Figure 4-2.2 to obtain the desired frequency response. The signalling enables the sections to synchronize during powerup and could be used for diagnosing faults which cause the system to fall out of synchronization or to resynchronize after such a fault. The sample program simply sets an error flag and stops if data is not available when needed or taken when the calculation is complete. Since only a fourth of the PROM space is needed for the filter, a strapping option can be used for the coefficient addressing to allow a single design to be used for up to four cascaded sections. A filter constructed out of these sections requires five clock periods to produce an output from each input data sample so 10 MHz parts are capable of operating at a 2 MHz sampling rate and can handle input frequencies up to 1 MHz. These figures can be scaled linearly when faster or slower parts are used.

Data management in the Am29520 is accomplished by configuring the device into two shift registers each containing two levels. The A registers hold the two previous input samples and the B registers hold the two previous outputs. The tri-state buffers on the input isolates it while the output data is routed to shift register B. Two FPC outputs are used to control the complementary tri-state enables in order to eliminate the need for an inverter.

A variation of this design could be used for a higher order filter by substituting a RAM or register file for the Am29520s. An example of a sixth order IIR filter is shown in Figure 4-2.3. This example is programmed to handshake on input and output since it is not a section of a cascaded filter. In order to obtain a NOP in the MAC during the handshaking, a zero is stored in the coefficient PROM and a zero product is accummulated while waiting for the input data ready or output data taken handshake. In the previous example it was possible to route operands to the MAC on every cycle since the registers could input and output simultaneously. However, this variation uses two cycles during which no calculations take place to store input and output values in the RAM. These cycles can be reclaimed to increase the throughput by adding hardware to route data to the MAC while writing into the RAM or by selecting write-transparent RAMs which place the data being written on the output during the write cycle. However, this would only be of value if the data flow was synchronous and the input and output handshaking loops could be removed.

The two shift registers required for the filter are emulated by a RAM in which logical addresses represent the position in the shift register. A counter and an adder are used to translate the logical address to a physical address for the RAM. Reading an input data sample increments the counter and "shifts" the data in the RAM. With a minimum of handshaking time, this filter is capable of sampling data at a 600 kHz rate and filtering data containing frequencies up to 300 kHz when operated at 10 MHz.

Because RAM sizes are available in discrete steps, the direct implementation of a sixth order filter is an acceptable choice. However, the 16 RAM storage locations can be used more efficiently if the canonical form of the IIR filter is implemented. This alternate form of the filter can be used whenever the number of previous inputs in the calculation is equal to the number of previous outputs, i.e., M = L for the summation limits. The circuit shown in Figure 4-2.3 can be used to implement an order-15 filter by changing the FPC program to store intermediate calculation values instead of inputs and outputs. The equations to be calculated become

$$Z_n = X_n + \sum_{k=0}^{M} a_k * Z_{n-k}$$

and

$$y_n = \sum_{k=0}^{M} b_k * Z_{n-k}$$

The intermediate value z is calculated from the input and previous values of z and then entered in the shift register. The output is then calculated from the values of z in the shift register. A zero coefficient is required to allow the MAC to hold an output value during handshaking as in the previous example. These number of calculations required limit the filter to a sampling rate of 300 kHz.

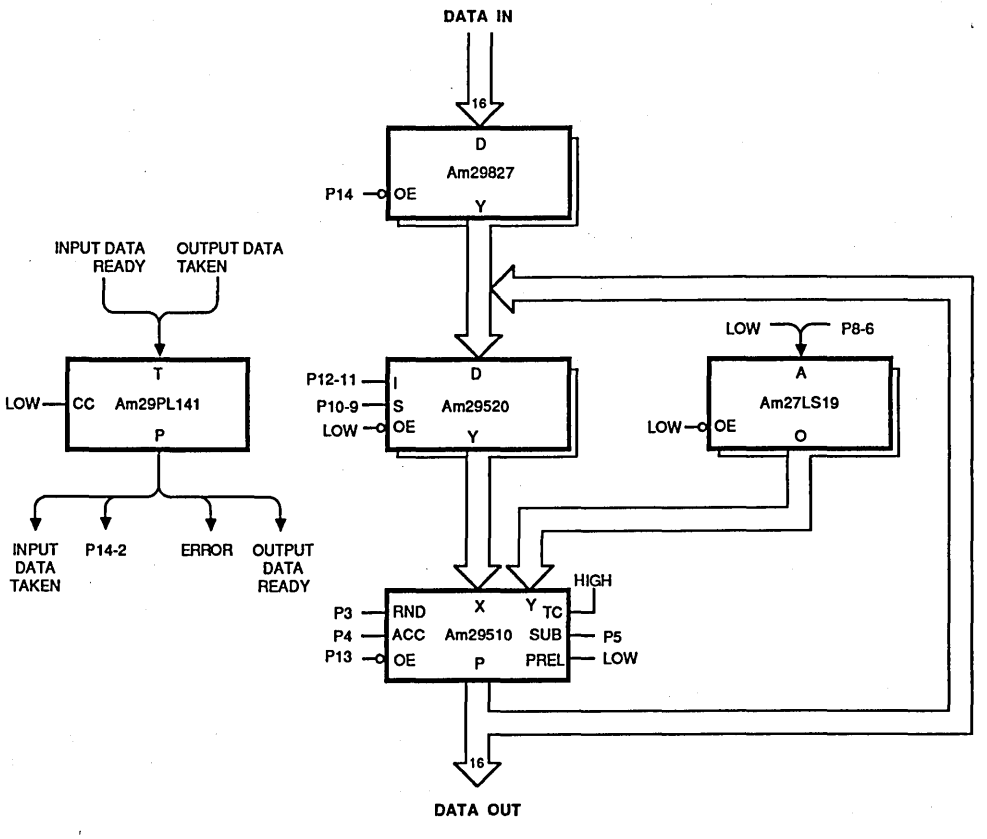


Figure 4-2.1

Figure 4-2.2

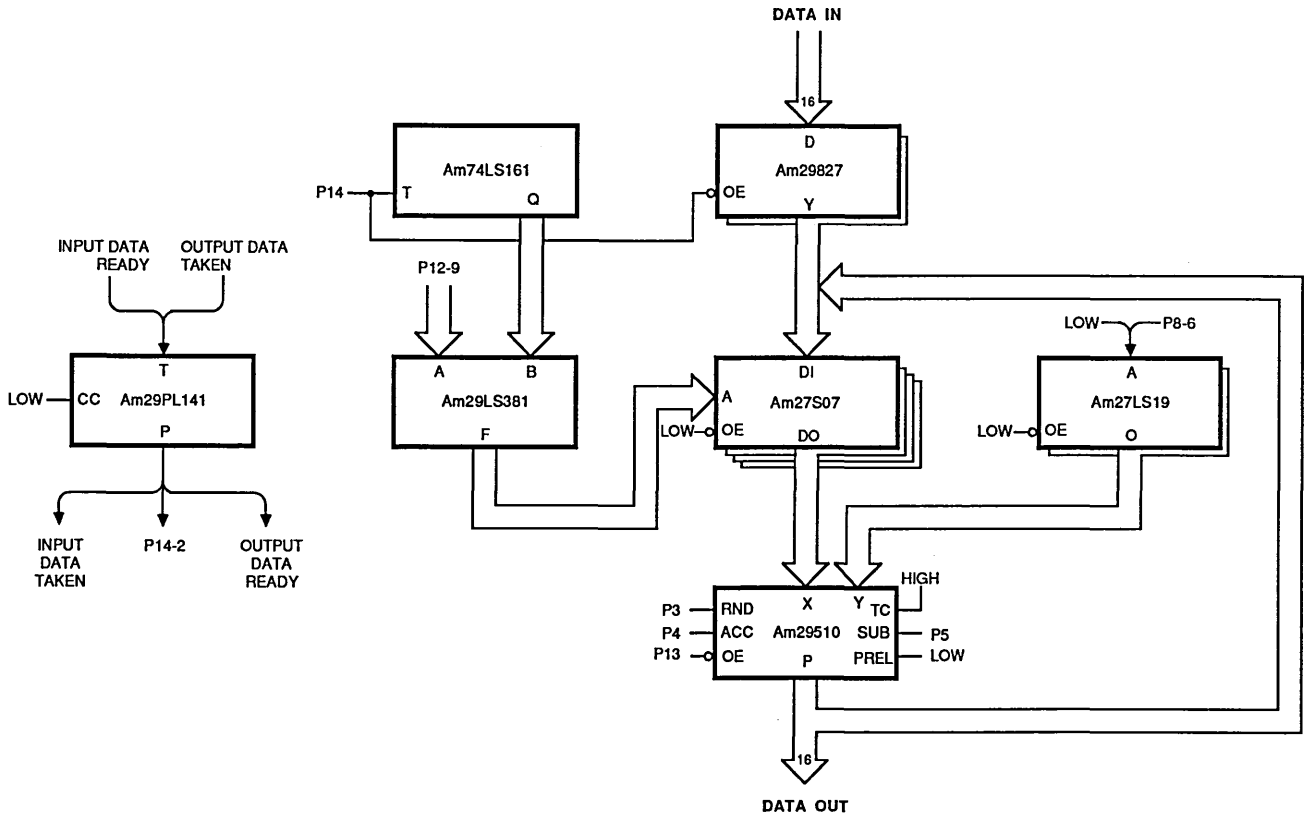

DATA IN
16
D
Am29827
OE
Y
Am74LS161
P14
T
Q
INPUT DATA READY
OUTPUT DATA TAKEN
P12-9
T
LOW
CC
Am29PL141
P
A
B
Am29LS381
F
DI
A
Am27S07
LOW
OE
DO
LOW
P8-6
A
Am27LS19
LOW
OE
O
HIGH
P3
RND
X
Y
TC
P4
ACC
Am29510
SUB
P5
P13
OE
P
PREL
LOW
INPUT DATA TAKEN
P14-2
OUTPUT DATA READY
16
DATA OUT

Figure 4-2.3

Record signal-processing rates spring from chip refinements

Improved buses, reconfigurability, pipelining, and parallelism unite in a bipolar family for building array and signal processors

by Bernard New and Lyle Pittroff, *Advanced Micro Devices Inc., Sunnyvale, Calif.*

☐ The number-crunching microprocessor requirements of the 1980s are ill-served by today's comparatively slow, conventional central processing units. Instead, the algorithms executed by both general-purpose array processors and the more specialized digital-signal processors require highly individual architectures for maximum speed and performance. Jumping on the fast track is a new group of bipolar devices—the AM29500 family—that combines internal emitter-coupled-logic circuit design for speed with TTL outputs for compatibility with the outside world.

The family is able to overcome such speed-retarding problems as inadequate data-bus memory and bandwidths and slow execution times through a redesigned bus structure and parallel and pipelined processing. In fact, the bus structure is designed so that there are enough parallel buses to keep a device's multiplier or its arithmetic processing unit, or both, busy during each cycle. These features, plus programmable reconfigurability, make the 29500 family the fastest group of large-scale integrated parts for signal processors to be commercially available. In one series of tests, a 29500-based system had three times the speed achieved by the older 2900 family.

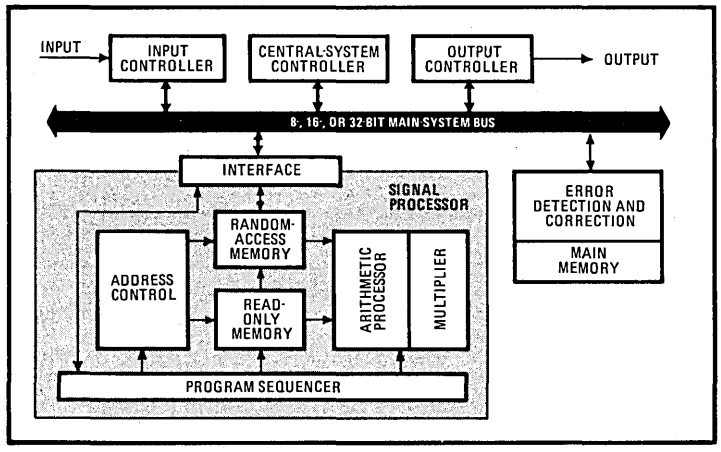
The 29500 series are general-purpose building blocks. They include a byte-slice, multiple-port programmable signal processor (the 29501), a 16-by-16-bit parallel multiplier with programmable input/output (the 29516/17), a multilevel pipeline register for data and address pipelining (the 29520/21), and a fast-Fourier-transform address sequencer (the 29540).

To increase processor speed, architectural enhancements had to be made to the older 2900 device designs. That family took some steps in the right direction because it provides many of the peripheral building blocks, like interface devices and direct-memory-access chips, needed for real-time signal processing. But the 2900's arithmetic devices are targeted at general-purpose computing. They do not have the parallel channels that are required for a high-speed array or signal processor environment.

One way of satisfying this need was to upgrade the 2900 family's bus structure, number organization, and resource management. The new bus structure can support addition or subtraction and multiplication on every cycle because of extra parallel buses. Number organization can now handle complex numbers in parallel quickly. In addition, flexibility of resource management permits the building blocks to be interconnected in enough ways to support all algorithms of interest efficiently.

For dedicated-function and multiple-algorithm processing (Fig. 1), a special-purpose processor like the 29501 operates under the control of a host computer system that switches large blocks of data between its main memory and temporary slave through DMA transfer. Once this transfer is complete, the special-purpose processor operates under local program control. Each algorithm is executed by its own software routine, which is stored in its own local memory independently of the host computer and its high-level language.

Although the precise architecture of Fig. 1 varies with the algorithm used, all array- and signal-processing algorithms have similar needs for



1. Dual-purpose. In a typical array- or digital-signal-processor architecture, both dedicated and multiple algorithm functions can be implemented. A host computer provides overall guidance and a large memory.

writing it into RAM locations 000000 to 000007.

When the first four memory cycles are over, U_3 goes into the high state and the decoding occurs. The other half of U_2 furnishes a switch-reset pulse when the system has stabilized. This 11-microsecond pulse sets the processor but does not clear the register. Thus, for all reset conditions set by the U_2-based switch, the vectors must be fetched from the RAM, thereby allowing the operating system to alter them. □

Designer's casebook is a regular feature in *Electronics*. We invite readers to submit original and unpublished circuit ideas and solutions to design problems. Explain briefly but thoroughly the circuit's operating principle and purpose. We'll pay $75 for each item published.

Generating a negative voltage for portable instruments

by J. D. McK. Watson, *Biomedical Engineering Research Group, University of Sussex, Falmer, Brighton, UK*

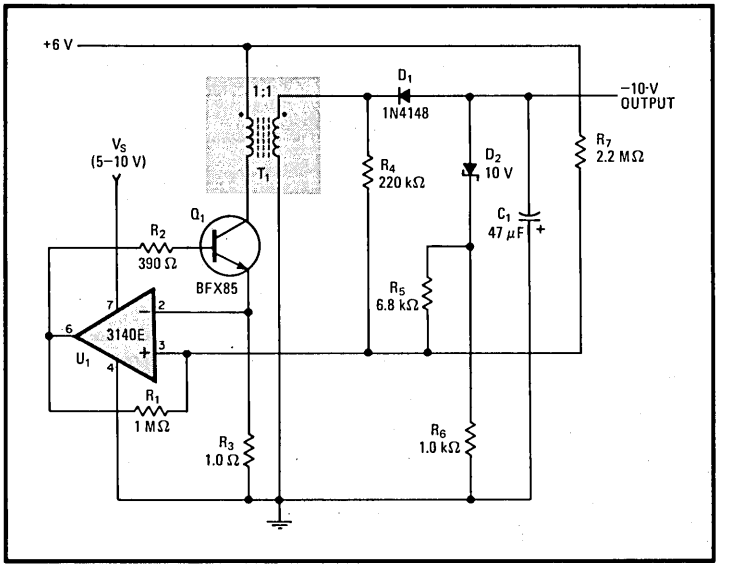
Many recently designed microcomputer-based portable instruments require +5- and −10-volt dc supplies. Though +5 V can be readily derived from a battery supply by means of a linear regulator, the latter needs a special circuit. This flyback converter presents a novel power supply design that uses just one operational amplifier and a few discrete components. The circuit efficiency is about 75% for a load of about 10 milliamperes, and the output voltage can be changed by substituting an alternative zener diode.

Operational amplifier U_1 functions as a current-sensing threshold switch and is capable of providing a wide output-voltage swing. This threshold is adjusted for optimum supply efficiency and output-voltage regulation. Q_1 is driven by the output of U_1 and operates as a saturating switch, with pulse transformer T_1 functioning as its collector load. The transformer is designed for a turns ratio of 1:1 with primary and secondary inductance of 3 millihenrys and a resistance of 1 ohm.

The current in T_1's primary through Q_1 provides a signal to the inverting input of U_1 whose noninverting input is fed from three sources. A portion of the op amp's output provides positive feedback to ensure fast switching, an ac signal from T_1's secondary results in the collapse of the flux before recycling, and a dc component tapped from the output lowers the threshold when the output exceeds zener diode D_2's breakdown voltage.

When the circuit is switched on, U_1 delivers a high output to Q_1 and turns it on. Current in the primary of T_1 increases linearly, developing a positive voltage at its secondary. This rising primary current also creates a voltage at the inverting input of U_1 that is sufficient to turn it off. As a result, the flux in T_1 collapses and the secondary current charges capacitor C_1. During this energy transfer, R_4 holds the noninverting input negative and inhibits the switch from turning on.

As subsequent cycles add charge to C_1, a point is reached when D_2 conducts and inhibits U_1 through R_5. This stage is disabled until the dc output voltage falls below the zener threshold, whereupon the circuit resumes oscillation. The amplitude of the output voltage is approximately equal to the zener voltage of D_2. Because of the nonlinear method of regulation, a small amount of ripple is superimposed on the output. For the component values shown, the ripple is of the order of 40 millivolts, but can be reduced by using a RC filter network at the output. Maximum power output is limited by the supply voltage and by the saturation current of T_1. □



Flyback converter. This novel flyback converter uses just one op amp, U_1, pulse transformer T_1, and a few discrete components to provide a −10-V dc voltage. The supply ripple contents are low and the circuit efficiency is approximately 75%. Zener diode D_2 is used to set the output voltage.

arithmetic and addressing—short, repetitive calculation loops requiring parallelism and pipelining. In addition, in digital-signal processing, arithmetic operations using complex numbers may be necessary, whereupon the computational load increases to twice as many additions or subtractions and four times as many multiplications as for real numbers.

Because calculation loops for arithmetic operations are short, the 29500 family surrounds the additions with continuous memory accesses—data is fetched, the calculation loop performed, and the results written back into memory. Hence there are many times more memory accesses than there are data points. For FFTs, the number of repetitive memory accesses is multiplied by the number of passes through the data. Fortunately, although the memory-access sequence is long, it is well structured, making it possible as a result to design dedicated address sequencers.

Divide and rule

The purpose of pipelining is to allow lengthy operations to be divided into suboperations, so that when one piece of data has completed a suboperation, the same hardware can start on the next piece. In this way, the 29501 allows up to a 500% speed improvement.

For example, because a typical processor handles a set number of algorithms, its architecture can be very specific concerning arithmetic and address generation—no longer does the CPU have to mix addressing with arithmetic computations. Also, separate sections can be streamlined to calculate each type in parallel and fast.

A significant feature of the data path for the 29500 family is the fact that the devices handle only data and do no address calculations. The data path can, therefore, be optimized for arithmetic.

The 29501 multiport parallel processor also represents the current thinking about multiport organization. It has a data-bus port, an output port to a multiplier, and an input port from a multiplier. The chip can process an FFT fast because of its highly parallel internal bus structure. In this structure, six registers operate as pipelines and are connected to the I/O ports and an arithmetic and logic unit by 10 separate byte-wide internal buses.

A typical cycle on the 29501 consists of data input from memory, data output to the multiplier, retrieving a previous product from the multiplier, and register-to-register ALU operations and data moves. Because these operations can occur during the same cycle, data manipulation is limited only by the designer's creativity. This flexibility, plus the possibility of parallel processors operating on complex numbers, is what makes high-speed operation possible.

Twice as fast

The 29500 family uses two high-speed parallel 16-by-16-bit multipliers—the 29516 and 29517. The 29516 is compatible with TRW's MPY-16HJ multiplier but is more than twice as fast and has an output multiplexer. Either the least or the most significant product can be selected at this multiplexer output for use in many pipelined architecture calculations.

On the other hand, the 29517 multiplier incorporates all the features of the 29516 but has a modified I/O-register clocking structure to provide a single-clock input with register enables. This approach is preferred to the older clock-gating method, which suffers from skews.

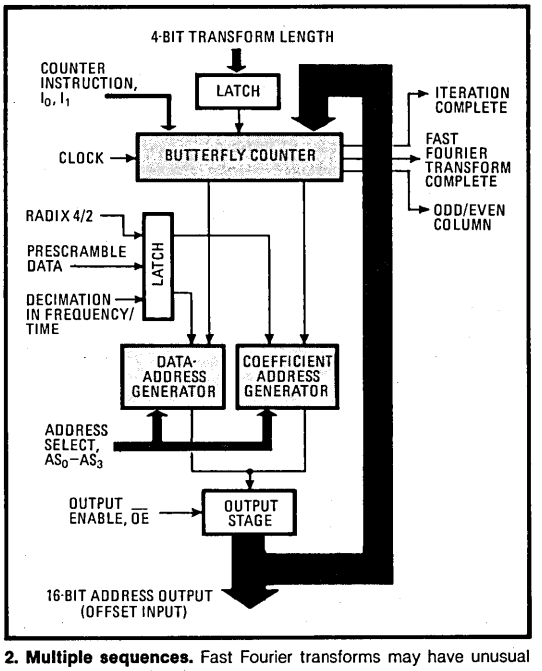
Dedicated addressing

Address-sequencing complexity for array and signal processors can range from integer counting to the complicated number patterns of FFTs. To keep addressing speeds high, the 29500 series generates addresses in parallel to the data path. However, other architectural considerations must also be weighed.

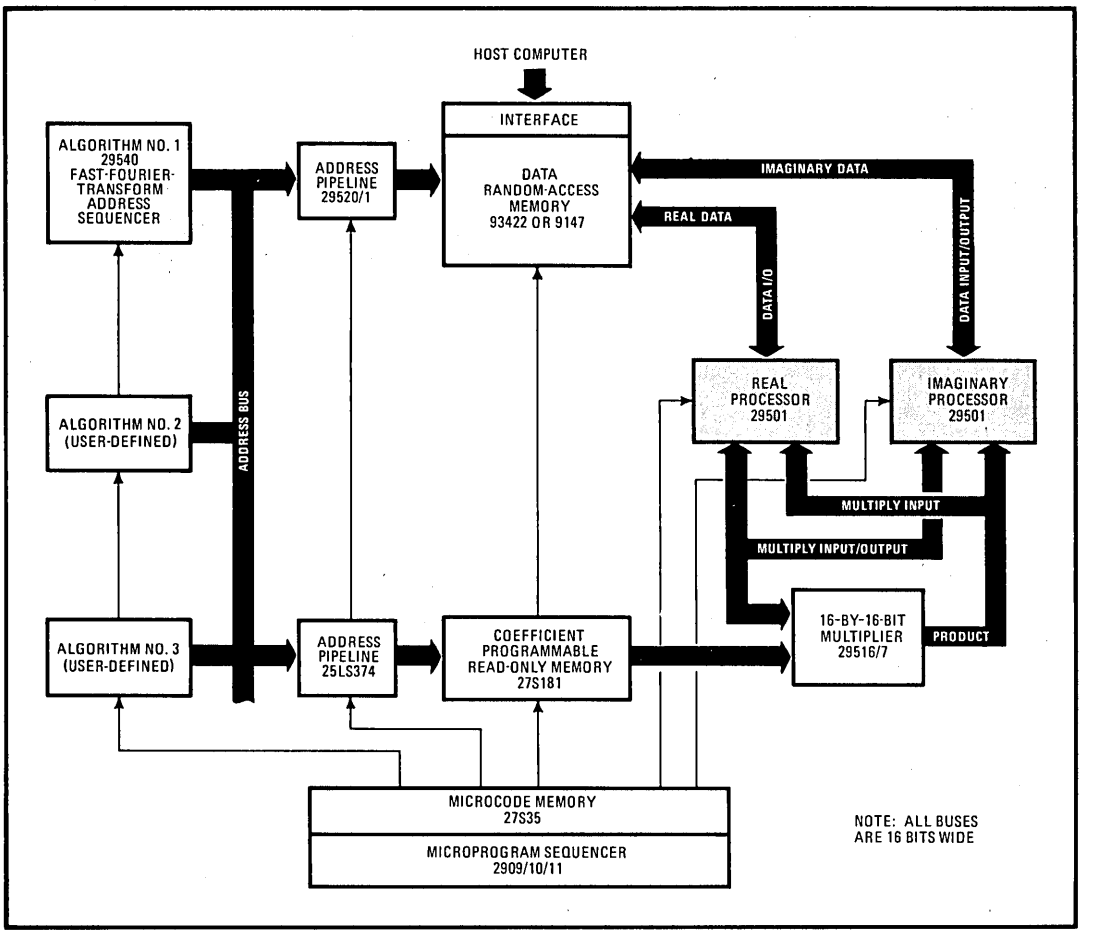
For a specific application, several system implications affect the choice of algorithm from the diversity of FFTs available. This choice, together with the transform length (or lengths) to be implemented, determines the address sequence to be generated. Usually, the nested-count nature of these sequences has forced the designer to use many medium-scale integrated-circuit packages.

The 29540 is a single-chip solution to the address-sequencing problem (Fig. 2). Four control inputs allow programmed or hardwired control of the actual number of data points in the transform. From this and other control-input commands, the 29540 can be sequenced through the entire transform while providing output flags. These flags indicate when each data pass is over and when the entire transform is complete.

For their part, the 29540's control inputs accept the most common FFT formats. The designer can opt for bit-reversed output order or bit-reversed input order, radix-2 or radix-4 address sequences, and decimation-



2. Multiple sequences. Fast Fourier transforms may have unusual address sequences, and with its four control inputs, the address-sequencing 29540 chip is designed to handle all of them. It provides output flags when a calculation is complete.



3. Complete. A typical signal-processing system provides separate, parallel paths for complex data. But in the 29500 setup, address pipelining handles both data and coefficient addressing operations for fast Fourier and other common transforms.

in-frequency or decimation-in-time sequences.

The 16-bit output port of the address sequencer is controlled by the counter and transform-length–input instructions. Any transform from 2 to 65,536 points long can be selected. The higher-order bits not required for the specified transforms (a 1,024-point transform only requires 10-bit addresses) can be preloaded through a bidirectional address port to access the next data block.

Easy address pipelining

Because the primary objective of this architecture is to operate on array- or signal-processor systems in a highly parallel manner, addresses must also be pipelined. As a result, each address must be tracked, which requires a pipeline register—such as the 29520 or 29521. These are byte-slice pipelining registers configurable as a dual two-level or a single four-level pipeline. In both devices, the single four-level configuration operates as a push-only stack. The selection of register is determined by the designer's choice of system timing and data movement.

The architecture of a typical 29500 signal-processing system (Fig. 3) can employ separate parallel data paths for complex data. Three possible address-generator blocks are shown, and together they represent a general-purpose processor. Address sequences for other than FFTs might be configured from programmable read-only memory or 2901-based designs. Address pipelining is shown for both data and coefficient addresses.

In this design, either bipolar or MOS static random-access memories store data temporarily, and high-speed bipolar PROMs and RAMs or MOS ROMs are used for coefficient look-up tables. The local-control store may be either a PROM or a writable control-store RAM and can be controlled by a 2910 program sequencer.

A common benchmark for signal processing is the execution speed of an FFT. The 29500 processor, operating at a 10-megahertz clock rate, can perform the transform in 400 nanoseconds. This speed allows a 1,024-point complex radix-2 butterfly to be completed in 2.0 milliseconds. Compared with the best throughput available in current bit-slice CPU architectures, this figure is more than a twentyfold improvement. □

One-chip sequencer shapes up addressing for large FFTs

The addressing circuitry of a single IC accesses both data and coefficient memories for performing a broad class of fast Fourier transforms.

As one of the most useful algorithms in the digital signal-processing repertoire, the fast Fourier transform provides a quick, orderly, and convenient means of computing the frequency spectrum of a signal. When combined with other operations, the FFT is also useful in correlating or convolving two or more waveforms, techniques required to perform radar, sonar, and image processing.

One of the most difficult problems facing the FFT hardware designer is creating the circuitry to address the memories that hold the data variables and coefficient constants. The difficulty arises partially because of the memory space required and the resulting complexity of either accessing a large number of personalized address tables for each FFT or calling out a large data base in the proper sequence. Even when addressing is done in software, there is the problem of speed—the method is often too slow for real-time applications.

A new chip, however, contains all the addressing circuitry needed to access an FFT unit's data and coefficient memories so that a broad class of functions can be analyzed. The Am29540 programmable address sequencer is flexible enough to generate addresses for FFTs having as few as 2 or as many as 65,536 points. Twelve algorithms are supported in radix-2 and radix-4 systems, including operations on complex and real-valued input data (either in-place or non-in-place transforms); forward and inverse transforms; and decimation-in-time (DIT) and decimation-in-frequency (DIF) algorithms.

David Quong and **Robert Perlman**
Advanced Micro Devices Inc.

Robert Perlman is a senior product planning engineer with the DSP/array processing group at Advanced Micro Devices in Sunnyvale, Calif. He obtained a BSEE from the Rensselaer Polytechnic Institute and an MSEE from the Johns Hopkins University, and has previously done design work in airborne digital signal processing at Westinghouse.

David Quong is a product planning engineer with the DSP/array processing group. He received a BSEE from California State University in Sacramento.

A web of nets

Included in the 16-bit sequencer are a butterfly counter (see "Generating Addresses Efficiently," p. 160), a data address generator, and a coefficient address generator (Fig. 1). The butterfly circuit actually has two counters, one for columns and one for rows. The column counter points to the current FFT stage, or column; the row counter, to the butterfly currently being performed within that stage. The counters are programmable and can be initialized to perform transforms of various lengths by prestoring the appropriate 4-bit transform-length code in an on-chip latch. The transform-length code is placed on input lines TL_0–TL_3 and latched with signals

One-chip FFT sequencer

Transform Select (TSEL) and Transform Strobe (TSTRB).

The butterfly counter executes one of four instructions: Reset, Reset/Load, Count, and Hold. These instructions are selected with control lines I_0 and I_1 and are executed on the rising edge of the Clock Input line (CP). An FFT is begun by initializing the butterfly counter with a Reset or Reset/Load instruction. The Count and Hold instructions are then used to advance the counter to the next butterfly operation or to hold it at the present butterfly position.

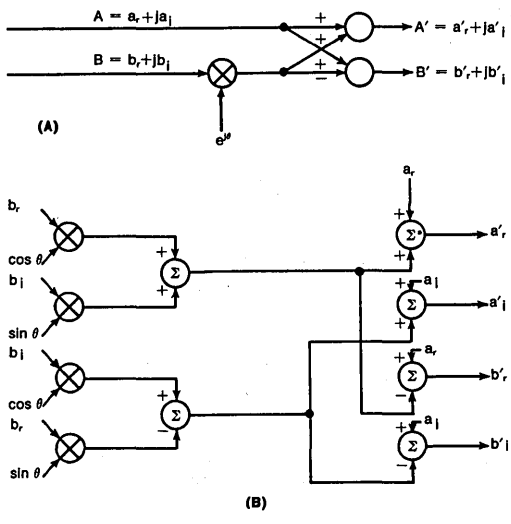
The counter section generates four flags to help control FFT sequencing. The Iteration Complete flag (IT COMP) indicates the last butterfly operation performed in a stage or column; the last butterfly operation in a particular FFT is signaled by the FFT Complete flag (FFT COMP). The Even/Odd flag changes

Generating addresses efficiently

A quick look at the structure of a fast Fourier transform reveals why the data and coefficient circuitry is so complex. At the heart of the FFT algorithm is the butterfly operation, which takes its name from the schematic representation that shows how output data is generated from an input waveform.

In the butterfly operation on a radix-2 DIT FFT (Fig. A), two complex data points, A and B, and one complex coefficient are used to compute two new complex data points, A′ and B′. The coefficient is a complex exponential of the form $e^{-j\theta} = \cos\theta - j\sin\theta$. Each butterfly requires one complex multiplication, one complex addition, and one complex subtraction or four real multiplications, three real additions, and three real subtractions (Fig. B).

An FFT is performed by concatenating butterfly operations. The butterflies are arranged in columns, or stages; an N-point, radix-2 FFT comprises $\log_2 N$ stages, each containing N/2 butterflies, and so a total of (N/2) ($\log_2$N) operations must be done.



The structure of a 16-point FFT contains 32 butterflies (Fig. C). Each circle represents a single radix-2 butterfly operation. The fractional number accompanying each butterfly of the last three stages is the coefficient needed to perform that butterfly: if the value of the fraction is k, the corresponding coefficient value is $e^{-j\pi k}$. Memory locations in which data is stored are represented as blocks; in the case of the 16-point FFT, 16 contiguous memory locations must be allocated to store the 16 complex data points. Each butterfly is performed by taking input points from the data memory, doing the necessary mathematical operations with the appropriate coefficients, and returning the results. The algorithm shown is in-place, meaning that the data points produced by each butterfly are stored in the same locations as the input data points.

The order in which data must be accessed is not straightforward. For the FFT shown, data must be accessed in order 0, 8, 1, 9, ... 7, 15, for the first stage. For the second and following stages, however, data addressing is somewhat more involved. The second stage, for example, is performed by accessing addresses 0, 4, 1, 5, 2, 6, 3, 7, for the first group of four butterflies; then 8, 12, 9, 13, 10, 14, 11, 15, for a second group. The butterflies in the third and fourth stages are also addressed in groups. Stage m has 2^{m-1} groups of $N/2^m$ butterflies with a group spacing of $N/2^{m-1}$.

Coefficients must also be accessed. For the first stage of this FFT, only sin 0 and cos 0 need be acquired. Stage two, however, uses angles 0 and $\pi/2$; stage three uses $0,\pi/2,\pi/4$, and $3\pi/4$; and stage four needs angles $0,\pi/2,\pi/4,3\pi/4,\pi/8,5\pi/8,3\pi/8$, and $7\pi/8$. In general, the coefficient address sequence for the mth stage of this FFT is 0, BR(1)π/N, BR(2)π/N, ... BR $(2^{m-1}-1)\pi$/N, where BR(x) is a function that reverses the order of the bits of a binary word.

A new coefficient must be accessed for each group of butterflies. Other types of FFTs have various addressing sequence requirements, but this example is a good representative. FFT analyzers use several techniques to

state after every stage and can be used to control memory operations for non-in-place transforms. The fourth flag, KNZ/KZ, is of special use when performing transforms with real-valued inputs. The last two flags are multiplexed onto a single pin. When the sequencer produces a data address for a transform with a real-valued-input, KNZ/KZ appears on the pin; for any other type of data, Even/Odd appears.

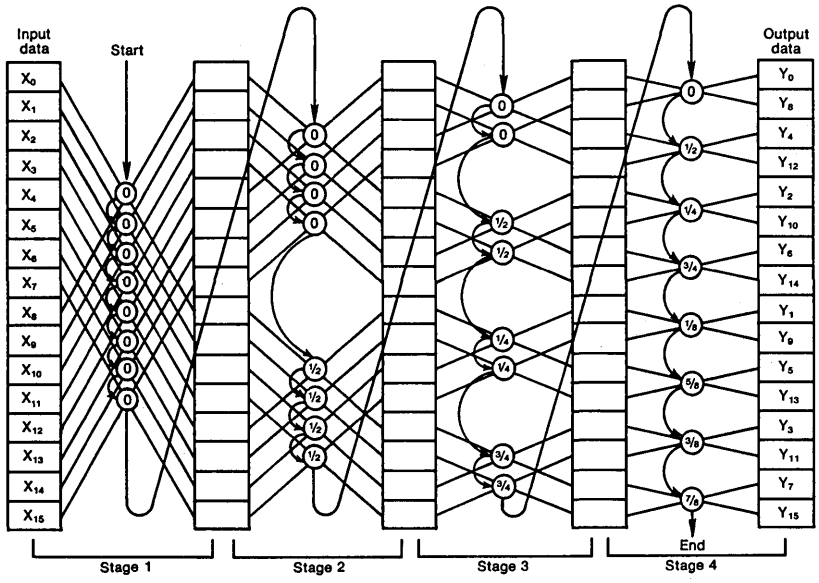
When performing a transform, the sequencer uses its address generators to create addresses combinatorially—that is, the data address generator produces an address for each input and output data point, and the coefficient address generator creates addresses for coefficients and weighting functions. Three control signals—PSD, DIT/DIF, and Radix4/2—configure the address generators for various types of FFTs. These signals are stored in an

generate these data and coefficient addresses.

One of the most common solutions is to place the data and coefficient addresses in PROMs and then to sequentially address the PROMs with a counter. This approach has several serious drawbacks, however. First the number of data addresses becomes prohibitively large as the size of the FFT grows. A 4096-point, radix-2 FFT with complex inputs, for example, must address 24,576 butterflies, each requiring two data addresses and a coefficient address, for a total of 73,728 addresses. Although that number can be reduced by employing constant-geometry FFT algorithms that use the same data addresses for every stage, these algorithms have the disadvantage of being non-in-place, thereby requiring twice the data memory of an in-place transform. The second disadvantage of the PROM approach is that if a single system is to perform several different sizes or types of FFT, a different address table is needed for each FFT.

Another method uses as much SSI and MSI logic as is practical. This approach is easily implemented but usually results in a circuit that consumes considerable board space, is a headache to control, and takes a long time to debug. A circuit for the addressing function in a 4096-point FFT might require 10 to 20 chips.

A third approach is to compute the necessary addresses in software, a method that is often too slow for real-time applications.



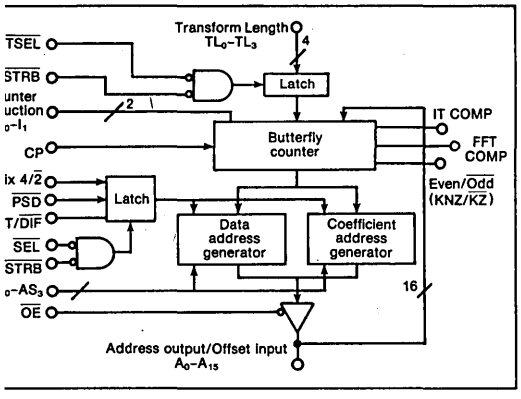
(C)

One-chip FFT sequencer

on-board latch controlled by the Select and Strobe lines (SEL, STRB). The user selects the desired input data, output data, or coefficient address with control lines Address Select 0 through 3 (AS_0-AS_3). The address chosen by lines AS_0-AS_3 is placed on Address lines A_0-A_{15}. Those lines can be forced to a high-impedance state by the Output Enable signal (OE), thus allowing other address generation devices to be tied to the same address bus.

Conserving memory

When addressing data, the sequencer can generate as many as 2^{16} addresses; the actual number needed for a particular FFT depends on the size and type of the transform. An N-point, radix-2, in-place FFT with inputs that are complex quantities, for example, must address N complex-data points during each stage. If N is 16, only 16 memory locations need be addressed, leaving much of the available address space unused.



1. The Am29540 offers a one-chip solution to the problem of addressing data and coefficient memories for performing fast Fourier transforms. The butterfly counter can be programmed to address anywhere from 2 to more than 65,000 points.

The Am29540 offers two data-addressing options for the user who needs less than 64 kwords of space. The first sets the unused upper address bits to zero by initializing the butterfly counter with the Reset instruction. For a 16-point transform, then, the upper 12 address lines would contain 0s for any data address. The four remaining lines are available to call 2^4, or 16, values. The other option is to program the upper address lines to a user-selected value to address a given data block in a large memory. The upper data bits are programmed by bringing OE high, placing the desired bit pattern on address lines A_0-A_{15}, and then executing the butterfly counter's Reset/Load instruction. If, for example, the bit pattern $ABC0_{16}$ is used to initialize a 16-point, complex-input FFT, the sequencer will address a block of sixteen data locations beginning at address $ABC0_{16}$.

Non-in-place transforms present additional problems. Unlike in-place transforms, non-in-place algorithms cannot store the output data from a butterfly operation in the same locations previously occupied by the input data. That problem is overcome by generating both the input and output data addresses for such transforms.

Typically, non-in-place transforms are performed with two data memories, one the source of input data, the other the destination for output data. When a butterfly operation for a given stage is completed, the roles of these memories are reversed, with the output data memory of one stage providing the input data for the next. The Even/Odd signal is particularly useful in such cases; since it changes state after every stage, it can be used to control the direction of data flow between the two memories.

Getting the coefficients

To access coefficients, the Am29540 generates a 16-bit address corresponding to one of 2^{16} equally-spaced angles between 0 and 2π radians. For coefficient address A, the angle addressed is $2\pi A/2^{16}$ the angle $\pi/2$, for instance, would have the address 4000_{16}. The coefficient address is fed to look-up memory, usually PROM, containing sine and cosine values for the angles selected.

A given FFT will use some subset of the more than 65000 angles available. As a case in point, an N-point, radix-2 FFT with complex inputs must access only N/2 equally spaced angles in the range 0 to π radians; a 16-point FFT, then, needs only eight different angles. The sequencer automatically chooses the angles needed in the proper sequence, skipping over unused values.

The coefficient-addressing scheme employed carries a significant benefit for systems in which various sizes of FFTs are to be implemented. Because the chip automatically accesses only those sine and cosine values needed, a single sine/cosine table can be used to perform FFTs of various sizes. If, for example, the user creates a look-up table containing 2048 sine and cosine values between 0 and π radians, that table can be used to perform all radix-2 complex FFTs with 4096 or fewer points.

Most FFT algorithms currently in use are designed to process complex input data. The Am29540 supports 12 different types of this transform (see the table, below). The choices include:

- Radix-2 or radix-4 transforms. The butterfly structure of a radix-4 transform is more complicated than that of radix-2 but offers somewhat greater computational efficiency. Each radix-4 butterfly produces four output data points from four input data points and three coefficients, and consumes 12 real multiplications and 22 real additions. Radix-4 transforms are selected with the Radix4/$\overline{2}$ signal.
- Decimation-in-time or decimation-in-frequency transforms. These terms refer to two basic classes of FFTs; they reflect the manner in which each class is derived. DIT and DIF

Fast Fourier transforms supported by the Am29540

Input data type	Radix	Decimation type	In-place/ non-in-place	Input data ordering	Output data ordering	Direction of transform
Complex	2	DIF	In-place	Normal	Digit-reversed	Forward and inverse
	2	DIF	In-place	Digit-reversed	Normal	Forward and inverse
	2	DIF	Non-in-place	Normal	Normal	Forward and inverse
	2	DIT	In-place	Normal	Digit-reversed	Forward and inverse
	2	DIT	In-place	Digit-reversed	Normal	Forward and inverse
	2	DIT	Non-in-place	Normal	Normal	Forward inverse
	4	DIF	In-place	Normal	Digit-reversed	Forward and inverse
	4	DIF	In-place	Digit-reversed	Normal	Forward and inverse
	4	DIF	Non-in-place	Normal	Normal	Forward and inverse
	4	DIT	In-place	Normal	Digit-reversed	Forward and inverse
	4	DIT	In-place	Digit-reversed	Normal	Forward and inverse
	4	DIT	Non-in-place	Normal	Normal	Forward and inverse
Real-valued (RVI)	2	DIT	In-place	Normal	Unique	Forward
	2	DIF	In-place	Unique	Normal	Inverse

butterfly structures differ somewhat but require an identical number of arithmetic operations. The DIT/$\overline{\text{DIF}}$ signal determines the desired transform type.

•In-place or non-in-place transforms. Non-in-place transforms require twice the data memory of their in-place counterparts. It might seem, then, that in-place transforms would always be preferred. Unfortunately, the in-place approach has a drawback—the digits of the address of the input or output data must be called for in reversed order. This scheme requires a reordering operation. The choice between in-place and non-in-place algorithms is made by using the appropriate values of A_0–AS_3 to select the desired addresses. Should the user select an in-place transform, the choice of digit-reversed-address input or output can be made with the signal $\overline{\text{PSD}}$.

Useful inversions

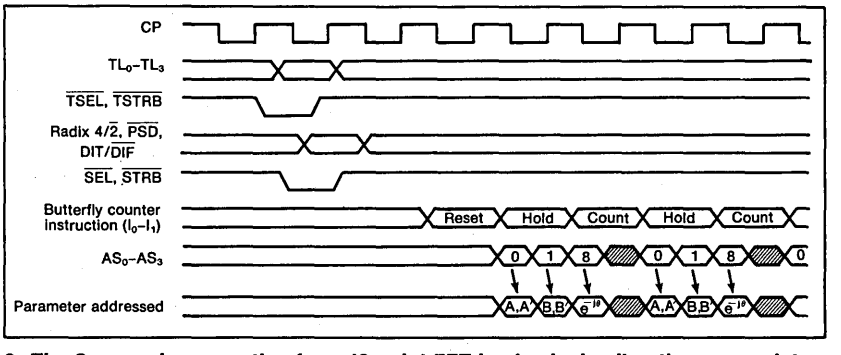
The sequencing chip also can be used to perform inverse transforms, a useful feature in applications requiring a route from the frequency to the time domain. Computing inverse transforms is straightforward—the address sequences needed are the same as those for the forward operations. With radix-2 transforms, the only difference between the inverse and forward transforms is the complex exponential: $e^{-j\theta}$ must be replaced with $e^{j\phi}$. Changing the sign of the complex exponential's argument is equivalent to replacing the coefficient $\sin\theta$ with $-\sin\theta$, an operation that can be executed by slightly modifying the addition and subtraction operations performed in the butterfly. Radix-4 inverse transforms require somewhat similar minor accommodations to sign changes in the butterfly calculation.

Some applications demand FFT transforms with real-valued inputs. The sequencer generates data addresses for both forward and inverse real-valued-input (RVI) transforms of a type first described by Bergland.[1]

A weighty matter

FFT filter characteristics can often be significantly improved by premultiplying the input data with a series of weighting factors. This technique, also called windowing, or shading, can significantly lower filter sidelobes and thus simplify the analysis. The properties of a number of common weighting functions are well-documented.[2]

The sequencer supports two weighting approaches for radix-2 transforms. The first and



2. The Sequencing operation for a 16-point FFT begins by loading the appropriate transform-length code and control signals into on-board latches. The butterfly counter is then reset. After initialization, the first butterfly's memory addresses are selected with lines AS_0–AS_3. The sequencer is then advanced to each succeeding butterfly using the count instruction.

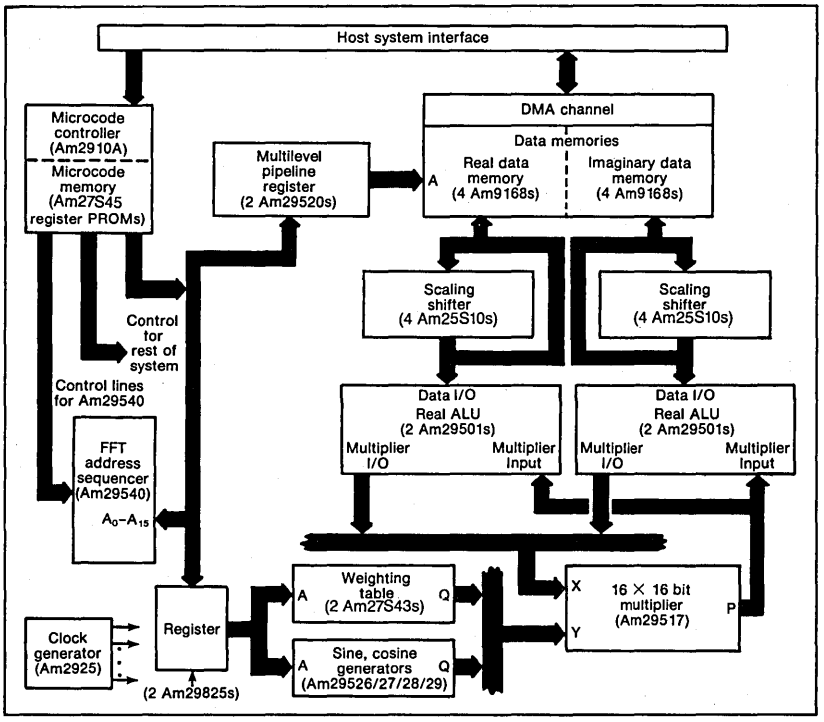
simplest approach is to perform a weighting prepass before the FFT begins.

The sequencer is programmed to perform the first stage of a radix-2 DIF transform. The resulting prepass data addresses access the input data, the coefficient addresses access weighting values stored in a look-up table. On completion of the prepass, the part is reprogrammed to address the type of radix-2 FFT desired.

The second approach takes advantage of the structure of a DIT FFT. For the first stage of the transform, only the coefficient values sin 0 and cos 0 are needed. Weighting can thus be incorporated in this first stage, using the stage's multiplier. By configuring the part to perform a radix-2, DIF FFT for stage 1, and then changing the FFT type from DIF to DIT for all remaining stages, the necessary data, weighting, and coefficient addresses can be generated.

A look at an FFT

Virtually all useful weighting functions are symmetrical. If Y(n) is a symmetrical N-point weighting function, point Y(x) is equal to Y(N−x). This symmetry implies that the user need not store all N points of the weighting function: (N/2)+1 points are sufficient. The sequencer addresses such half tables by generating both x and N−x. The often-used von Hann weighting function is one such example,



3. In a typical system, the Am29540 is used to access data, weighting, and sine and cosine (coefficient) values in a microcode-controlled FFT processor. This system can support a 4-kpoint radix-2 transform.

easily derived from the table of cosines required by the FFT algorithm itself. Thus, the need for a separate weighting-function memory is altogether eliminated.

The sequencer's operation can be best understood by considering its performance of a typical FFT. Suppose, then, that an in-place, radix-2, 16-point DIT FFT is to be implemented (see "Generating Addresses Efficiently," Fig. C, p. 160). To initialize the device, the appropriate transform-length code and control bits are loaded into the on-chip latches. For this example, the transform-length code has the value 0011_2; the control bits must assume the values $\overline{PSD} = 1$, Radix4/$\overline{2} = 0$, and DIT/$\overline{DIF} = 1$. After this data has been entered, the butterfly counter is initialized with a Reset or Reset/Load instruction.

Once initialized, the part generates data and coefficient addresses for the FFT's first butterfly. For this algorithm, the input and output data addresses are set at 0 and 1, respectively, with lines AS_0–AS_3; the coefficient address is similarly set to 8. After all the addresses have been read, the device is advanced to the next butterfly by executing a count instruction (Fig. 2).

Defining the system

The Am29540's working environment is a microcode-driven FFT processor. That system can be divided into several basic blocks (Fig. 3): the address sequencer, arithmetic processor, high-speed data memory and coefficient memory, system controller, and the host interface.

The address computer generates the read and write addresses to access data, as well as coefficient and weighting addresses.

The arithmetic processor, consisting of a multiplier (here, the Am29517) and two ALUs (one for real and the other for imaginary data), efficiently calculates complex data from the data memory. Using coefficient and weighting generators, it processes the information and returns it to the data memory. The data width is 16 bits; therefore each ALU requires two 8-bit-slice multiport pipelined processors (here, Am29501s). A scaling shifter is provided in each data path from the memory to the ALUs.

The high-speed data memory stores input and output data from an FFT operation. It is also divided into two banks, one for real, one for imaginary data. The sequencer can be loaded with a data address offset, allowing data memory to be addressed at starting locations other than zero, and permits the addressing of selected blocks of data. A set of coefficient generators (Am29526/27/28/29) provide the coefficients needed when performing an FFT and produce up to 2048 words of sine and cosine data. This is sufficient to support up to a 4096-point, radix-2 transform. A PROM contains the weighting values for the FFT input data.

The system controller, as overseer, accepts instructions through the host computer interface, determines which function must be performed, issues the proper instructions to other components, and informs the host when the operation is done. It employs a microsequencer (the Am2910A) and microcode memory.

The host interface consists of logic to handle the host system protocols and a DMA controller for high-speed data transfer. During block data transfer the DMA circuitry has direct access to the data memory.

The address sequencer generates both read and write addresses for the data memory. When, as is usual, the operations for a sequence of butterflies are overlapped, those addresses must be temporarily stored in an agile shift-register pipeline. This structure must unravel the intertwined sequence of addresses for the several butterflies that are in progress at any given time. Here, a multilevel pipeline register consisting of two Am29520s is used. It can serve as dual two-level or a single four-level pipeline register, and each of the registers is available to the output at any time. □

References

1. G. D. Bergland, "A Fast Fourier Transform Algorithm for Real-Valued Series," *Communications of the ACM*, Vol. 11, Number 10, October 1968, pp. 703-710.

2. F. J. Harris, "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform," *Proceedings of the IEEE*, Vol. 66, Number 1, January 1978, pp. 51-83.

500-kHz single-board FFT system incorporates DSP-optimized chips

VLSI devices optimized for digital signal processing can realize architectures that, compared with traditional designs, save space, power and money. Such chips serve as the basis for a single-board system that uses fewer than 40 standard components.

Robert Cohen *and* **Robert Perlman,**
Advanced Micro Devices Inc

By employing VLSI devices to implement the fast Fourier transform, you can build a single-board digital-signal-processing system that supports sampling rates to 500 kHz and requires fewer than 40 packages (including processor, sequencer and local memory).

In such systems, the FFT makes possible many applications that would otherwise be unrealizable because of computational complexity. FFT techniques require a great number of calculations, and general-purpose computers incorporating the FFT aren't fast enough for such real-time high-bandwidth signal-processing systems as radar, video processing and telecommunications. Until the introduction of VLSI devices that are optimized for DSP tasks, only expensive array processors and special-purpose systems constructed with hundreds of SSI and MSI components could serve such applications.

Optimize butterfly execution

Effectively applying these VLSI circuits requires a familiarity with the FFT's computational requirements (see **box,** "FFTs reduce DFT computations"). Then, you can implement an appropriate algorithm in hardware. Because the FFT's basic operation is the butterfly, you can start by designing a butterfly processor.

Fig 1 lists the steps required to process a butterfly. The list helps you to determine the minimal resources required: an ALU, a multiplier and enough memory to hold the real and imaginary components of N samples. By adding resources, you can increase parallelism and boost throughput. For example, separate memories for the real and imaginary components of the sample data allow you to read A or B (or write A′ or B′) in one cycle. Extending this concept, you can divide the data pathway into a real-variable processor and an imaginary-variable processor **(Fig 2).**

The multiplier (or set of multipliers) acts as a shared

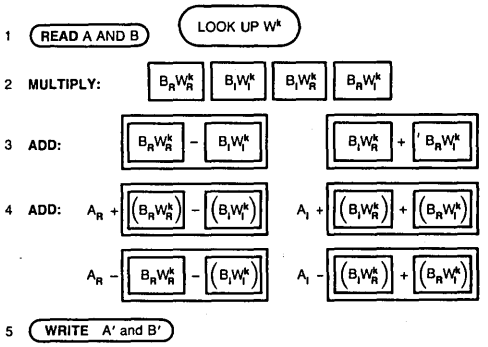


Fig 1—Five sequential steps *implement the butterfly, which is the primitive DFT operation. This list shows that the absolute minimum resources required to implement a butterfly are an ALU, a multiplier and memory.*

Reprinted with permission from EDN, October 31,1984

A butterfly processor needs an ALU, a multiplier and memory

resource for both processors because it operates on both types of data. Each processor consists of an ALU and registers that hold intermediate results. Although you could add more ALUs, they prove superfluous for the FFT algorithm used here. (Additional ALUs are useful in radix-4 algorithms; see **reference.**)

To achieve the best performance, you minimize the number of cycles needed to execute the butterfly. Parallel computations allow the processor to accomplish more in each cycle to effect the desired reduction.

With an architecture like the one suggested—two memories, two processors and several multipliers—what is the smallest number of required cycles? To find out, examine **Fig 1** and start by using two cycles to read A and B from memory. (You can store W^k in a PROM and read it concurrently with A and B.) Assuming the processor has four multipliers, step 2 executes in one cycle. Step 3 also executes in one cycle if it determines the left-column difference in the real ALU and the right-column sum in the imaginary ALU. Step 4 requires two cycles, the left two operations being performed in the real-variable ALU and the right two in the imaginary-variable ALU. A′ and B′ are then written to memory in two cycles. This process executes a complete butterfly in eight cycles.

You now estimate how fast this processor can operate. Assuming that N=1024, the processor must perform $(N(\log_2)N)/2$ butterflies (a total of 5120). At eight cycles per butterfly, the processor needs 40,960 cycles. Next, assume a 100-nsec cycle time. (Cycle time depends on the slowest pathway through the system, which is typically via the multiplier; 16×16-bit combinatorial multipliers with sub-100-nsec propagation delays are common.) Under these conditions, a 1k-sample transform requires 4 msec, corresponding to a 0.25-MHz sampling rate, which is quite respectable for many applications. Further scrutiny will reveal ways to reduce hardware and increase throughput.

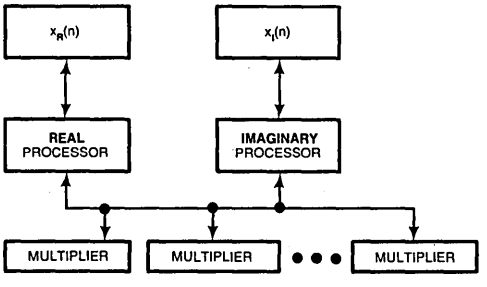


Fig 2—Separate real and imaginary data pathways *allow you to share multipliers and reduce required system resources.*

Less hardware does the job faster

Fig 3 shows a resource-utilization table for an 8-step butterfly. Note that all resources are idle most of the time: The data bus is active only 50% of the time, the ALUs 38%, and the multipliers 13%. You can take advantage of this idle time by executing butterflies concurrently, a technique known as pipelining.

For example, after reading A and B for the first butterfly, the data bus can read A and B for the second butterfly during cycles 3 and 4 while the multipliers and ALUs are busy. The multiplier could then begin working on the second butterfly immediately after computing results for the first. Using this technique, the processor still requires eight cycles to complete a

	DATA BUS	REAL ALU	IMAGINARY ALU	MULTIPLIERS
1	A			
2	B			
3				$B_RW_R^k$ $B_IW_I^k$ $B_RW_I^k$ $B_IW_R^k$
4		$B_RW_R^k - B_IW_I^k$	$B_IW_R^k + B_RW_I^k$	
5		$A_R + B_RW_R^k - B_IW_I^k$	$A_I + B_IW_R^k + B_RW_I^k$	
6		$A_I - B_IW_R^k - B_RW_I^k$	$A_R - B_RW_R^k + B_IW_I^k$	
7	A′			
8	B′			

Fig 3—An 8-step butterfly, *which implements the algorithm prior to optimization, uses its constraining resource—the data bus—only 50% of the time.*

	DATA BUS	REAL ALU	IMAGINARY ALU	MULTIPLIERS
1	A			
2	B			
3				$B_RW_R^k$ $B_IW_I^k$ $B_RW_I^k$ $B_IW_R^k$
4		$B_RW_R^k - B_IW_I^k$	$B_IW_R^k + B_RW_I^k$	
5	A	$A_R + B_RW_R^k - B_IW_I^k$	$A_I + B_IW_R^k + B_RW_I^k$	
6	B	$A_I - B_IW_R^k - B_RW_I^k$	$A_R - B_IW_R^k + B_RW_I^k$	
7	A′			$B_RW_I^k$ $B_IW_I^k$ $B_RW_I^k$ $B_IW_R^k$
8	B′			

LOOP ON THESE STEPS

Fig 4—By starting a second butterfly concurrently, *you can create a 5-cycle loop and improve throughput 38% compared with* **Fig 3***'s operation.*

particular butterfly, but it reduces the average number of cycles per butterfly because it works on more than one butterfly at a time.

The most heavily used resource determines the minimum average number of cycles per butterfly that you can achieve. By using the four idle bus cycles, you can reduce the average number of cycles per butterfly to four and double system throughput.

You can see this doubling of throughput clearly in **Fig 4**'s resource-utilization table. A second concurrent butterfly starts on cycle 5; A and B are then read, and the new products are calculated in cycle 7. (You can also start the second butterfly on cycle 3 or 4.) After completing cycle 8, the processor jumps to cycle 4 instead of cycle 1, because it has already read A and B and computed the new products for the second butterfly. This technique creates a 5-cycle loop instead of an 8-cycle loop, improving throughput by 38%.

	DATA BUS	REAL ALU	IMAGINARY ALU	MULTIPLIERS
1	A			
2	B			
3				$B_RW_R^k$ $B_IW_I^k$ $B_RW_I^k$ $B_IW_R^k$
4		$B_RW_R^k - B_IW_I^k$	$B_IW_R^k + B_RW_I^k$	
5	A	$A_R + B_RW_R^k - B_IW_I^k$	$A_I + B_IW_R^k + B_RW_I^k$	
6	B	$A_I - B_IW_R^k - B_RW_I^k$	$A_R - B_RW_R^k + B_IW_I^k$	
7	A'			$B_RW_R^k$ $B_IW_I^k$ $B_RW_I^k$ $B_IW_R^k$
8	B'	$B_RW_R^k - B_IW_I^k$	$B_IW_R^k + B_RW_I^k$	

LOOP ON THESE STEPS

Fig 5—Data-bus utilization is 100% *in a 4-cycle butterfly. Here you can see that using just one multiplier doesn't hinder throughput.*

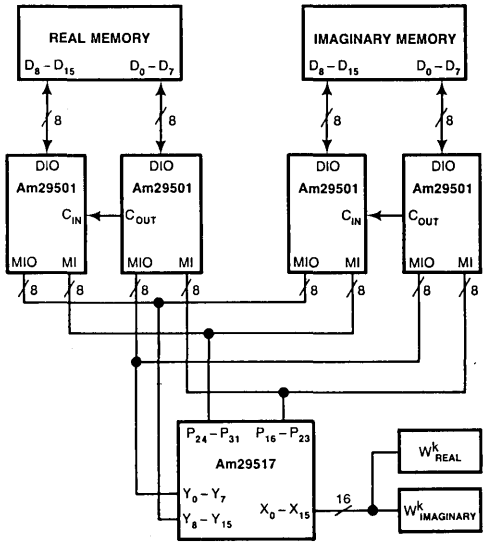


Fig 6—This FFT circuit *uses only one multiplier and a handful of other components.*

Fig 5's table shows how to achieve even higher performance. You can copy cycle 4 into cycle 8, which allows the processor to jump to cycle 5 and produce a 4-cycle loop. This action doubles the original throughput. In this case, the data bus experiences 100% utilization and the ALUs 75%, but the multipliers are still employed only 25% of the time. Clearly, you don't need four multipliers. In fact, you can achieve the same performance with only one multiplier by pipelining an additional butterfly. A design example demonstrates this technique.

Start a butterfly every four cycles

The **Fig 6** design uses a real-variable processor and an imaginary-variable processor, each with two Am29501s to provide 16-bit precision. (The Am29501 is an 8-bit, cascadable processor comprising an ALU, a set of six registers, and three data ports.) The two processors also share an Am29517 16-bit parallel multiplier, which has two 16-bit inputs, X and Y. The Y input connects to the multiplier I/O (MIO) port on the real and imaginary 29501s; the X input is driven either by a PROM containing the complex constants W^k or by Am29526/27/28/29 sine/cosine generators. The high-order 16 bits of the multiplier output ($P_{16\text{-}31}$) go to the 29501s' multiplier input (MI) ports, while the low-order 16 bits of the product are ignored. Memory consists of static RAM with a cycle time of less than 100 nsec.

The microcode needed to perform one butterfly is 10 cycles long (**Fig 7a**), but you should note two things. First, registers are never used for more than four cycles, so the processor can load them with new values every four cycles. This, in turn, means that it can start a new butterfly every four cycles.

Second, you can superimpose each line of code onto the line four cycles below it without causing resource conflicts. For example, **Fig 7b**'s code superimposes lines 1 through 4 over lines 5 through 8 to start computing a second butterfly while the first is still executing. This process repeats in **Fig 7c**'s code, where lines 5 through 8 are then superimposed over lines 9 through 12. These last four lines contain the code necessary to compute three concurrent butterflies. You must ensure only that, when the processor reads or writes A or B, it knows exactly to which butterfly the data applies.

FFTs reduce DFT computations

Fourier transforms mathematically approximate a signal's transformation from the time domain to the frequency domain, and several algorithms implement the technique. All are based on the discrete Fourier transform (DFT), which sums time-domain samples (x(n)) that are multiplied by complex constants:

$$X(k) = \sum_{n=0}^{N-1} x(n)W^{2uk/N}$$
$$k = 0, 1, \ldots, N-1,$$

where $W=e^{-j\pi}$ and each X(k) is a frequency-domain Fourier coefficient. The computation of each coefficient requires N complex multiplications, where N is the number of samples. This results in N^2 complex multiplications.

The fast Fourier transform (FFT) reduces complex multiplications by eliminating redundant calculations, using the equation

$$X(k) = G(k) + W^{2k/N} H(k)$$
$$k = 0, 1, \ldots, N-1, \quad \textbf{(1)}$$

where G(k) is the DFT of the even samples in x(n), and H(k) is the DFT of the odd samples. (The algorithm discussed here is a radix-2 decimation-in-time algorithm; other schemes may provide additional benefits.)

Shaving points

Based on **Eq 1, Fig A** shows an 8-point DFT that's divided into two 4-point DFTs, one of which operates on even samples while the other operates on odd ones. As **Fig A** shows, the results are summed to produce the 8-point DFT result. This configuration takes advantage of the fact that G(k) and H(k) have period N/2. In other words,

$$G\left(k + \frac{N}{2}\right) = G(k)$$

$$H\left(k + \frac{N}{2}\right) = H(k).$$

Each 4-point DFT requires $N^2=16$ complex multiplications, and combining the intermediate results to obtain the eight frequency-domain coefficients requires one complex multiplication for each coefficient (the arrows represent multiplication by the noted constant). Thus, the **Fig A** transform requires a total of 40 (16+16+8) complex multiplications—a savings of 24 compared with the 64 multiplications required to compute an 8-point DFT directly.

By repeating this process and dividing the 4-point DFTs into 2-point DFTs, you can eliminate even more computations. The 8-point FFT represented in **Fig B** requires eight complex multiplications for the four 2-point DFTs plus 16 other complex multiplications, for a total of 24. In general, the number of complex multiplications equals the number of columns in the representation ($\log_2 N$) times the number of samples.

Another technique allows you to cut multiplications in half again. In **Fig C,** each circle represents a

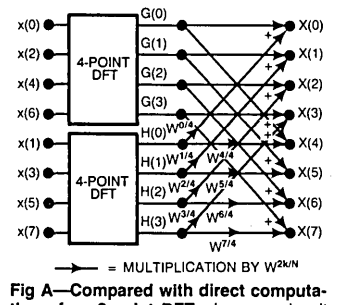


Fig A—Compared with direct computation of an 8-point DFT, *decomposing it into two 4-point DFTs saves 24 complex multiplications.*

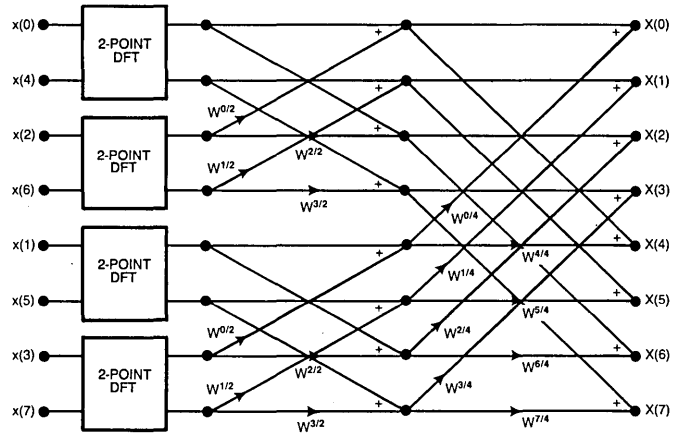


Fig B—This scheme, which uses four 2-point DFTs *to transform eight time-domain samples, requires a total of 24 complex multiplications: eight for the four 2-point DFTs plus the 16 represented here by arrows.*

sum and a difference. Using the structure in **Fig D**, you define A′ and B′ as follows:

$$A' = A + BW^{2k/N}$$
$$B' = A - BW^{2k/N}.$$

This notation results because W is a complex exponential and therefore periodic:

$$W^{2k/N} = -W^{2(k+\frac{N}{2})/N}.$$

Because you can use the product $B \times W^{2k/N}$ to calculate both A′ and B′, the total number of complex multiplications drops to $(N\log_2 N)/2$. This structure is the primitive operation in FFT calculations and is called a butterfly operation. Note that each circle in **Fig C** is a butterfly operation. This fact suggests a pipelined operation, optimized to execute butterflies, that can exploit the algorithm's highly repetitive nature.

Dissecting the butterfly

Each butterfly consists of two calculations:

$$A' = A + BW^k$$
$$B' = A - BW^k,$$

where A′, B′, A, B and W^k are complex numbers. (Here, the exponent 2k/N is consolidated into one term, k, for simplicity.) Dividing these into their real and imaginary parts yields

$$A' = (A_R + j\,A_I) + (B_R + j\,B_I)(W_R^k + j\,W_I^k)$$
$$B' = (A_R + j\,A_I) - (B_R + j\,B_I)(W_R^k + j\,W_I^k).$$

Expanding the products gives

$$A' = (A_R + j\,A_I) + (B_R\,W_R^k + j\,B_I\,W_R^k + j\,B_R\,W_I^k - B_I\,W_I^k)$$
$$B' = (A_R + j\,A_I) - (B_R\,W_R^k + j\,B_I\,W_R^k + j\,B_R\,W_I^k - B_I\,W_I^k).$$

Dividing A′ and B′ into their real and imaginary parts yields four equations:

$$A_R' = A_R + B_R\,W_R^k - B_I\,W_I^k$$
$$B_R' = A_R - B_R\,W_R^k + B_I\,W_I^k$$
$$A_I' = A_I + B_I\,W_R^k + B_R\,W_I^k$$
$$B_I' = A_I - B_I\,W_R^k - B_R\,W_I^k.$$

These equations share common terms that need be calculated only once per butterfly. Regrouping terms yields additional savings:

$$A_R' = A_R + (B_R\,W_R^k - B_I\,W_I^k)$$
$$B_R' = A_R - (B_R\,W_R^k - B_I\,W_I^k)$$
$$A_I' = A_I + (B_I\,W_R^k + B_R\,W_I^k)$$
$$B_I' = A_I - (B_I\,W_R^k + B_R\,W_I^k).$$

You can now determine the number of calculations necessary per butterfly: four multiplications to compute $B \times W^K$, a subtraction to calculate real A′ and B′, an addition to calculate imaginary A′ and B′, two final additions for A′, and two final subtractions for B′. This process yields a total of four products, three additions and three subtractions per butterfly.

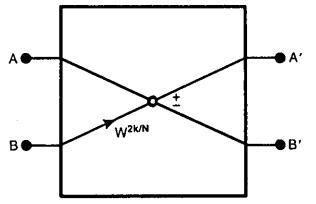


Fig D—The 2-point DFT, or butterfly, *is the primitive operation in FFT calculations.* ***Fig C*** *includes many such operations and thus lends itself to a pipelined implementation.*

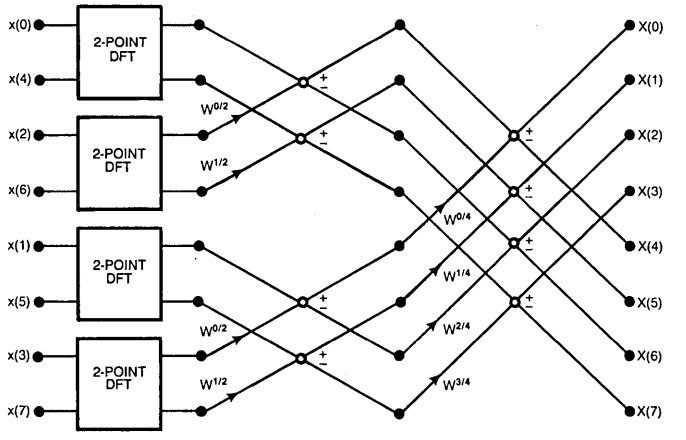


Fig C—In this DFT representation, *intermediate results are multiplied by the complex constant* $W^{2k/N}$ *and then summed. Note that each sum of products, illustrated by a circle, is actually a butterfly operation.*

(a)

A′ = A + BK
B′ = A − BK

STEP	DIO	REAL							IMAGINARY							MULTIPLIER	
		ALU	A_1	A_2	A_3	B_1	B_2	B_3	ALU	A_1	A_2	A_3	B_1	B_2	B_3	MIO	MULT
1	READ B					DI							DI				
2																B_R	
3	READ A		DI							DI							$B_R W_R$
4		A_1 − MI			MI		ALU									B_I	$B_R W_I$
5									A_1 − MI			MI		ALU			$B_I W_R$
6		$A_1 + A_3$		ALU					B_2 − MI					ALU	MI		$B_I W_I$
7		B_2 + MI				MI	ALU		$A_1 + B_3$		ALU						
8	WRITE B_2								$A_2 + A_3$		ALU						
9		$A_2 − B_1$		ALU													
10	WRITE A_2																
11																	
12																	
13																	
14																	
15																	
16																	
17																	
18																	

(b)

A′ = A + BK
B′ = A − BK

STEP	DIO	REAL							IMAGINARY							MULTIPLIER	
		ALU	A_1	A_2	A_3	B_1	B_2	B_3	ALU	A_1	A_2	A_3	B_1	B_2	B_3	MIO	MULT
1	READ B					DI							DI				
2																B_R	
3	READ A		DI							DI							$B_R W_R$
4		A_1 − MI			MI		ALU									B_I	$B_R W_I$
5	READ B					DI			A_1 − MI			MI	DI	ALU			$B_I W_R$
6		$A_1 + A_3$		ALU					B_2 − MI					ALU	MI	B_R	$B_I W_I$
7	READ A	B_2 + MI	DI			MI	ALU		$A_1 + B_3$	DI	ALU						$B_R W_R$
8	WRITE B_2	A_1 − MI			MI		ALU		$A_2 + A_3$		ALU					B_I	$B_R W_I$
9		$A_2 − B_1$		ALU													
10	WRITE A_2																
11																	
12																	
13																	
14																	
15																	
16																	
17																	
18																	

(c)

A′ = A + BK
B′ = A − BK

STEP	DIO	REAL							IMAGINARY							MULTIPLIER	
		ALU	A_1	A_2	A_3	B_1	B_2	B_3	ALU	A_1	A_2	A_3	B_1	B_2	B_3	MIO	MULT
1	READ B					DI							DI				
2																B_R	
3	READ A		DI							DI							$B_R W_R$
4		A_1 − MI			MI		ALU									B_I	$B_R W_I$
5	READ B					DI			A_1 − MI			MI	DI	ALU			$B_I W_R$
6		$A_1 + A_3$		ALU					B_2 − MI					ALU	MI	B_R	$B_I W_I$
7	READ A	B_2 + MI	DI			MI	ALU		$A_1 + B_3$	DI	ALU						$B_R W_R$
8	WRITE B_2	A_1 − MI			MI		ALU		$A_2 + A_3$		ALU					B_I	$B_R W_I$
9	READ B	$A_2 − B_1$		ALU		DI			A_1 − MI			MI	DI	ALU			$B_I W_R$
10	WRITE A_2	$A_1 + A_3$		ALU					B_2 − MI					ALU		B_R	$B_I W_I$
11	READ A	B_2 + MI	DI			MI	ALU		$A_1 + B_3$	DI	ALU						$B_R W_R$
12	WRITE B_2	A_1 − MI			MI		ALU		$A_2 + A_3$		ALU					B_I	$B_R W_I$
13																	
14																	
15																	
16																	
17																	
18																	

Fig 7—The microcode needed to execute one butterfly *in Fig 6's circuit is 10 cycles long* ***(a)****. By following the code in* ***(b)*** *you can start computing a second butterfly concurrently. The code in* ***(c)*** *starts a third concurrent butterfly.*

Pipeline the FFT processor to reduce bus's 50% idle time

To help keep these values straight, the circuit uses the Am29520 multilevel pipeline register. **Fig 8** depicts how this device operates in the address pathway to memory. Every four clock cycles, at the beginning of a new butterfly, the Am29540 FFT address sequencer generates a new set of addresses for A and B. The 29520s store these addresses temporarily in internal registers that are configured as a 4-deep pipeline. As each new address is clocked into the first pipeline register, previously stored addresses advance to the next register. You can select any register for output and access the appropriate address for any microcode cycle. **Fig 9** illustrates the order in which this design stores and retrieves data addresses.

The 29540 also generates addresses for the PROM containing W^k. It creates a new address for each butterfly and then stores it in an external register. Because complex products are computed on successive cycles, the address to the PROM changes at the beginning of each new butterfly (that is, every four cycles).

Microcode lines 9 through 12 then execute as a loop until the 29540's FFT Complete signal goes active. The entire transform requires only 12 words of microcode: The first eight preload the pipeline, while the last four perform the computations.

Word size can almost triple

In these FFT implementations, you must be concerned about word growth. Because the FFT butterfly produces outputs by adding terms, butterfly outputs may require more bits than each input has. Specifically, consider the equation $A'=A+BW^k$, where A', A, B and W^k are complex. This equation, one of the butterfly's two basic calculations, represents vector rotation and addition. The term BW^k merely describes a rotation of vector B by unit vector W^k; the result is added to vector A. The magnitude of A' can therefore be twice as large as A or B.

Unfortunately, this problem is more insidious than it appears. Although complex magnitudes do no more than double at each stage, the real and imaginary components of these complex values can increase by more than that amount. Indeed, they can increase by $1+\sqrt{2}$, or 2.41, for decimation-in-time algorithms, which is the type used here. They can even increase by $2\times\sqrt{2}$, or 2.82, for decimation-in-frequency algorithms, which use a different butterfly technique (see **reference**). In either case, you must allow for as much as two bits of growth in every stage. You could design a system with sufficient extra growth bits, but this approach is wasteful and expensive, particularly if the transform has many stages.

An inexpensive alternative is to use the block-floating-point scheme. This technique uses a common block

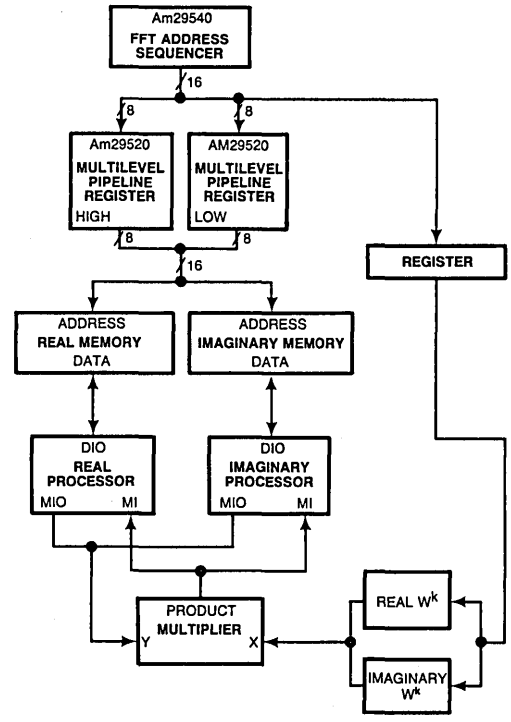


Fig 8—The system must keep track of many sets of data *when computing concurrent butterflies. To do so, it must incorporate an FFT address sequencer.*

CYCLE	MEMORY OPERATION	29520 INPUT	29520 CONTENTS — REG A_1	REG A_2	REG B_1	REG B_2	29520 OUTPUT
0		$*B_0$					
1	READ B_0		$*B_0$				$*B_0$
2		$*A_0$	$*B_0$				
3	READ A_0		$*A_0$	$*B_0$			$*A_0$
4		$*B_1$	$*A_0$	$*B_0$			
5	READ B_1		$*B_1$	$*A_0$	$*B_0$		$*B_1$
6		$*A_1$	$*B_1$	$*A_0$	$*B_0$		
7	READ A_1		$*A_1$	$*B_1$	$*A_0$	$*B_0$	$*A_1$
8	WRITE B_0	$*B_2$	$*A_1$	$*B_1$	$*A_0$	$*B_0$	$*B_0$
9	READ B_2		$*B_2$	$*A_1$	$*B_1$	$*A_0$	$*B_2$
10	WRITE A_0	$*A_2$	$*B_2$	$*A_1$	$*B_1$	$*A_0$	$*A_0$
11	READ A_2		$*A_2$	$*B_2$	$*A_1$	$*B_1$	$*A_2$
12	WRITE B_1	$*B_3$	$*A_2$	$*B_2$	$*A_1$	$*B_1$	$*B_1$

*indicates "address of"

Fig 9—Four registers in the address sequencer *prove sufficient to store the various data addresses needed to compute three concurrent butterflies.*

Efficient microcode executes three butterflies concurrently

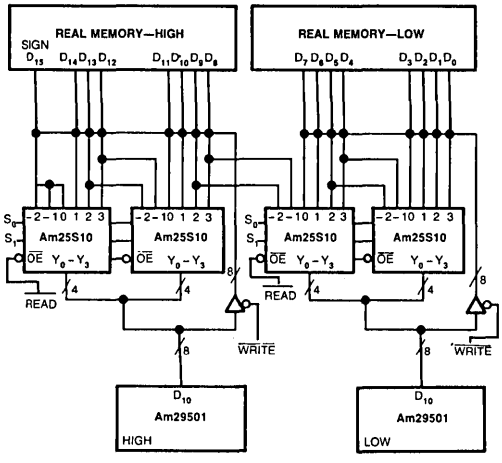


Fig 10—To avoid data overflow caused by word growth, *implement a block-floating-point technique with 4-bit shifters inserted into the data-read pathway.*

exponent for all data. If the system expects or detects an overflow, it shifts data to the right and increments the block exponent.

The circuit shown in **Fig 10** implements this approach with two Am25S10 4-bit shifters inserted between memory and the real and imaginary processors in the data-read pathway. The shifters allow you to divide data read from memory by 1, 2 or 4. Each time the system writes data to memory, external logic compares the two high-order data bits to the sign bit.

If the high-order bits differ from the sign bit, the data's magnitude has expanded into the high-order bits, and an overflow could occur in the next column of butterflies because data could increase by 2.41. Consequently, if logic detects an expansion into the high-order bits, it sets a flag. Then, when the next column begins (signaled by Iteration Complete from the 29540), the system reads all data as shifted to the right by zero bits (if no expansion took place), by one bit (if the expansion occurred only in D_{13}) or by two bits (if the expansion occurred in D_{14}). Note that the sign bit must be duplicated in the high-order bits. Upon receipt of Iteration Complete, the block-exponent counter increments by 0, 1 or 2. The host CPU can then read this value to determine the Fourier coefficients' absolute magnitude.

Though the 29540 FFT address sequencer has many operating modes to accommodate varying architectures and algorithms, the system described here executes a radix-2 decimation-in-time transform that doesn't produce data in a convenient sequence. You'd like data for the first frequency notch to occupy the lowest memory location, data for the second notch to occupy the next lowest location, and so on. To remedy this situation, you can either scramble data points before they enter the algorithm so that they emerge in the proper sequence, or you can scramble them afterwards.

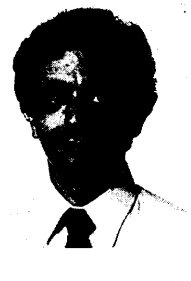
Although this article's architecture describes a special-purpose FFT processor, you can use it as a general-purpose signal processor. Many signal-processing algorithms have a sum-of-products notation that is well suited to this design. Essentially, you can substitute the PROM that contains W^k with a RAM that the host processor loads. In this way, you can easily implement windowing and scaling operations. **EDN**

Reference

Oppenheim, Alan V and Schafer, Ronald W, *Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1975.

Authors' biographies

Robert Cohen worked as a design engineer in product planning at Advanced Micro Devices (Sunnyvale, CA) from 1981 to 1984. He is now a private consultant. He received a BSE degree in computer science and engineering from the University of Pennsylvania in 1981. His favorite food is a combination of guacamole and knishes.



Robert Perlman is a senior product planning engineer with the DSP/array processing group at Advanced Micro Devices. He holds a BSEE degree from the Rensselaer Polytechnic Institute and an MSEE degree from Johns Hopkins University, and he has done design work in airborne digital design processing for Westinghouse.

Trim DSP overhead by changing your sampling rate

The sampling rate of a signal may be altered very easily as it passes through the various stages of a digital signal processing system. This can reduce the number of cycles required to perform operations in the digital domain.

Kenn Lamb
Advanced Micro Devices
Sunnyvale, California

Emerging digital signal processing techniques require arithmetically intensive real-time processing. Each sample fed into your system must be operated on many times. Therefore, the performance required of the DSP processor is determined by both the type of processing to be applied to the signal and by its sample rate.

The sampling rate of a DSP system is usually determined by the analog-to-digital converter at the front end of the system. The choice of this sampling rate affects how well available arithmetic resources are used. Here's how to create a highly efficient system.

The techniques used are called "decimation" and "interpolation." Both are used to decrease and increase the sampling rate, respectively. When implemented using digital signal processing components (such as slices, programmable sequencers, and multipliers), these methods allow the construction of very efficient narrowband filters which can outperform direct implementations of the desired filter.

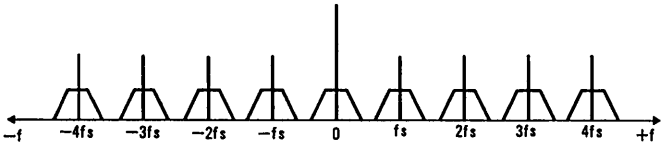


Fig. 1. A sampled signal's spectrum is repeated at periodic intervals, centered at integer multiples of the sampling frequency. The bandwidth of the images must be less than the sampling frequency to avoid aliasing.

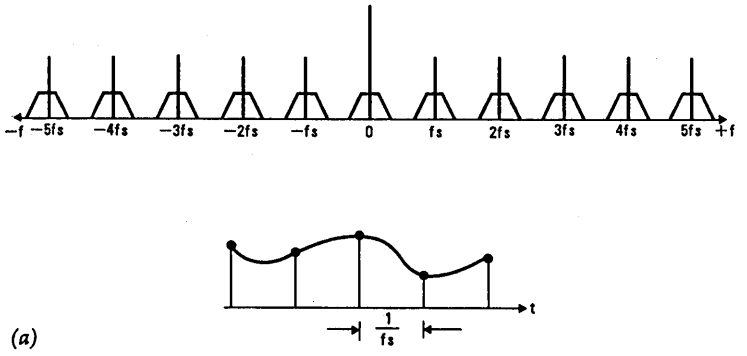


Fig. 2. Sampling theory describes how the original signal repeats at intervals (a). With three zeros placed between each of the original samples, the bandwidth increases (b). Three of the signal's images are now included in the spectrum.

The sampling rate of a DSP system must satisfy a number of criteria, beginning at the front end within the analog-to-digital (A/D) converter. In theory, it is only necessary to sample a signal at a rate greater than twice the signal's bandwidth.

In practice, this is not possible, because the band of interest may not extend from DC, and "brick-wall" antialiasing filters are not available. Nevertheless, certain techniques allow us to approach the theoretical minimum.

You—as a designer—could for example, shift the signal's frequency content, so that the band of interest extends from DC to ensure that the maximum signal frequency is equal to the signal's bandwidth. Quadrature sampling, another technique, extends this approach by allowing two analog-to-digital converters to share the work.

Finally, second-order sampling permits the signal to be examined at twice the bandwidth, even if there are higher frequencies present. In the latter case, the sampling rate reduction is achieved at the cost of taking twice as many samples; it also presents an onerous filtering problem to the first stage of the digital signal processor. A sampling rate reduction greater than two must be achieved if this technique is to be of any benefit.

Avoid Last Resorts

These sampling techniques are usually a last resort, used when the analog-to-digital conversion task would otherwise border on the impossible. There are only a few such applications in which the sampling rate will approach the theoretical minimum.

In other situations, there are advantages in oversampling the input signal: reduced specifications for antialiasing filters and

improved resolution from A/D converters. These advantages usually dictate an initially high sampling rate, one that is invariably maintained throughout the rest of the system, and which therefore results in inefficient use of the available resources.

A block diagram of a DSP system is usually drawn as a cascade of processing stages, each performing different operations on the signal. This is conceptually the simplest way to specify and analyze the processing, but it is rare for the individual blocks to map directly into separate pieces of hardware.

All digital signal processing algorithms are based on the same set of arithmetic operations, typically addition, multiplication, and multiplication/accumulation. An arithmetic processor specifically tailored to DSP applications can perform all the operations specified within the separate sections of the system block diagram.

The Fewer the Better

The objective of the system designer is to achieve all the processing required with the least number of processor units. Obviously, any reduction in the number of cycles needed to execute individual stages of the processing leads to overall savings. Typically, it may result in a reduction in processor units, an increase in the number of channels that may be accommodated, or higher processing quality in cases in which only a single DSP unit is used.

Intelligent Rationing

These benefits result from maintaining an efficient ratio between a signal's sampling rate and its bandwidth, as the overall frequency content of the signal is modified by its passage through the digital signal processing system.

Additional processing savings accrue from integrating the sampling rate changes directly into the processing stages themselves. These savings are most apparent in operations such as lowpass or narrowband filtering.

Modify Those Sample Rates

The sampling rate of a signal may be modified by either removing unnecessary samples or calculating and inserting additional samples. These techniques make up decimation and interpolation, respectively. In order to understand the effects on a signal of the interpolation and decimation processes, it is important to be familiar with the frequency domain representation of a sampled signal.

The Nyquist Criterion, so familiar to workers in the field of DSP, avoids aliasing distortion by specifying a minimum ratio between the sampling and maximum signal frequencies. In practice, this minimum ratio of two is often exceeded to alleviate the rolloff specification of the antialiasing filter.

In the frequency domain, the sampled signal's spectrum is repeated at periodic intervals, centered at integer multiples of fs (the sampling frequency). See Fig. 1. The bandwidth of these images must be less than fs or they will overlap, a condition termed aliasing.

Decimation and interpolation change fs and hence also alter the interval at which the images repeat. These images may, therefore, be moved selectively closer together or further apart.

Decimation Explained

Given that a signal is oversampled (that is, the maximum signal frequency is less than half the sampling frequency), the sampling rate may be reduced by eliminating unnecessary samples. At first sight, this may seem to entail simply removing a number of the samples from the time record.

To avoid catastrophic distortion of the signal, however, the time increment between each sample must be the same. This implies that the minimum achievable decimation ratio is a factor of two, corresponding to the elimination of every other sample.

Such a technique has been used for multistage filters and is termed "decimation by octaves." While it is possible to decimate in this manner by an integer ratio, it is rare to have such grossly oversampled signals in a real application.

Lowpass filtering leads to oversampled signals, as it is the high frequencies that are attenuated. Therefore, decimation techniques are usually associated with lowpass filters. Assuming that a signal is oversampled by a factor of less than two, lowpass filtering must occur before any decimation to avoid aliasing distortion. A simplistic approach would be to filter the signal and then decimate by discarding unwanted filter outputs.

If filter outputs are to be discarded, then why bother to calculate them in the first place? Unfortunately, recursive filter structures require all outputs to be calculated, since these outputs are fed back into the filter to influence subsequent outputs. Transversal filters do not suffer from this restriction and, consequently, permit more efficient lowpass decimating filters.

Assume that a lowpass filter of N coefficients filters a signal resulting in an output oversampled by a factor of P. Without decimation, the filter would have

COMPLEX HETERODYNING—HOW MUCH PROCESSING POWER?

Why complex heterodyning? Just what is it? For an answer, consider that digital systems take full advantage of quadrature frequency shifting techniques. Unlike analog systems, there is no possibility of frequency or phase drift.

As such, let's look at some real continuous signals, such as those diagrammed in the accompanying series of figures. Take a look at the "snapshot" of Figure Aa. Multiplying this continuous signal by the cosine of frequency fc (illustrated in Figure Ab), yields a spectrum (Figure Ad).

This spectrum displays significant aliasing between the two frequencies —fc and fc. However, multiplying the original signal by the sinusoid (Figure Ac) yields another spectrum (Figure Ae).

Then, combining the information (that shown in Figure Ad and Ae) allows the reconstruction of the original signal. It is shifted in frequency by fc. This technique is known as complex heterodyning.

Repeat Performance

In the sampled world, the original signal repeats at intervals of fs, the sampling frequency, as illustratea in Figure Af. The sampled equivalents of the SIN and COS have the form shown in Figures Ah and Ag, respectively.

Multiplication of the original signal by these two sinusoids yields the spectra (Figures Aj and Ai).

Again, the original signal information, with a shift in frequency of —fc, may be extracted (as indicated by Figure Ak).

From a practical point of view, the original real signal of sampling rate fs has been converted into a complex signal with the same sampling rate. This implies that twice as much processing will now be required to accommodate the real and imaginary components of the complex signal.

Closer examination reveals that the spectra (of Ai and Aj) contain duplicated information that may be removed with lowpass filters and decimation of both components of the complex signal (by a factor of two). The composite sampling rate of the complex components is the same as that of the original signal, while still extracting the spectrum of Figure Al.

Decimation

The process of decimation may be thought of as oc-

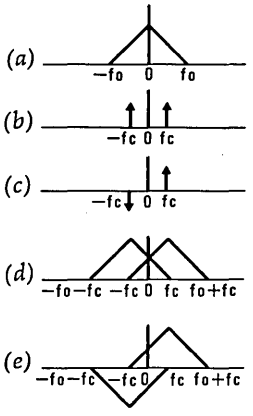
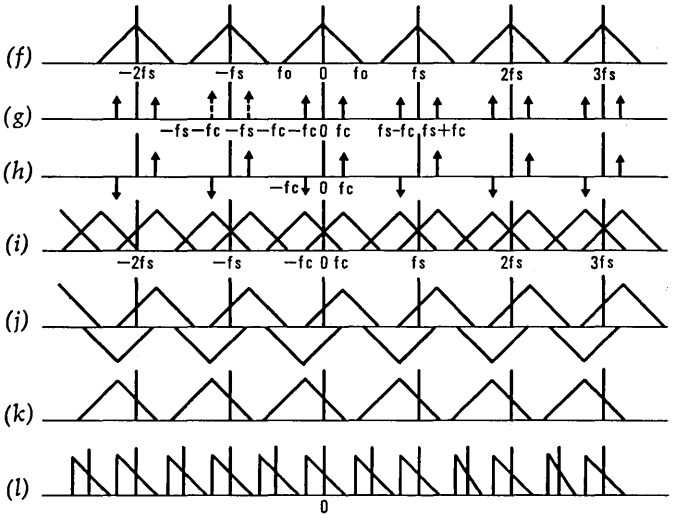


Figure A



curring in two stages: a lowpass filtering stage, followed by sampling rate reduction. The original signal (Figure Ba) of maximum frequency fo is sampled at a frequency, fs.

The signal is then passed through a lowpass filter with a cutoff frequency of fc. The resulting signal (shown in Figure Bb), may be decimated by a factor P to yield the spectrum of Figure Bc. This latter spectrum has a sampling rate of fs/P where P=2.

Interpolation

Next comes interpolation. Interpolation is the opposite of decimation and is again achieved in two stages. The first stage involves padding the sampled signal with (Q-1) zero-valued samples between each of the original samples.

This operation changes the spectrum of the original signal (that of Figure Ca) to that of Figure Cb, for Q=2. The spectrum (Figure Cb) has much the same form as that in Figure Ca, the major difference being labelling of the frequency axis.

*The effective sampling rate has been increased to Q*fs, as expected; however, the signal's spectrum now contains (Q-1) additional images. A lowpass filter, with cutoff frequency fo, will extract the desired portion of the spectrum (from Figure 8b) to yield the interpolated signal of Figure 8c. This lowpass filter must exhibit a gain of Q to compensate for the energy lost in filtering out the (Q-1) signal images.*

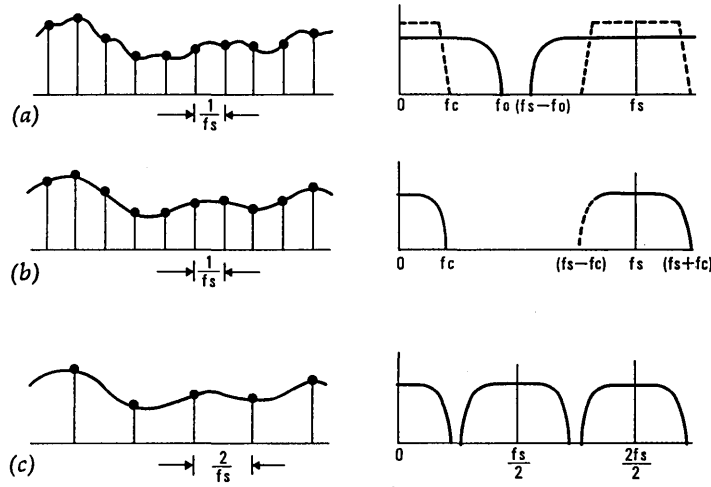


Figure B

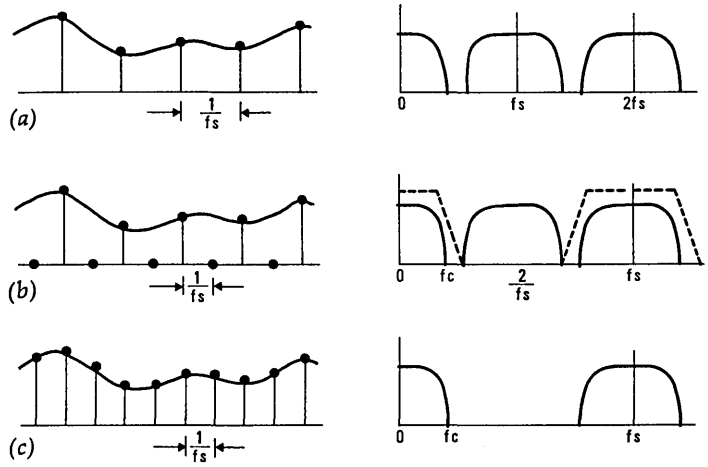


Figure C

to perform N operations per input point. With decimation, this is reduced to N/P. The result is only one output for every P input.

The throughput of this filter, as well as the rest of the processing downstream, is therefore increased by a factor of P. The filter does its job in fewer cycles, thereby reducing the processing burden on all subsequent operations—a true "win-win" scenario!

Interpolation, Too

Interpolation is the opposite of decimation. This operation involves increasing the sampling rate to create an oversampled signal. It cannot be used to recover signals distorted by aliasing, because this type of distortion is irreversible in the majority of situations.

Typical applications for interpolation include the reduction of the output reconstruction filter specification in audio systems or the smoothing of gaps between discrete line spectra in the output from a fast Fourier transform (FFT).

The first stage in interpolating a signal involves padding the signal with extra zero-valued samples. Since the previously stated restrictions about keeping the sampling interval constant apply, interpolation must be done by integer ratios. This is not a limitation, because interpolation is usually implemented to significantly increase the number of output samples.

The original signal, sampled at fs, repeats at intervals of fs as dictated by sampling theory (Figure 2a). Padding the signal with zeros increases the sampling rate, but also changes the form of the signal.

In the case of three zeros placed between each of the original samples (interpolating by a factor of four), the effective repetition frequency becomes four times fs. However, the signal's bandwidth increases in proportion, so that three of the signal's images are now included as part of its spectrum (Figure 2b).

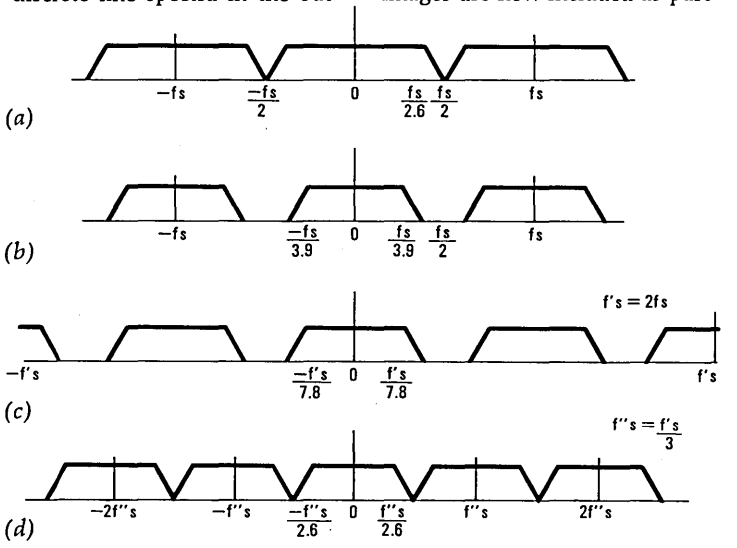


Fig. 3. Lowpass filtering in the decimation process needs a transition bandwidth of 15 percent of the interpolated signal's bandwidth (a). The original signal is lowpass filtered before interpolation (b). The new signal is interpolated to yield another signal (c); further decimation gives the output (d).

The frequency domain representation of the signal has not changed in shape; the padding has the effect of relabelling the frequency axis. To obtain the desired interpolation operation requires—surprisingly—the use of a lowpass filter. This filter removes the three unwanted images, yielding the original signal, oversampled by the desired factor of four.

At first sight this might appear to be a difficult lowpass filtering task, requiring operation at the increased sampling rate. But this need not be the case if the transversal filter structure is used.

Clearly, three out of four inputs to the filter are zero and, consequently, will not contribute to the filter output. These samples may, therefore, be skipped over when performing the filter operation saving valuable processor cycles. This technique allows the construction of efficient interpolating lowpass filters.

Let's assume that an N-point filter is used to interpolate a signal by a factor of Q. The simplistic approach would require N*Q operations for each one of the original samples. The interpolating lowpass filter only requires N operations per original sample, implying that the interpolating filter operates at the rate of the input data, regardless of the interpolation ratio applied.

Cascading Processes

The ability to decimate by non-integer ratios—in particular, ratios of between one and two—is essential to make effective savings in system applications. Non-integer ratios may be achieved by cascading interpolation and decimation processes.

To interpolate a signal by a factor Q, and then to decimate it by a factor P, changes the effec-

tive sampling rate to Q/P. While Q and P themselves must be integers, the ratio of one to the other can yield non-integer decimation ratios. The interpolation operation must, of course, precede the decimation to avoid aliasing distortion.

Two for One?

Again, both interpolation and decimation are effectively performed by a lowpass filter. What about the possibility of concatenating both operations into the same filter?

Assume that the interpolation ratio is Q. Then, for an N-point filter, N operations are required for every input point, despite the fact that Q*N output points are produced. The decimation implies that only one output is generated for every P inputs, so that only N/P operations per input point fulfill the requirements of both operations.

It may be tempting to assume that a large value for P, the effect of which could be offset by a correspondingly large value for Q, would significantly improve the efficiency of the decimation process by reducing the effective value of N/P. However, this will not work because N is related to Q and increases proportionally with any increase in Q.

The increase in N occurs because the lowpass filtering for both the interpolation and decimation is required to select a proportionally smaller percentage of the padded signal's bandwidth. The number of points in a filter is related to the percentage of the signal's total bandwidth that is taken up by the transition band of the filter.

As Q increases, the transition bandwidth remains constant, but the bandwidth of the padded signal increases; hence, N increases proportionally with Q. This

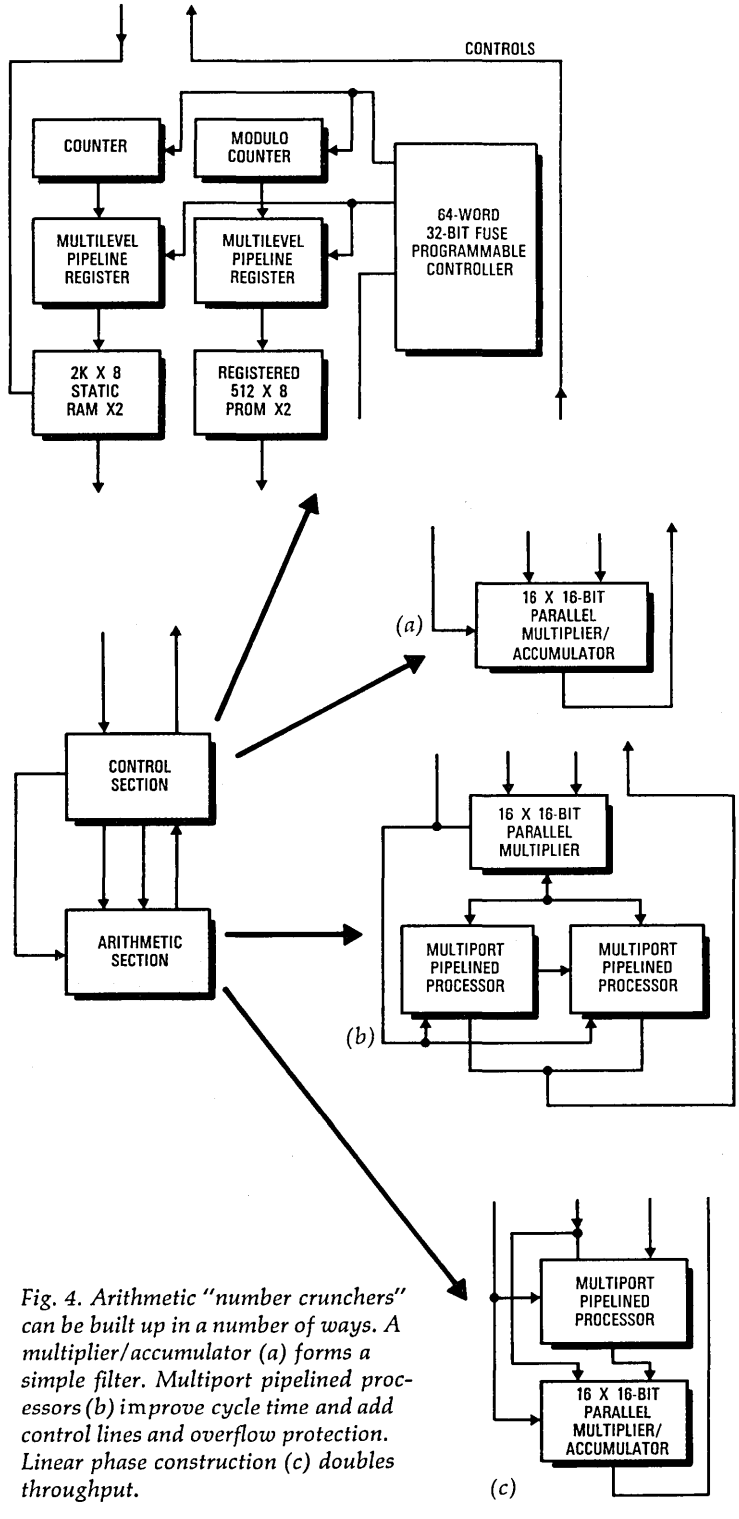


Fig. 4. Arithmetic "number crunchers" can be built up in a number of ways. A multiplier/accumulator (a) forms a simple filter. Multiport pipelined processors (b) improve cycle time and add control lines and overflow protection. Linear phase construction (c) doubles throughput.

means that the processing overhead involved in changing the sampling rate of signal is directly related to how closely the sampling rate of the decimated signal approaches the Nyquist rate. This is the design parameter that specifies the maximum width of the transition band.

Less Than You Imagine

A non-integer decimating process actually requires significantly less processing than a cursory examination would indicate. For example, if one of the processing stages in a system reduces a signal's bandwidth to 0.67 of its former value, then a decimation ratio of 1.5 may be applied without aliasing. This ratio may be achieved with values for Q and P of 2 and 3, respectively.

Assume a practical sampling rate Fs of Fs = 2.6 * Fo (where Fo is the maximum signal frequency, Figure 3a). The lowpass filter for the decimation process would need a transition bandwidth of fifteen percent of the interpolated signal's bandwidth. This gives N a value of 54. (The lowpass filtered signal is illustrated in Figure 3b.)

This new signal is interpolated to yield the signal shown in Figure 3c, which is then decimated back down to give the output of Figure 3d.

The value of P reduces the required number of operations per input sample from 54 to just 18. The output rate of the decimation stage is 0.67 of the input rate, resulting in a saving of 33 percent in the number of cycles required to execute all subsequent operations.

If the number of subsequent cycles exceed 54, then an overall saving will be achieved. For example, given 300 cycles of downstream processing, the total system savings would be:

$$[(1\text{-}1/1.5) * 300] - 18 = 82 \text{ cycles}$$

per input sample, which corresponds to 27 percent of all subsequent processing. This saving enables you to reduce system size and cost or process more channels.

Lowpass Filter It

If a lowpass filter is used to reduce the signal's bandwidth in the previous example, then as this function is duplicated by the decimation process, the original filter becomes superfluous. It may be discarded, saving 27 cycles per original input sample.

It should be clear that the decimating lowpass filter performs the same filtering operation as the original filter but in 18 instead of 27 cycles, a saving of thirty percent in itself. This saving is in addition to those in subsequent downstream processing that occur as a result of the reduced output rate. For these reasons, lowpass filters are usually implemented using decimating techniques.

A Look at Hardware

A decimating/interpolating stage, such as the one required in the system example, is constructed from an enhanced finite impulse response (FIR) filter. The filter structure performs a discrete convolution, according to the following formula:

$$Yk = \sum_{n=1}^{N} Cn * X(k-n+1)$$

where Xk and Yk are the filter inputs and outputs, respectively, and Cn are the coefficients; or, for the linear phase case:

$$Yk = \sum_{n=1}^{N/2} Cn * [X(k=n+1) + X(k-N+n)]$$

The arithmetic section of the FIR filter is unchanged from the non-decimating version, since the decimation is achieved by adapting the control and addressing sequence.

The control flexibility of the components used to construct the filter determine how much of the expected savings from decimation are realized in practice. The microprogrammable environment is ideally suited to this type of application.

Let's look at a hardware example (Figure 4). The arithmetic section performs the "number crunching" and may be tailored to suit processing requirements by varying the number of multipliers, multiplier/accumulators, and ALUs used.

If desired, the arithmetic section of a simple filter may be constructed with one multiplier/accumulator (Figure 4a). Alternatively, multipliers and multiport pipelined processors (Fig. 4b) may be used for arithmetic to improve the cycle time and provide the advantage of microprogrammable control lines and overflow protection. An efficient linear phase construction may be achieved with devices like the AMD Type Am29501 multiport pipelined processor and Am29510 16 X 16-bit parallel multiplier/accumulator combination, for example (Figure 4c), effectively doubling the throughput.

The control section sequences the operations and selects the correct data points and coefficient values for the arithmetic section. It does so by defining the appropriate addresses within the data RAM and the coefficient PROM.

For the direct or linear phase implementation of a filter, count-

ers enhanced by hardware, such as multilevel pipeline registers, can generate addresses. For decimating or interpolating filters, the addressing sequence becomes more complex and requires additional modulo counters; this task also requires the overall control flexibility offered by a device such as the AMD Type Am29PL141 fuse-programmable controller.

Control circuitry eliminates redundant cycles that would result from zero inputs or unwanted outputs. The method is different for the decimating and interpolating stages of the filter.

In interpolation, a normal filter sequences through all of its coefficients, multiplying each by a corresponding data sample. For a padded signal, many of these data points are zero and, consequently, do not contribute to the output.

Zero padding may be achieved by incrementing the data addresses by one, and the coefficient addresses by an interval equal to the interpolation ratio. This padding technique has the beneficial side effect of automatically avoiding all redundant operations.

However, it is necessary to keep track of the location of the first non-zero data point, since this defines the first coefficient address to be used by the filter. Any subsequent decimation operations will change this address.

To avoid redundant cycles in decimation, filter outputs that would be subsequently discarded are simply not calculated. A filter without an interpolating stage accomplishes this by writing more than one point at a time into the filter's cyclic buffer. For example, writing in two new samples for each calculation of an output gives a decimation ratio of two.

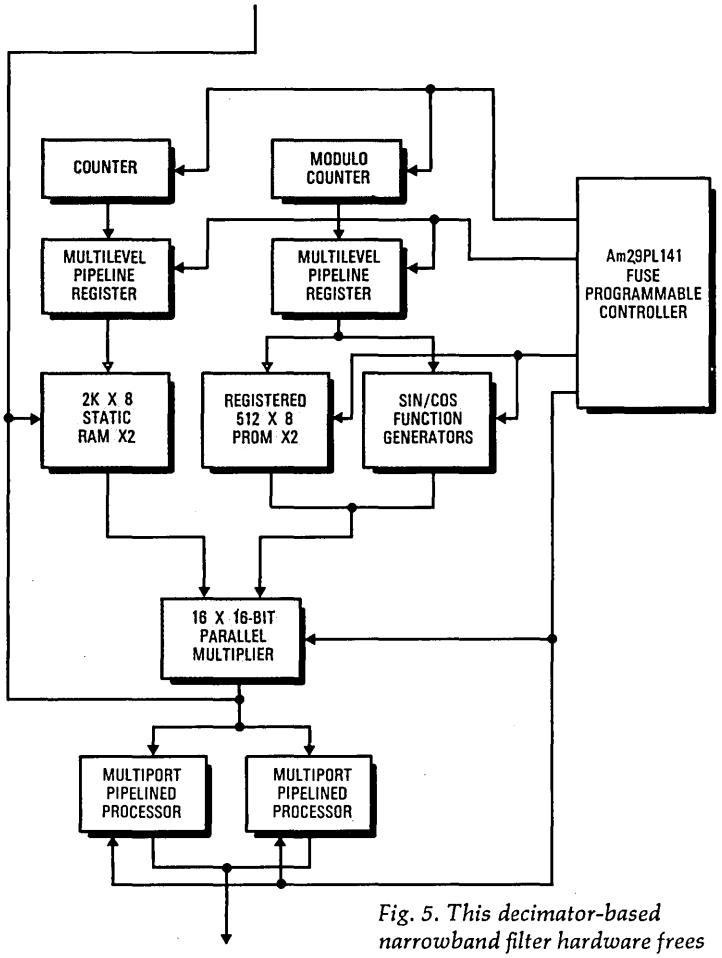


Fig. 5. This decimator-based narrowband filter hardware frees the multiplier to perform heterodyning without overhead. No filtering occurs while samples are written into the cyclic buffer.

For a filter that incorporates an interpolation stage, the decimation could require the writing of zeros or samples (or both) into the cyclic buffer. Since zeros are not explicitly written into the STORE, the coefficient START address is incremented to compensate.

The coefficient START address is RESET to zero every time a true sample is written into the STORE. Depending upon the applicable decimation and interpolation ratios, a varying number of true samples and apparent zeros will be written into the cyclic buffer between each output from the filter. The relevant coefficient START address may be calculated by incrementing a

modulo counter by the decimation ratio. The count modulus is equal to the interpolation ratio.

Very efficient narrowband filters may be constructed, using the same techniques that resulted in efficient lowpass filters; this is because with narrowband filters, significant decreases in the signal's bandwidth occurs, allowing a high decimation ratio.

A narrowband filter can be constructed from a lowpass filter with half the bandwidth of the equivalent narrowband filter. The input signal must first be shifted to baseband, with the signal frequency lying in the center of the bandpass filter shifted to be at DC, so that it fits within the passband of the lowpass filter.

The shift operation can be done with a complex heterodyne stage that simply multiplies the input signal by a complex frequency (SIN and COS) equal to the center frequency of the bandpass filter. This complex, frequency-shifted signal is then filtered by a decimating lowpass filter, interpolated, and shifted back up to the original band with the same complex heterodyne technique. The signal may, of course, be shifted back up to any band desired.

A given narrowband filter may have many different center frequencies, each determined by the complex frequency employed in the heterodyne stage. The same filter may be used to divide a wide bandwidth signal up into a number of smaller blocks by stepping the heterodyne frequency in increments equal to the filter's bandwidth. Since the bandwidth of the narrowband filter depends on the bandwidth of the lowpass filter, it is independent of the effective center frequency. A number of lowpass filters may be stored within the arithmetic processor, allowing the narrowband filter's center frequency and bandwidth to be changed at will.

The Complex Heterodyne

The complex heterodyne operation requires two cycles per input sample; these are used to multiply the sample by digitized values of SIN and COS, to yield the imaginary and real components of the basebanded complex signal.

The values of SIN and COS derive from the sampling of a complex sinusoid of the required frequency at a rate equal to the effective sampling rate of the signal. This lowpass filtering leads to an advantageous reduction in the bandwidth of the signal, which may be exploited by decimating the signal accordingly. There is an additional inherent decimation factor of two introduced when the signal is translated into a complex representation.

Shifting to restore the signal back to its original band is again achieved through complex heterodyning. This operation must, of course, be performed after the signal has been interpolated back to the original sampling rate, to avoid aliasing distortion.

For example, assume that a narrowband filter is required to select ten percent of the band of an input signal with transition bands occupying an additional ten percent. A direct implementation of this filter would operate at the input sampling rate and require about 160 cycles per input sample to process.

The decimating equivalent would require two cycles per input point, operating at the input sampling rate, to perform the complex heterodyne. The baseband signal occupies only 15 percent of the original signal's bandwidth, and therefore may be decimated by a factor of 6.7 resulting from values of P and Q of 20 and 3, respectively. After the initial interpolation, the transition band occupies only 3.3 percent of the interpolated signal's bandwidth, suggesting that the lowpass filter will require 242 taps.

The effective processing rate of this filter is N/P cycles per input point, resulting in an overall requirement of twelve cycles for each of the original input points. Two filters are required for the real and imaginary channels, resulting in a total processing requirement of 26 cycles per input, a saving of about eighty percent.

To shift the filtered signal back up to its original band and sampling rate at this point would not achieve any overall savings, because there will be no net decrease in the sampling rate. If further processing of the filtered signal is required, then all stages that now operate at the reduced sampling rate will benefit from the eighty percent saving in processing.

Narrowband filter hardware (Figure 5) is also based on the decimator. The complex heterodyne operations are best implemented by a multiplier using devices such as the AMD Type Am29517 multiplier and Am29501 multiport pipelined processor as the arithmetic section.

No filtering will occur while samples are being written into the cyclic buffer, so that the multiplier is free to perform the heterodyne operation without any overhead. The necessary complex frequency coefficients may be obtained by incrementing the addresses of a pair of Am29526 sine and Am 29527 cosine generators. The ICs share the same address space and bus as the registered PROMs that contain the filter coefficients. ■

DATA FILE 130

DSP BUILDING BLOCKS ALLOW RESOURCE OPTIMIZATION.

by
Bernard J New
Manager
Product Planning and Applications
AMD
901 Thompson Place
Sunnyvale, CA. 94088

Introduction

The essential generality of general purpose computing usually prevents optimization of the processor. In digital signal processing, however, this is not usually the case. Even in multi-purpose signal processors, the algorithms to be implemented will have many similarities. In particular, they will be repetitive, intensive in both arithmetic and memory operations, and branch infrequently.

The repetitive, non-branching characteristic is exploited through extensive use of pipelining, more than would be considered advantageous in a general purpose machine. The large and predictable number of arithmetic operations and memory accesses permits an arithmetic processor to be contructed with resources balanced to match the problem at hand.

The construction of this processor will not normally allow it to be multiplexed effectively with the task of address generation. As the addresses are often the only variation in successive iterations of the program and follow a well-defined pattern, it is desirable that they are generated in an independent processor which operates concurrently with the arithmetic processor.

The Am29500 family of digital signal processing and array processing products provides devices for use in both areas. In particular, for use in the arithmetic processor, the family includes the Am29501 Multi-port Pipelined Processor and the Am29516/7 16-bit Parallel Multipliers. These constitute two major resources. The parallel multiplier is essential to signal processing calculations, and the Multi-port Parallel Processor complements this with register and ALU facilities. This device's three ports provide the communication necessary for efficient use of multiple elements.

The third major resource to be managed in the arithmetic processor is the memory. Using the fast Fourier transform (FFT) as an example, this paper describes an approach by which an algorithm may be analyzed and an appropriate allocation of resources made. It should be stressed, however, that the Am29500 family is of general application, and is not limited to the FFT.

The Balancing Act

The objective in optimizing an arithmetic processor is to provide facilities in proportion to their usage in the algorithm being implemented. In this way, the processor is balanced with no one resource lying idle while another completes its task, and all are used at peak efficiency. Increasing just one resource will not necessarily increase performance due to the creation of an imbalance.

Consider the fast Fourier transform. This comprises the repeated evaluation of what is commonly known as the FFT "Butterfly." This is shown diagramatically, together with its formulae, in Figure 1. There are two input data points, A and B, which are complex numbers. These are combined together, and with a complex coefficient, W, to form two outputs, A' and B'.

Inspection of the formulae shows that a single implementation would require one complex addition, one complex subtraction and one complex multiplication. Also five complex memory operations are required; three reads for A, B and W, and two writes for A' and B'. In terms of real operations, this reduces to four multiplications, six addition/subtractions and ten memory transactions. It is to these requirements that an FFT processor must be matched.

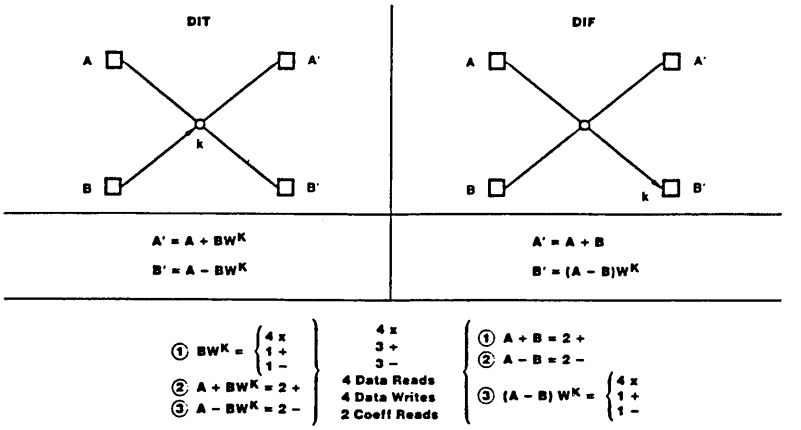


FIGURE 1: The FFT Butterfly

Moving the Bottleneck

From the above analysis it may be seen that if all operations take equal time, the throughput is limited by the memory requirement, which makes it impossible to perform butterflies more frequently than every ten cycles. However, this may be reduced by exploiting two factors, which are common to many signal processing algorithms.

Firstly, the memory may be re-organized to reflect the complex nature of the data. Making the memory twice as wide but only half as deep does not increase the size of the memory, but allows for simultaneous access to real and imaginary parts, effectively doubling the memory bandwidth. Secondly, it may be noted that the multiplications are performed between data and a coefficient. If the coefficients are stored in a separate memory, they may be accessed concurrently with the data.

These changes reduce the number of data memory accesses to four. However, the butterfly will still require six cycles, as the throughput is now dominated by the ALU requirement. The bottleneck has moved from the memory to the ALU.

For Each Butterfly	Memory Accesses	Add/Subtract	Multiply
	8	6	4

Resource / # Cycles	Memory Buses		ALU		Multipliers	
	#	Usage	#	Usage	#	Usage
8	1	8/8	1	6/8	1	4/8
6	2	4/6	1	6/6	1	4/6
4	2	4/4	2	3/4	1	4/4
3	4	2/3	2	3/3	2	2/3
2	4	2/2	4	1.5/2	2	2/2

TABLE 1: FFT Resource Comparison

Measuring Efficiency

A useful measure of efficiency is the proportion of time each resource is active. In the six cycle butterfly described above, the ALU will be used 100% of the time. The memory and multiplier will each be used on only four of the six cycles, 67% of the time. This is summarised in the first line of Table 1. This base system contains one memory, one ALU and one multiplier.

As the multiplier is an expensive resource, it is desirable to utilize it more efficiently. To do this necessitates adding more capacity to the current bottleneck, the ALU. The second ALU reduces the necessary ALU cycles to three, and moves the bottleneck to both the memory and the multiplier, each requiring four cycles. In this case the memory and multiplier efficiency is 100%, and the ALU efficiency is 75%.

As shown in the table, further improvement requires that both the memory and the multiplier be duplicated. This gives a situation similar to the base system, but with twice the hardware resulting in twice the throughput.

A Reasonable Solution

In theory this procedure could be repeated until four memories, four multipliers and six ALUs allowed a butterfly to be completed every cycle. However it is unlikely that this solution would be practical.

The problem encountered is in partitioning the memory such that two reads and two writes can be performed simultaneously. This partitioning must be consistent with the data flow of the

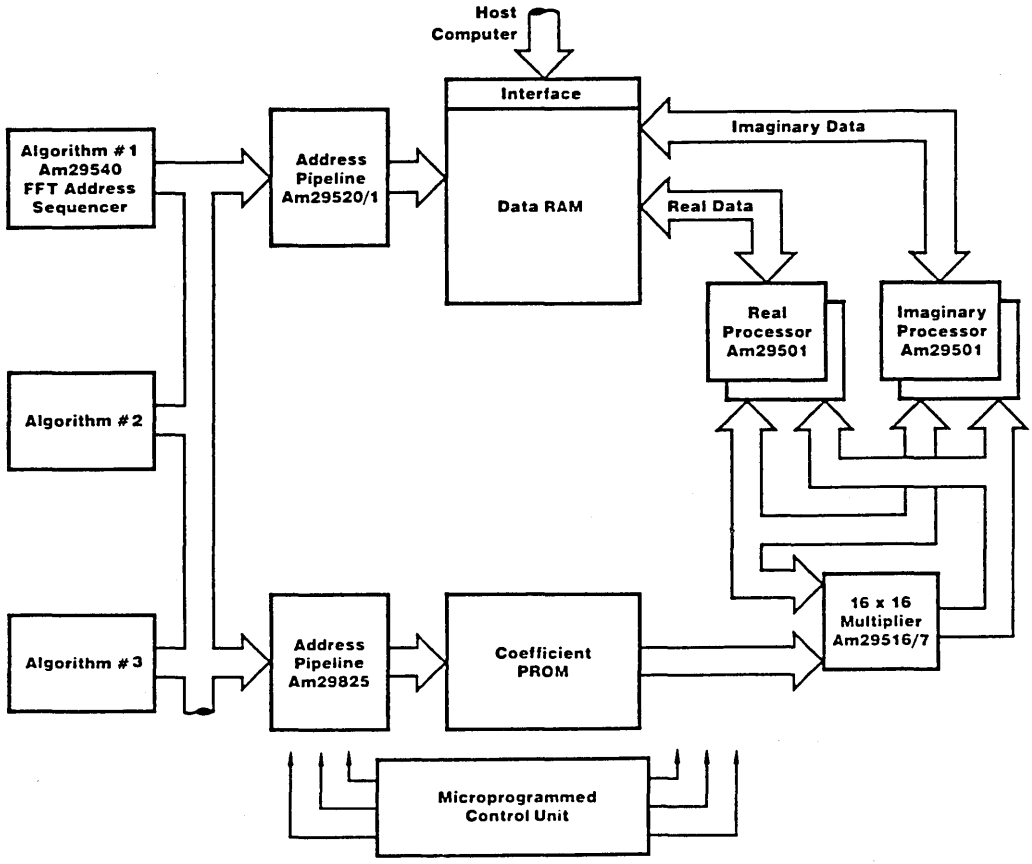


FIGURE 2: FFT Processor Architecture

FFT. Even the base system is not without problems. This requires that the result of a complex memory read be loaded into a real ALU. Also the inclusion of additional hardware to achieve more speed than is necessary for the application is obviously undesirable.

When comparing architectures which trade a doubling of throughput for a doubling of hardware, as occurs in the table, system integrity may be a deciding factor. While it may be convenient to build a single large, fast machine, if latency requirements allow, it may be better to alternate two slower machines. This allows for reduced operation, rather than failure, if one of the machines should fail.

The table obtained from the above analysis should be viewed only as an initial survey of the options available. Even after selecting a likely candidate is is necessary to show that the algorithm can be programmed into the processor in the number of cycles anticipated.

Overlapped Programming

Let us assume that the single memory, two ALU, single multiplier architecture has been selected. This is shown in Figure 2. The objective is to implement the FFT butterfly in this architecture in four cycles.

A simple inspection of the algorithm shows that in order to generate and store the real part of A', it necessary to read A and B, complete the real part of the multiplication, perform an addition and write the result to memory. Allowing for maximum concurrency this requires five cycles, and it obvious that the full program would be longer than that.

However, this does not imply that the goal of four cycle throughput is impossible, only that the latency will be greater than four cycles. The resources in the processor are such that if the program takes more than four cycles they will be idle for part of the time. While causality does not allow these resources to be applied to the current iteration of the program, the program repeats, and the resources may be applied to previous or following iterations.

This leads to a situation where an iteration of the program commences before the previous ones are complete, and the iterations overlap. This is similar to pipelining except that whole programs are involved and the hardware is multiplexed between overlapped programs.

The completed program for the FFT butterfly is shown in Table 2a. Ten cycles are required to complete this program. Table 2b shows the instruction stream when this program is

		Real							Imaginary							Multiplier	
Step	DIO	ALU	A_1	A_2	A_3	B_1	B_2	B_3	ALU	A_1	A_2	A_3	B_1	B_2	B_3	MIO	MULT
1																	
2																	
3																	
4																	
5	Read B					DI							DI				
6													H			B_R	
7	Read A		DI							DI			H				B_RW_R
8		A1−MSP	H		MSP		ALU			H						B_I	B_RW_I
9			H		H		H		A1−MSP	H		MSP		ALU			B_IW_R
10		A1 + A3		ALU			H		B2−MSP	H		H		ALU	MSP		B_IW_I
11		B2+MSP		H		MSP	ALU		A1 + B3		ALU	H		H			
12	Write B2			H		H			A2 + A3		ALU						
13		A2 − B1		ALU							H						
14	Write A2																
15																	
16																	
17																	
18																	

TABLE 2a: The Butterfly Program

		Real							Imaginary							Multiplier	
Step	DIO	ALU	A_1	A_2	A_3	B_1	B_2	B_3	ALU	A_1	A_2	A_3	B_1	B_2	B_3	MIO	MULT
1	Read B	A2 – B1	H	ALU	H	DI	H		A1 – MSP	H	H	MSP	DI	ALU			B_IW_R
2	Write A2	A1 + A3		ALU			H		B2 – MSP	H		H	H	ALU	MSP	B_R	B_IW_I
3	Read A	B2 + MSP	DI	H		MSP	ALU		A1 + B3	DI	ALU	H	H	H			B_RW_R
4	Write B2	A1 – MSP	H	H	MSP	H	ALU		A2 + A3	H	ALU					B_I	B_RW_I
5	Read B	A2 – B1	H	ALU	H	DI	H		A1 – MSP	H	H	MSP	DI	ALU			B_IW_R
6	Write A2	A1 + A3		ALU			H		B2 – MSP	H		H	H	ALU	MSP	B_R	B_IW_I
7	Read A	B2 + MSP	DI	H		MSP	ALU		A1 + B3	DI	ALU	H	H	H			B_RW_R
8	Write B2	A1 – MSP	H	H	MSP	H	ALU		A2 + A3	H	ALU					B_I	B_RW_I
9	Read B	A2 – B1	H	ALU	H	DI	H		A1 – MSP	H	H	MSP	DI	ALU			B_IW_R
10	Write A2	A1 + A3		ALU			H		B2 – MSP	H		H	H	ALU	MSP	B_R	B_IW_I
11	Read A	B2 + MSP	DI	H		MSP	ALU		A1 + B3	DI	ALU	H	H	H			B_RW_R
12	Write B2	A1 – MSP	H	H	MSP	H	ALU		A2 + A3	H	ALU					B_I	B_RW_I
13	Read B	A2 – B1	H	ALU	H		H		A1 – MSP	H	H	MSP	DI	ALU			B_IW_R
14	Write A2	A1 + A3		ALU			H		B2 – MSP	H		H	H	ALU	MSP	B_R	B_IW_I
15	Read A	B2 + MSP	DI	H		MSP	ALU		A1 + B3	DI	ALU	H	H	H			B_RW_R
16	Write B2	A1 – MSP	H	H	MSP	H	ALU		A2 + A3	H	ALU					B_I	B_RW_I
17	Read B	A2 – B1	H	ALU	H		H		A1 – MSP	H	H	MSP	DI	ALU			B_IW_R
18	Write A2	A1 + A3		ALU			H		B2 – MSP	H		H	H	ALU	MSP		B_IW_I

TABLE 2b: The FFT Instruction Stream

restarted every four cycles. Through careful programming this can occur without the programs interfering. It may be seen from this table that the three major resources, memory, ALU and multiplier are used 100% of the time. This differs from the original estimate of 75% utilisation of the ALU, the extra 25% being accounted for by "inefficient" use in the program. However, the extra use of the resource is free, and not using it would at least have increased the latency.

Inspection of Table 2b also shows the instruction stream to be periodic, repeating every four cycles. These four cycles contain the information for the complete ten cycle butterfly program, and constitute the inner loop which must be executed to implement the FFT. At the beginning and end of the FFT, previous and following iterations are absent and the instruction stream becomes aperiodic. This must be accommodated by segments of linear code derived from the instruction stream, which fill and empty the overlap pipeline. In this case the program could be written to give the desired throughput. If this had not been the case, it would have been necessary to move to a "faster" architecture.

The only outstanding task is to generate the addresses to provide the data stream into the arithmetic processor. For the specific case of the FFT this may be effected using the Am29540 FFT Address Sequencer. In other cases an address computer may be designed and programmed using the techniques described above.

Conclusion

Using the FFT as an example, a method has been described whereby a digital signal processor may be optimized through resource management. This technique is applicable to architectures using building block components, such as the Am29500 Family. Indeed it was about this family that it was developed. Processors designed in this way exhibit maximum usage of components included.

A New Approach to Floating Point DSP

Robert Perlman
senior engineer, Product Planning
Advanced Micro Devices
Sunnyvale, CA 94088

ABSTRACT

A new high-speed, single-chip floating point processor, the Am29325, is introduced; this processor incorporates features of interest to those implementing high-performance digital signal processing systems. Processor architecture is described, and the advantages of the architecture for DSP and array processing applications are discussed. Typical small- and large-system designs are presented.

INTRODUCTION

Floating point arithmetic engines are natural candidates for very-large-scale integration, due to the popularity of the function, and to the large amounts of design time and circuit board space needed to implement such a function in SSI and MSI. Early efforts to integrate floating point operators in a single chip or chip set usually resulted in serial-parallel designs which, while considerably faster than software floating point implementations, did not approach the speeds of fully parallel, dedicated hardware designs.

Recent improvements in process technology have made possible, for the first time, the joining of combinatorial floating point addition/subtraction and multiplication functions in a single VLSI device. The Advanced Micro Devices Am29325 Floating Point Processor contains all hardware necessary to perform high-speed, 32-bit floating point addition, subtraction, multiplication, and format conversion operations, in either IEEE or DEC floating point formats. The device also contains a flexible 32-bit data path, with facilities for local operand storage.

The integration of three elements - a combinatorial adder/subtractor, combinatorial multiplier, and data path - marks the fundamental difference between the Am29325 and previous floating point implementations. By combining these functions, the design addresses not only the problem of implementing fast floating point operators, but also the equally important problem of efficiently transferring operands from one operation to the next. The data path architecture is optimized for performing often-used arithmetic sequences, such as multiplication- accumulation and Newton-Raphson division.

The Am29325 is fabricated with the IMOX-S (for Ion-iMplantation, OXide isolation with Scaling) process, a refinement of earlier AMD bipolar processes. IMOX-S has a feature size of 1.5 microns; three layers of metal are used for interconnects. The Am29325 die contains 48,000 devices on 129,000 square mils of silicon, and is packaged in a 144-lead pin-grid-array. Standard cell techniques were used to reduce design time and simplify chip layout. Improvements in turn-around time were significant: custom design of the Am29116, a 16-bit bipolar microprocessor, took 51 months, while design of the Am29325, a device three times as large, took only 31 months.

The floating point processor is the first in a series of general-purpose, microprogrammable devices primarily intended for 32-bit systems. Other family members include the Am29331 microprogram sequencer, the Am29332 ALU, the Am29323 32-by-32-bit fixed-point multiplier, and the Am29334 register file.

Am29325 ARCHITECTURE

The Am29325 comprises a high speed floating point arithmetic unit, a status flag generator, and a 32-bit data path (fig. 1).

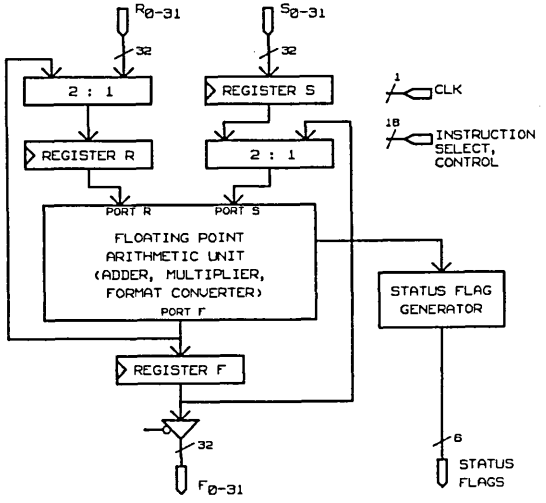


Fig. 1: Am29325 Floating Point Processor block diagram.

Arithmetic unit - The three-port, combinatorial arithmetic unit contains a high speed adder/subtractor, a 24-by-24-bit multiplier, an exponent processor, and other logic needed to implement floating point operations. Two input ports, R and S, provide operands for the instruction to be performed. The result of an operation appears on output port F.

The arithmetic unit executes one of eight instructions (table 1). Three of the instructions - R PLUS S, R MINUS S, and R TIMES S - add, subtract, and multiply 32-bit floating point numbers. A fourth instruction, 2 MINUS S, subtracts 32-bit floating point operand S from 2. The 2 MINUS S instruction is used to perform Newton-Raphson division, a means of calculating the quotient A/B. Unlike conventional division, in which quotients are calculated with a series of subtractions and shifts, the Newton-Raphson division algorithm first calculates the reciprocal (1/B) using an iterative equation, then computes the quotient by post-multiplying the reciprocal by A.

The remaining four instructions perform data format conversions. Instructions INT-TO-FP and FP-TO-INT convert between floating point and 32-bit, 2's complement integer formats. The IEEE-TO-DEC and DEC-TO-IEEE instructions convert between IEEE and DEC floating point formats.

Instructions may be performed in either of two single-precision floating point formats - the IEEE format, as specified in proposed standard P754, draft 10.0 (ref. 1), or the DEC F format (ref. 2). These formats are similar, each having an 8-bit biased exponent, a 24-bit significand comprising a 23-bit mantissa appended to an implied or "hidden" MSB, and a sign bit. There are, however, a number of differences between IEEE and DEC floating point conventions, in both the format and the manner in which operands are handled during the course of an operation. These differences are automatically accounted for when the desired format is selected.

MNEMONIC	OPERATION
R PLUS S	Add floating point operands R and S
R MINUS S	Subtract floating point operand S from floating point operand R
R TIMES S	Multiply floating point operands R and S
2 MINUS S	Constant floating point subtraction for Newton-Raphson division (see text)
INT-TO-FP	Convert floating point operand R to integer
FP-TO-INT	Convert integer operand R to floating point
IEEE-TO-DEC	convert IEEE floating point operand R to DEC floating point format
DEC-TO-IEEE	convert DEC floating point operand R to IEEE floating point format

Table 1: Floating point arithmetic unit operations

The arithmetic unit implements four IEEE-mandated rounding modes that map the infinitely precise result of a calculation to a representable floating point value. An additional VAX-compatible rounding mode is provided for users of the DEC floating point format.

Status flag generator - The status flag generator produces six flags that report operation status. Four of the flags report exception conditions stipulated in IEEE standard P754. The first of these, the INVALID flag, indicates that an operation does not have a sensible answer; multiplying infinity by zero is one example of an invalid operation. Operations producing results either too large or too small to be represented in the selected floating point format are identified by the second and third exception flags, UNDERFLOW and OVERFLOW. The fourth exception flag, INEXACT, indicates that the result of an operation is not infinitely precise. Two additional flags not called for in the IEEE standard, ZERO and NAN, identify zero-valued or non-numerical results.

Data path - The integrated data path comprises two input buses, a three-state output bus, and two data feedback buses, all 32 bits wide. Operands enter the Am29325 through input buses R_{0-31} and S_{0-31}; results exit through three-state output bus F_{0-31}. Each of the R, S, and F buses has a 32-bit edge-triggered register for data storage. An independent clock enable is provided for each register, so that new data can be clocked in or old data held. Input registers R and S, and output register F can be made transparent independently. When all three registers are made transparent, the Am29325 operates in a purely combinatorial "flow-through" mode.

The two feedback data paths transport processor output operands back to the inputs. The first feedback path routes data from the output of the arithmetic unit to a 32-bit multiplexer at the input of register R; the multiplexer selects the operation result or R_{0-31}. The other feedback path carries the output of register F to a second 32-bit multiplexer, which selects either register S or register F as the input for port S of the arithmetic unit.

To allow easy interface with a variety of 16- and 32-bit systems, buses R, S, and F can be programmed to operate in one of three I/O modes. The first and most straightforward of these is the 32-bit, 2-input-bus mode; in this mode, the R and S buses are configured as independent 32-bit input buses, the F bus as a

32-bit output bus. The second I/O option is a 32-bit, single-input-bus mode, in which the R and S operands are taken from a single 32-bit input bus on alternate clock edges. For the third option, a 16-bit, two-input-bus mode, the R, S, and F buses are 16 bits wide. Thirty-two-bit operands are placed on these 16-bit buses by time-multiplexing the 16 MSBs and LSBs of each data word during alternate halves of the clock cycle. Internal data paths and registers remain 32 bits wide when this 16-bit I/O mode is selected.

ARCHITECTURAL ADVANTAGES FOR DSP APPLICATIONS

The architecture of the Am29325 offers several advantages to the implementor of DSP and array processing systems:

Efficient data routing - Three aspects of the Am29325 architecture contribute to efficient data routing. First, placing the adder/subtractor and multiplier on the same die eliminates the shuffling of data between separate adder/subtractor and multiplier chips. Minimizing chip-to-chip communication is an important consideration in high performance system design, since, in VLSI-based systems, the time needed to transfer data between chips can often limit maximum operating speed.

Second, the on-board data paths allow the intermediate result of a calculation to be routed to the input of the floating point arithmetic unit, for use as an input operand in the next phase of the calculation. This feature not only keeps data on-chip, but also makes an external implementation of a similar data path unnecessary. Such an external data path would be expensive, both in components and circuit-board real estate; implementing the two 32-bit multiplexers alone would consume over a dozen MSI devices.

Third, the absence of pipeline delays in the floating point arithmetic unit makes it possible to use the result of one calculation as the input operand for the very next calculation, a crucial feature when implementing algorithms with tight data feedback loops. Users of floating point processors with pipeline delays have one of two choices when implementing such an algorithm - they can either halt the operation while waiting for the desired result to drop out of the pipeline, thus reducing computational efficiency, or can interleave different sets of calculations to keep the arithmetic unit busy, at the cost of more complicated programming. Using a zero-pipeline-delay arithmetic unit avoids both of these unappealing choices.

Am29325 data routing efficiency is best appreciated by considering the manner in which multiplication-accumulation is performed. In a typical multiplication-accumulation calculation, N input terms x_i are multiplied by coefficients k_i. These products are then added, producing the weighted sum

$$s = \sum_{i=0}^{N-1} k_i x_i$$

Multiplication-accumulation is performed in a two-step loop, with two additional steps for initialization (fig. 2a-d). To initialize the process, data and coefficient values x_0 and k_0 are clocked into registers R and S (fig. 2a). Next, the values x_0 and k_0 are multiplied, and the product placed in register F; at the same time, data and coefficient values x_1 and k_1 are clocked into registers R and S (fig. 2b). In the first step of the multiplication-accumulation loop, values x_1 and k_1 are multiplied, and the product placed in register R (fig. 2c). In the second step, products x_1*k_1 and x_0*k_0 are added, and their sum placed in register F; x_2 and k_2 are placed in registers R and S (fig. 2d).

The two loop steps are then repeated for as many iterations as needed to complete the calculation. The internal data paths wrap back products and accumulations, thus keeping the arithmetic unit busy with a multiplication or addition every clock cycle; a new multiplication-accumulation is performed every two clock cycles. Partial results remain on-chip until the multiplication-accumulation is completed.

High I/O bandwidth - The three 32-bit I/O buses provide high-bandwidth access to the floating point arithmetic unit. When the device is operated in the 32-bit, two-input-bus I/O mode, no multiplexing of I/O buses is required, thus improving system speed and easing critical timing constraints.

Transparent operation - In many applications, the R, S, and F registers will be used to store an operation's inputs and outputs; it is in this register-to-register mode that the Am29325 operates the fastest. In some applications, however, it may be desirable to bypass the internal registers, either because system requirements dictate a data path structure substantially different from that provided, or because the floating point operations must be concatenated with other combinatorial functions. These situations can be accommodated by making all three registers transparent, turning the floating point processor into a purely combinatorial device; this "flow-through" mode of operation would not be possible if the Am29325 used multiplexed I/O.

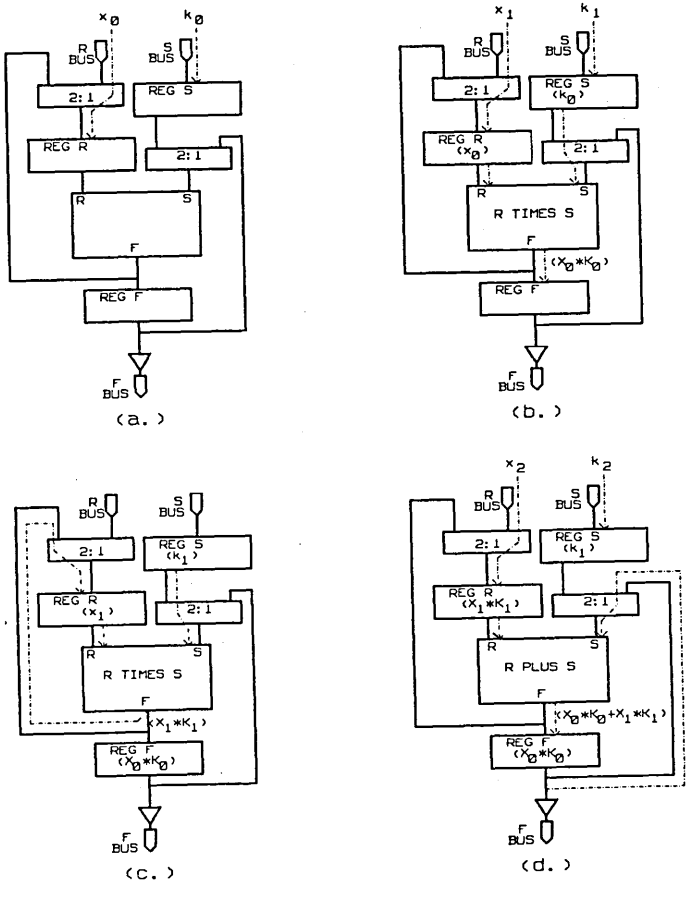


Fig. 2(a-d): Performing floating point multiplication-accumulation with the Am29325.

SYSTEM DESIGN

A block diagram for a typical small system design is shown in fig. 3. The system consists of an Am29325, an Am29334 four-port register file, data memory, coefficient memory, microprogrammed controller, clock generator, and host system interface. Although small enough to fit on a single circuit board, this system contains all the elements needed for floating point digital-signal and array processing.

Because of its three-bus I/O structure and internal feedback paths, the Am29325 can be used to advantage in both cascade and parallel configurations. Fig. 4 illustrates a simple cascade system, a variation on the previous architecture. In this system, the output port of one floating point processor feeds the input port of another. This arrangement is particularly advantageous when performing high-speed

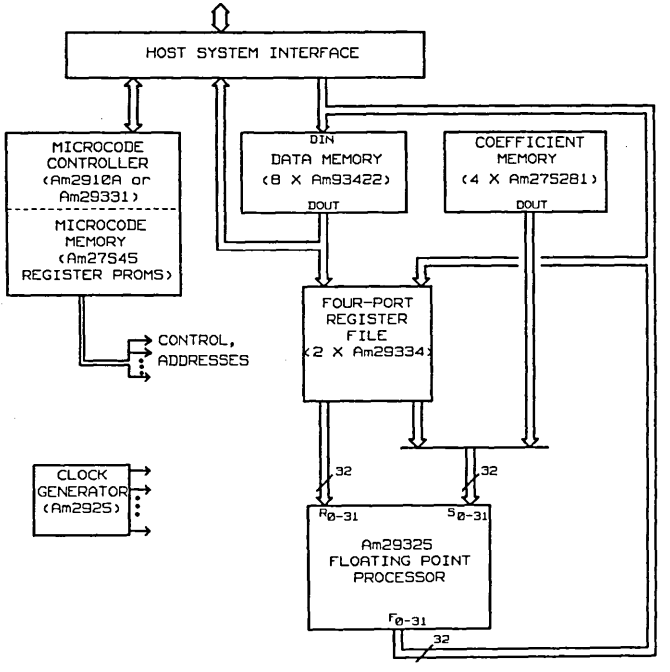


Fig. 3: Typical small-system design.

multiplication-accumulation; the first Am29325 forms products, while the second computes the accumulation in parallel. The accumulation is performed using a feedback data path in the second part - no external feedback path is necessary. By doing the multiplications and additions in parallel, the effective throughput rate is one clock per multiplication-accumulation, twice that of the system shown in fig. 3.

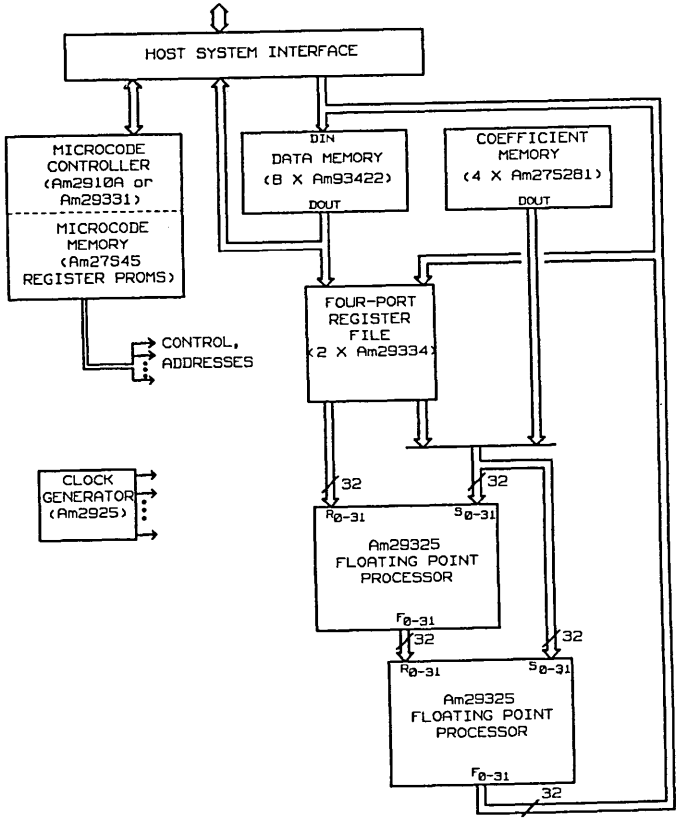


Fig. 4: System using two floating point processors in cascade.

Parallel configurations are also useful, and are easily implemented. In one such configuration, the Am29325 is used with other members of the Am29300 family to create a 32-bit floating-point/integer processor (fig. 5). In the system shown, the Am29332 ALU and the Am29323 32-by-32-bit parallel multiplier share three 32-bit buses with the Am29325; data can be passed from one processor to another through the Am29334 register file. Combining these parts produces a system that can perform high-speed floating point, integer, and logical operations. The user can further expand the system by adding 32-bit operators of his own devising to the three-bus architecture.

REFERENCES

1. **A Proposed Standard for Binary Floating-Point Arithmetic,** IEEE Floating-Point Working Group, draft 10.0, December 2, 1982.

2. **VAX Architecture Handbook,** Digital Equipment Corporation, 1981.

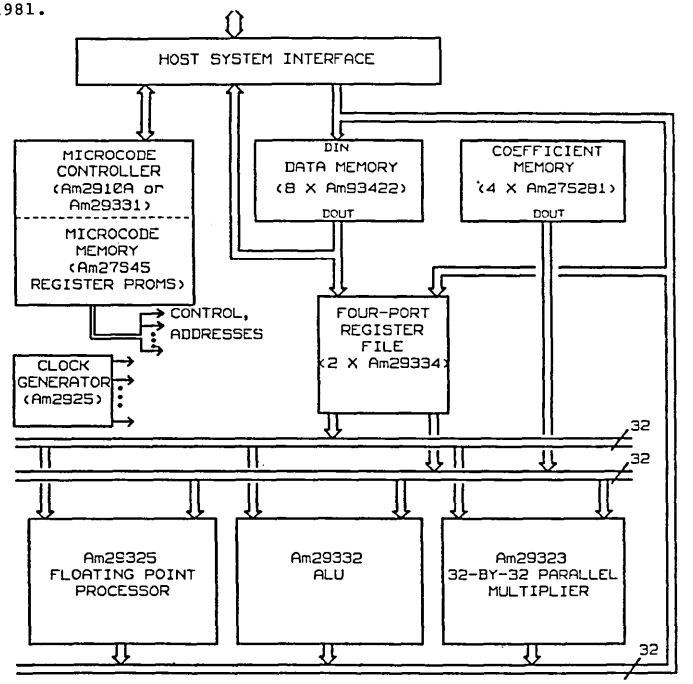
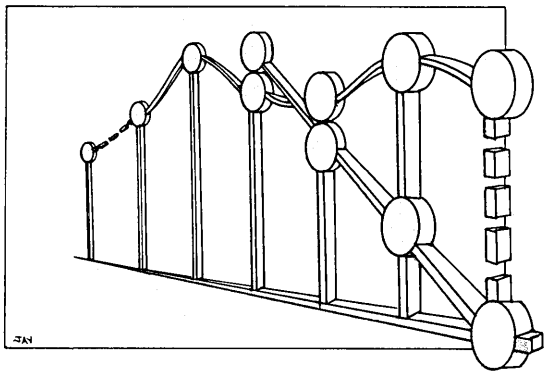


Fig. 5: Thirty-two-bit floating point/integer processor.

DIGITAL FILTER DESIGN MADE EASIER FOR FIRST-TIME USERS

Off-the-shelf components and simplified filter formulas ease entry into the world of digital filter design, and allow a quick evaluation of the cost-effectiveness of digital solutions.



by Kenn Lamb and Bob Perlman

Realtime digital filtering is becoming an attractive alternative in a growing number of analog-filtering applications. Today, specialized digital signal-processing part families, and a range of filter-design packages, make digital filters easier to implement. Nevertheless, getting into digital filter design for the first time is not easy. Some of the concepts are unfamiliar to novice filter designers and the tools generally available are aimed at the more experienced designers.

A "cookbook" approach, however, eases entry into digital filter design and provides a quick way to evaluate the cost-effectiveness of a digital solution. This approach uses off-the-shelf ingredients, such as the Am29500 family, and a simple "recipe" for a linear phase finite-impulse response (FIR) filter.

A digital filter performs the same function as an analog filter, but in a different "world." In the continuous world, a signal is monitored (or sampled) continuously, and filtering is described mathematically as a convolution operation.

In the discrete world, things can be done much more efficiently. The z-transform, in which powers of z can be equated to simple time delays, provides a formula that is the discrete equivalent of the convolution operation. In addition, sampling reduces the monitoring overhead to periodic snapshots of the signal. A digital filter simply implements the discrete convolution formula after an A-D converter has sampled the signal. Any arithmetic processor can perform the discrete convolution required for a digital filter, but the Am29500 family provides a processing package without the overhead of a microprocessor-based system, and can be optimized for digital-filtering applications.

Unlike the characteristics of an analog filter, a digital filter's characteristics are determined by arithmetic operations and coefficients, rather than by individual component values. This makes digital filtering inherently independent of component aging and environmental variables such as temperature. As a result, reliability is improved, and the filter response

Kenn Lamb is a former product planning engineer for Advanced Micro Devices (Sunnyvale, CA). He holds a BSC in electronics from Imperial College in London.

Bob Perlman is a product planning section manager at AMD. He holds an MSEE from Johns Hopkins University.

can be accurately reproduced. In addition, digital techniques permit useful characteristics that are not easily achieved in analog systems. Among these are zero passband insertion loss, very low frequency operation, and control over the stopband response.

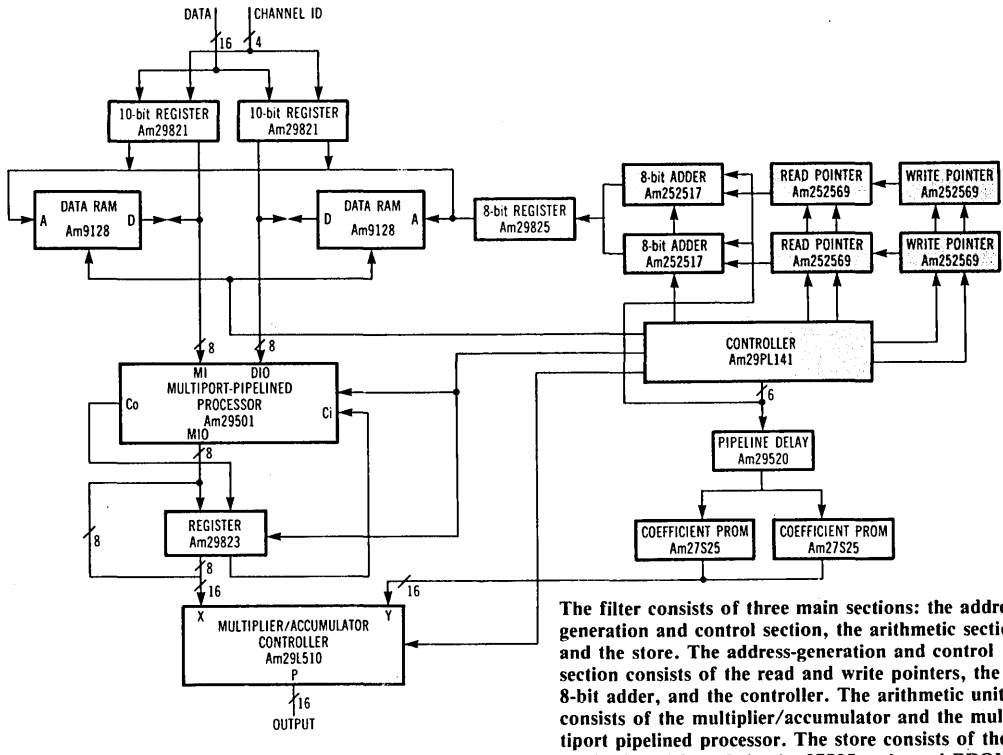
Using digital techniques, a designer can build linear-phase and all-pass filters that modify only the frequency or phase content of a signal. Linear-phase filters are used in multichannel environments where phase information is important, while all-pass filters are used typically for equalization. Cascading all-pass and linear-phase filters allows phase and frequency responses to be modified independently. In addition, filter coefficients, which are programmed by the designer, are easier to change than the components of an analog filter.

Because analog filters operate in the continuous world and can handle only one signal at a time, analog designs usually cascade a number of stages. As a result, there is a clear relationship between the physical size of an analog filter and its performance. The bandwidth of the active components within the filter determines the overall filter bandwidth, but spare bandwidth cannot be used for other channels.

Because the digital filter core is an arithmetic section that performs the calculations according to the discrete convolution formula, it may be used for one large, or many small, single-channel filters, or may be a stage in a very large, high-bandwidth filter.

Relationship between time and frequency

The match between design and actual response in an analog filter is determined by the tolerance with which analog components may be constructed; the digital equivalent is the resolution (number of bits) at which the digital coefficients are represented. A relatively lax analog specification places wide tolerance on the component values, which translates into a less accurate (fewer bits) resolution of the digital coefficients. For an analog filter, the dynamic range corresponds to the range between the noise floor and the point at which the signal starts to clip. Dynamic range in a digital filter is determined by the number of bits in the digital representation of the signal.



The filter consists of three main sections: the address-generation and control section, the arithmetic section, and the store. The address-generation and control section consists of the read and write pointers, the 8-bit adder, and the controller. The arithmetic unit consists of the multiplier/accumulator and the multiport pipelined processor. The store consists of the Am9128 RAMs and the Am27S25 registered PROMs.

What is convolution?

Two continuous functions x(t) and h(t) can be convolved by evaluating the equation:

$$y(t) = \int_{-\infty}^{\infty} x(\pi) \times h(t - \pi)\, d\pi$$

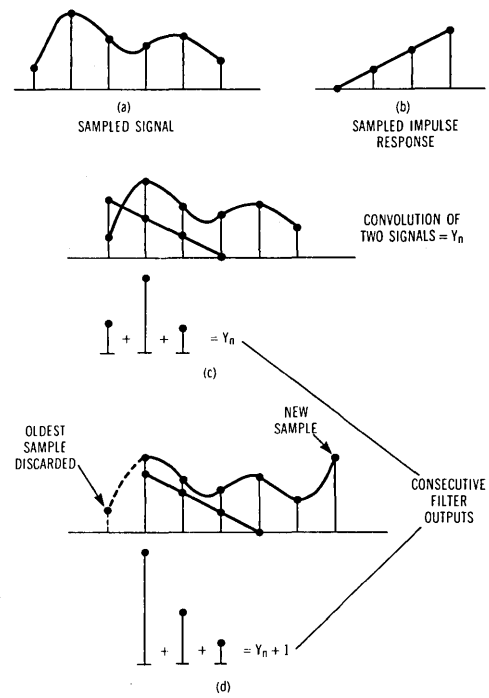
The discrete-time equivalent of this equation is:

$$y(n) = \sum_{k=0}^{N-1} x(k) \times h(n - k)$$

Although somewhat different in appearance, both forms of the convolution equation can be evaluated similarly. First, the function h is time-reversed, or flipped. Then, function h is shifted left or right, with the amount of shift indicated by time variables t or k. The resulting function is then either integrated (in the continuous case) or multiplied and summed point by point (in the discrete case) with function x.

One unfortunate aspect of convolution is that its worth is not readily apparent from the defining equations presented above. The real power of convolution is best appreciated by considering what happens in the frequency domain when two signals are convolved in the time domain. If the Fourier transforms of continuous signals x(t) and h(t) are X(f) and H(f), the convolution of x(t) and h(t) produces a signal whose frequency spectrum is equal to the product X(f)H(f). In other words, if one wishes to pass signal x(t) through a filter with transfer function H(f), one merely convolves x(t) with h(t). The same principle holds for discrete-time signals, but with the z transform taking the place of the Fourier transform.

The discrete convolution of a sampled signal (a) with a sampled-impulse response (b) may be achieved by the following process. First, flip the impulse response and place it over the signal (c). Sum the products of each impulse sample with its coincident signal sample. The total is the convolution result for the particular overlap, and corresponds to the first filter output. Now, slide the impulse response one sample interval to the right, so that it overlaps the next newest signal sample. Repeat the multiply/accumulate sequence. This result is the convolution for the new overlap, and corresponds to the second output from the filter. This process is repeated to calculate each new filter output. After each filter output, the oldest signal sample is no longer required, and may be discarded (d).



A signal to be filtered exists in the time domain of the familiar continuous world. The desired filter response for this signal is best represented in the frequency domain of the continuous world. These two domains are related through the continuous Fourier transform. The Fourier transform of a time-varying signal is its frequency spectrum, and the Fourier transform of the filter's frequency response is its impulse response. Viewed in the time domain, a filter's output is determined by the time-varying amplitude of the input signal and the filter's impulse response. In the frequency domain, on the other hand, the signal's frequency spectrum and the filter's frequency response determine the output. The frequency response and the impulse response say the same thing about a filter; the impulse response is simply the time-domain version of the filter's frequency response.

Just the continuous Fourier transform allows movement between the time and frequency domains of the continuous world, the z-transform allows movement directly from the continuous world into the sampled discrete world. Here the discrete time and frequency domains exist, linked by the discrete Fourier transform. Sampled versions of the input signal, the signal's frequency spectrum, and the filter's frequency and impulse response are used.

Design tradeoffs

To understand the design tradeoffs between filter size and performance, a designer must be familiar with two relationships between the time and fre-

quency domains: multiplication in one domain is equivalent to convolution in the other domain, and a signal cannot be duration-limited in both domains. A filter can perform a multiplication in the frequency domain in which the frequency spectrum of an incoming signal is multiplied by the frequency response of the filter, or it can perform a convolution operation in the time domain. For a digital filter, this requires the discrete convolution of a sampled version of the input signal with a sampled version of the filter's impulse response. The latter forms the coefficients of the filter.

The frequency response of the filter will almost certainly be duration-limited in the frequency domain, because it's unusual for a filter to pass all input frequencies. Obvious examples are low-pass and high-pass filters, where the aim is the elimination of great chunks of the input signal's frequency content. Because a signal cannot be duration-limited in both the time and frequency domains, a filter such as this will have an infinitely long impulse response. A simplistic approach to shortening this response is to truncate the impulse response to a convenient length.

Truncating the impulse response is equivalent to multiplying it by a function that has a value of one where the impulse response is to be preserved, and a value of zero where the impulse response is truncated. This truncating function is called rect (x) because its amplitude plot describes a rectangle. Since multiplication in the time domain is equivalent to a convolution in the frequency domain, the initial ideal filter response must be convolved with the Fourier transform of the rect (x) function. The Fourier transform is the well-known sinc function. The nature of this sinc function, however, smears the original ideal frequency response. And the more of the impulse response that is discarded, the worse the smearing effects of the sinc function.

A designer can obtain a duration-limited impulse response without wrecking the original filter's frequency response by multiplying the impulse response by a function with characteristics such that when the frequency response of this function is convolved with the desired filter's frequency response, it causes the minimum of smearing distortion. Window functions, such as the Hamming function do just this. A window function such as this has a narrow main lobe to maintain the selectivity of the filter, and small sidelobes to maintain the depth of the stopband.

Designing a low-pass filter

The design of a low-pass filter is particularly easy, because the impulse response is obtained from the Fourier transform of the ideal rectangular response, which then takes the form of a geometric series, with coefficients that can be expressed as the relatively simple sinc function sin (x)/x. The number (N) of coefficients (S_n) needed to implement the filter depends on the sampling rate (usually 2.5 times the maximum frequency in the signal), the cutoff frequency of the filter, and the frequency that defines the start of the stopband. These coefficients must be multiplied by appropriate window function coefficients (W_n) to yield windowed filter coefficients (C_n).

These decimal coefficient values must then be converted into 16-bit binary values, using the fractional two's complement numbering scheme. The coefficients are normally stored within registered PROMs, although RAM or EPROM storage offers advantages during development. Most A-D converters support the two's complement numbering scheme, ensuring that the representations of the data and coefficients are compatible.

Filter hardware is used to implement the discrete convolution operation given by the formula:

$$Y_k = \sum_{n=1}^{N} C_n \times X_{(k-n+1)}$$

where X_k and Y_k are the kth filter inputs and outputs, respectively. A filter implemented with this equation uses N data inputs to compute each filter output, and is therefore referred to as an N-tap or N-point filter.

One advantage of a linear phase filter is that the coefficients are symmetrical. This means the required number of multiplications and the size of the coefficient store can be halved by adding the two data points that will be multiplied by the same coefficient value. The modified formula is:

$$Y_k = \sum_{n=1}^{N/2} C_n \times [X_{(k-n+1)} + X_{(k-N+n)}]$$

The filter hardware used to implement this formula consists of three distinct sections: the address generation and control, the arithmetic section that performs the number crunching, and the store. The store holds a short time history of N samples for each of the channels to be filtered. These samples are held in a cyclic buffer with a length equal to the next integer power of two greater than the number of points in the filter. For each processed sample output from the filter, a new sample must be written in. This new sample overwrites the oldest sample within the cyclic buffer.

For each output from the filter, the N newest samples must be read into the arithmetic section so the discrete convolution operation can be performed. Two pointers are needed—one (which counts up) to indicate the write address for incoming samples, and

and one (which counts down) to indicate the read addresses for the discrete convolution operation.

The store will be required to read two data points for every cycle of the multiplier. The second data point may be found at an offset from the existing read pointer. An Am29PL141 fuse-programmable controller supplies this offset, and the modified address is calculated by an 8-bit adder formed from two Am25LS2517 ALUs.

For a six-point filter, the offsets (0, 3, 5, and 7) are applied, on alternate cycles, to the 8-bit adder by the Am29PL141, to permit generation of the second read address from the first read address.

For a six-point filter, all of the address calculations are performed modulo 8 (the next power of 2 greater than 6), which is done by masking the address applied to the store so that the store sees only the least significant three bits. After each new word is written into the store, the write pointer loads the read pointer with the correct cyclic buffer start address for the calculation of the next filter output.

Multiple channels may be accommodated by inhibiting the increments of the write pointer until all the channels have input a new sample. The channels are counted by the loop counter within the Am29PL141 fuse-programmable controller, and are separated in the store section by the high-order address bits latched with each new input. The channels may be presented in any order, but all channels must be processed at the same rate. When shifted one position so that the least significant bit is discarded, the values forming the offset sequence give

Determining the coefficients for a low-pass filter

The coefficients for a low-pass filter can be expressed in terms of the so-called sinc function. The number of coefficients required to implement the filter N is given by:

$$N = 4F_S/(F_{CS} - F_{CP})$$

F_S, the sampling rate of the A-D converter, is equal to 2.5 times F_O, the maximum frequency present in the input signal (set by the antialiasing filter). F_{CP} is the cutoff frequency of the filter (the end of the passband) and F_{CS} is the frequency that defines the start of the stopband (the end of the transition band).

This estimate for N (the number of coefficients) is usually conservative, so the value of N may be reduced safely by about 10 percent. This leeway allows an even value for N to be obtained. Having determined N, the coefficient values can be obtained by sampling the filter's impulse response. The required coefficients are given by the following sinc functions:

For $n = 1$ to N

$$S_n = \frac{\sin\,[2 \times \pi \times F_{CP} \times (n - (N+1)/2)/F_S]}{\pi \times (n - (N+1)/2)}$$

where the S_n values are the samples of the sinc function. These values must then be multiplied by the window function to yield the windowed filter coefficients (C_n):

$$C_n = S_n \times W_n$$

where W_n (the Hamming window coefficients) are given by:

$$W_n = 0.54 + 0.46 \times \cos\,[2 \times \pi \times (n - (N + 1)/2)/(N - 1)]$$

Choosing F_O equal to 5 kHz, F_{CP} equal to 3 kHz, and F_{CS} equal to 4.5 kHz gives an N equal to 33. Reducing this by 10 percent makes N equal to 30.

Inserting the values for F_S, F_{CP}, and N into the above equations gives:

$$S_n = [\sin (0.48 \times \pi \times (n - 15.5))]/(\pi \times (n - 15.5))$$
$$W_n = 0.54 + 0.46\ x \cos (2\ x\ \pi \times (n - 15.5)/29)$$

This gives the C_n coefficients listed in the table.

30-Point Low-Pass Filter Coefficients

Coefficient Index (n)	Impulse Samples (S_n)	Window Coefficients (W_n)	Filter Coefficients (C_n)
1	0.00275	0.080	0.00022
2	0.02353	0.090	0.00212
3	0.00000	0.122	0.00000
4	-0.02762	0.173	-0.00478
5	-0.00380	0.242	-0.00092
6	0.03291	0.324	0.01066
7	0.00931	0.417	0.00388
8	-0.04036	0.515	-0.02078
9	-0.01803	0.614	-0.01117
10	0.05236	0.710	0.03717
11	0.03407	0.798	0.02718
12	-0.07679	0.874	-0.06711
13	-0.07484	0.934	-0.06990
14	0.16350	0.976	0.15958
15	0.43580	0.997	0.43449
16	0.43580	0.997	0.43449
17	0.16350	0.976	0.15958
18	-0.07484	0.934	-0.06990
19	-0.07679	0.874	-0.06711
20	0.03407	0.798	0.02718
21	0.05236	0.710	0.03717
22	-0.01803	0.614	-0.01117
23	-0.04036	0.515	-0.02078
24	0.00931	0.417	0.00388
25	0.03291	0.324	0.01066
26	-0.00380	0.242	-0.00092
27	-0.02762	0.173	-0.00478
28	0.00000	0.122	0.00000
29	0.02353	0.090	0.00212
30	0.00275	0.080	0.00022

Addressing and Operation Sequence for a Six-Point Filter

Cycle Number	Write Count	Read Count	Offset	Read Address	Write Address	Add	Mult	Output
1	0	load	-	-	0	***	******	--
2	0	0	0	0	-	***	******	--
3	0	0	3	3	-	*---	******	--
4	0	7	0	7	-	0+3	******	--
5	0	7	5	4	-	0+3	------	**
6	0	6	0	6	-	7+4	0+3×C1	--
7	0	6	7	5	-	7+4	0+3×C1	--
8	1	load	-	-	1	6+5	7+4×2	--
9	1	1	0	1	-	6+5	7+4×C2	--
10	1	1	3	4	-	---	6+5×C3	--
11	1	0	0	0	-	1+4	6+5×C3	--
12	1	0	5	5	-	1+4	------	Y1
13	1	7	0	7	-	0+5	1+4×C1	--
14	1	7	7	6	-	0+5	1+4×C1	--
15	2	load	-	-	2	7+6	0+5×C2	--
16	2	2	0	2	-	7+6	0+5×C2	--
17	2	2	3	5	-	---	7+6×C3	--
18	2	1	0	1	-	2+5	7+6×C3	--
19	2	1	5	6	-	2+5	------	Y2
20	2	0	0	0	-	1+6	2+5×C1	--
21	2	0	7	7	-	1+6	2+5×C1	--
22	3	load	-	-	3	0+7	1+6×C2	--
23	3	3	0	3	-	0+7	1+6×C2	--
24	3	3	3	6	-	---	0+7×C3	--
25	3	2	0	2	-	3+6	0+7×C3	--
26	3	2	5	7	-	3+6	------	Y3
..	.	.	.	.	.	...	.....	
..	.	.	.	.	.	...	.....	

The write count is used to load the read count every seventh cycle. Read addresses are obtained by adding the read count to the offset modulo 8. Two cycles are required for each add operation, with the least significant halves being added in the first cycle, the most significant in the second. The Add column indicates the addresses of the data samples that are added together on each cycle. The multiplier/accumulator operates at half the speed of the adder, and therefore requires two cycles for each operation. The Mult column shows the results of prior add operations being multiplied by the filter coefficients. The filter outputs are shown in the output column. The filter coefficients are obtained by multiplying the impulse samples by the associated window coefficient.

the new sequence (0, 1, 2, 3) required to address the coefficients. An Am29520 must implement a two-cycle delay to ensure that the address is made available to the registered PROMs when needed.

The arithmetic section consists of a single 8-bit Am29501 and a 16-bit Am29L510. The 501 performs a 16-bit addition for every cycle of the 510, and operates at twice the clock rate of the 510. An additional 9-bit Am29823 register latches the least significant eight bits and the carry out of the add operation as it is performed by the 501. The 510 performs the multiply/accumulate operations required by the discrete convolution process; the guard bits within the accumulator accommodate word growth.

There are two store requirements within the filter—the data and the coefficients. The data resides in two 2-kword, 8-bit wide Am9128 RAMs that cycle at the same rate as the 501. The coefficients reside in two 8-bit wide Am27S25 registered PROMs that cycle at the rate of the 510.

This filter structure accommodates up to 16 channels, each filtered by one stage of a linear phase FIR filter that has up to 128 points. Each point requires 50 ns to process, yielding an effective sampling rate of 40 kHz for each of the 16 channels using the low-pass filter described in the panel "Determining the coefficients for a low-pass filter." There is sufficient capacity within the PROMs for 32 different sets of filter coefficients, allowing a different filter to process each channel. Outputs may be returned as inputs to a different channel, allowing steeper rolloffs by cascading filters, or more complicated responses by cascading low- and high-pass filters.

This filter structure can be configured to yield a filter bank capable of resolving up to 32 spectral components from an input signal sampled at 20 kHz (using filters of similar complexity to the low-pass one in the panel). This type of filter bank provides a stable platform for automatic speech-recognition algorithms. If 20 channels are dedicated to resolving

spectral components, 37 percent of the available processing bandwidth remains for squaring and integrating the outputs from each filter. A low-pass filter performs integration, and threshold detection may be performed with the Am29501.

The hardware described can perform all of the processing required to generate the "intensity" spectrum of the input signal, which is the major processing requirement of a speech-recognition system.

One of the advantages of a filter bank implementation is that the signal spectrum may be broken into completely arbitrary divisions (20 in this example). An FFT implementation, on the other hand, would require a linear scale, while decimating filter techniques would require divisions linked by integer ratios. A useful scale not usually available has a logarithmic spacing of center frequencies and bandwidths. This logarithmic spacing may be specified at the design stage. The band of interest ranges from 400 Hz to 8 kHz, which may be satisfactorily sampled at a rate of 20 kHz. Each of these center frequencies must be entered into the band-pass filter-impulse response equation, with the associated bandwidth, in order to generate the filter coefficients.

The design of a band-pass filter using windowed-impulse techniques follows the same steps as the low-pass filter design, with two differences: the number of coefficients roughly doubles because there are two distinct stopbands within the band-pass filter response, and the coefficients are computed using a modified formula.

Because a single input channel will pass through 20 different band-pass filters, the same allocation of 20 channels is made here as in the low-pass filter, but there is a slight difference in the control sequence. Previously, a new sample was written into the cyclic buffer each time a processed point came out of the filter. But, for this application, a new sample is written into the cyclic buffer for every 20 output points from the filter. This means modifying the controller microcode so that the write pointer increments are inhibited until 20 outputs have been calculated.

Determining the coefficients for a band-pass filter

The coefficients for a band-pass filter are derived from the following variation of the low-pass filter-impulse response formula:

$$S_n = \frac{\sin\left[\pi \times B \times (n-(N+1)/2)/F_s\right] \times 2 \times \cos\left[2 \times \pi \times F_c \times (n-N+1)/2/Fs\right]}{\pi \times (n-(N+1)/2)}$$

where B is the 3dB bandwidth, F_c is the center frequency, F_s is the sampling frequency, N is the number of coefficients, and n is the coefficient index. The same Hamming window formula used for the low-pass filter may be used here.

Specifying each filter to have the same number of coefficients simplifies the overall control of the filter bank. Up to 32 coefficients may be used for each filter without exceeding the available processing bandwidth; a reasonable design specification would set N equal to 32, with F_s at 20 kHz. Inserting these values in the impulse response formula yields:

$$S_n = \frac{\sin\left[\pi \times B \times (n-16.5)/20000\right] \times 2 \times \cos\left[\pi \times F_c \times (n-16.5)/10000\right]}{\pi \times (n-16.5)}$$

Substituting the 20 pairs of values for F_c and B into this formula will, after multiplying by the appropriate window term, yield the required filter coefficients. These coefficients must then be converted into a fractional two's complement format, and programmed into the PROMs.

Am29501A

Multi-Port Pipelined Processor (Byte-Slice™)

ADVANCED INFORMATION

DISTINCTIVE CHARACTERISTICS

- Expandable Byte-Slice™ Register-ALU
  - Speed improved version of the Am29501
- Eight instruction ALU
  - Four arithmetic operations
  - Four logic operations
  - Force/Inhibit carry modes
  - Flexible expansion – has carry and $\overline{P}$/ $\overline{G}$
- Three I/O ports for maximum system interconnect flexibility
- Ten internal data paths
  - Highly parallel architectures
  - Multiple simultaneous data manipulations
- Pipelining register file has six 8-bit registers
  - Multilevel pipelining
  - Multiple register-to-register moves
- Completely microprogrammable
  - No instruction encoding
  - All operation combinations available

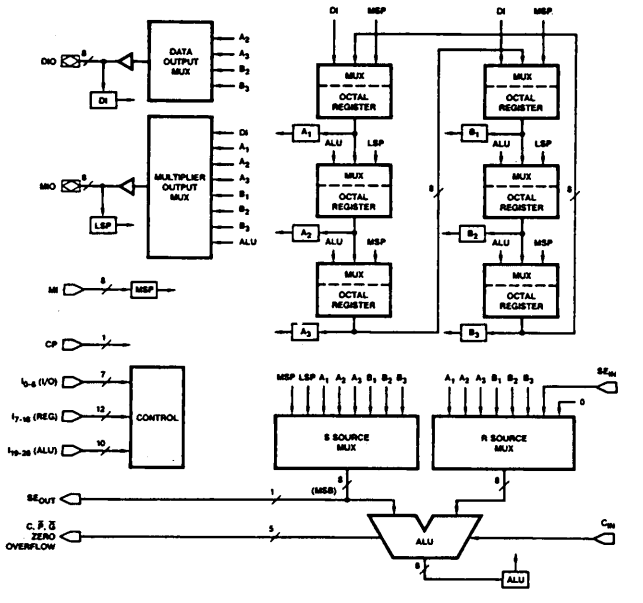
GENERAL DESCRIPTION

The Am29501A is an expandable Byte-Slice™ register-ALU designed to bring maximum speed to array processor and digital signal processor systems. It provides a flexible processor building block for implementing highly pipelined, highly parallel architectures where speed is achieved by a combination of optimized integrated circuit technology (IMOX™ process and internal ECL circuitry) and customized system architecture. I/O port flexibility and multiple concurrent data moves make it possible to construct processors capable of very high throughput. Parallel processors are especially efficient for array/vector operations or signal processing algorithms requiring complex number arithmetic (e.g. FFT, convolution, correlation, etc.).

The Am29501A's Pipeline Register File provides data storage and pipelining flexibility. Any combination of register instructions, ALU instructions, and I/O instructions can be microprogrammed to occur in the same cycle. This allows overlap of external multiplication, ALU operations, and memory I/O.

Three I/O ports support a wide variety of parallel, pipelined architectures by providing separate I/O ports for the multiplier and the memory data bus. Either of two bidirectional I/O ports, DIO and MIO, can interface to the data bus or multiplier Y-input port. A separate MI port connects to the multiplier output port.

BLOCK DIAGRAM



BD003060

Am29509/L509

12 x 12 Multiplier Accumulator

ADVANCED INFORMATION

DISTINCTIVE CHARACTERISTICS

- Uses two's complement or unsigned inputs and outputs
- Round control
- 27-bit product accumulation result
  - 24-bit product
  - 3-bit extended product
- Output register preload
- Three-state output control
- IMOX™ processing
  - ECL internal circuitry for speed
  - TTL I/O

IMOX is a trademark of Advanced Micro Devices, Inc.

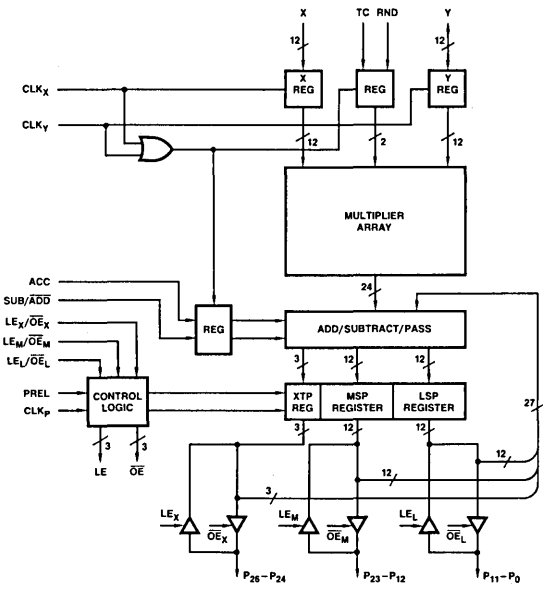
GENERAL DESCRIPTION

The Am29509 is a high-speed 12-bit x 12-bit multiplier/accumulator (MAC). The X and Y input registers accept 12-bit inputs in two's complement or unsigned magnitude format. A third register stores the Two's Complement (TC) and Round (RND) control bits. This register is clocked whenever the X or Y input registers are clocked.

The 27-bit accumulator/output register contains the full 24-bit multiplier output which is sign extended or zero-filled based on the TC control bit. The accumulator can also be preloaded from an external source through the bidirectional P-port. The operation of the accumulator is controlled by the signals ACC (Accumulator), SUB/$\overline{\text{ADD}}$ (Subtraction/Addition), and PREL (Preload). Each of the input registers and output register has independent clocks.

The Am29L509 is a low-power version of the Am29509. "L" devices consume half the power that the standard parts require, yet maintain more than one-half the speed.

BLOCK DIAGRAM



04986C-1

RELATED PRODUCTS

Part No.	Description
Am29520/21	Multilevel pipeline registers
Am29526/527	High speed Sine function generator
Am29528/529	High speed Cosine function generator
Am29540	Programmable FFT address sequencer

Am29510/L510

16 x 16 Multiplier Accumulator

PRELIMINARY

DISTINCTIVE CHARACTERISTICS

- Uses two's complement or unsigned inputs and outputs
- Round control
- Output register preload
- 35-bit product accumulator result
  - 32-bit product
  - 3-bit extended product
- IMOX™ processing
  - ECL internal circuitry for speed
  - TTL I/O, single 5V supply
- Fast
  - High speed version multiply accumulate time 80ns
  - Low power version multiply accumulate time 110ns

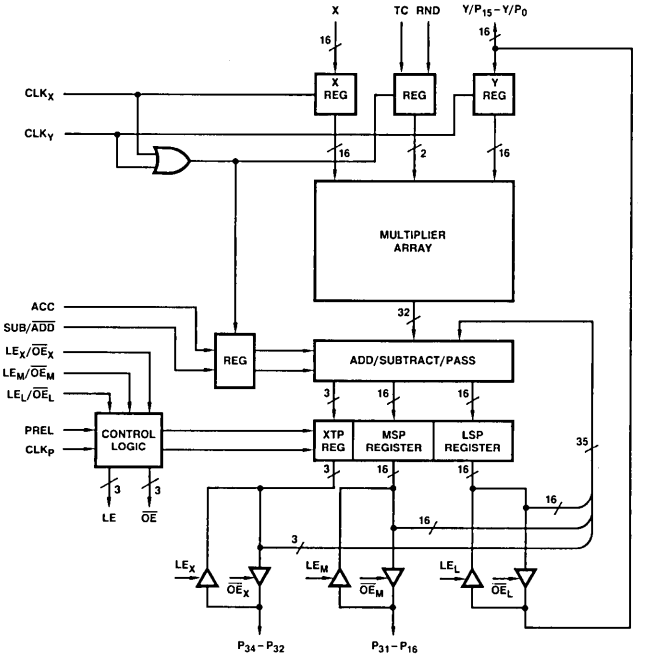
GENERAL DESCRIPTION

The Am29510 is a high-speed 16 x 16-bit multiplier/accumulator (MAC). The X and Y input registers accept 16-bit inputs in two's complement or unsigned magnitude format. A third register stores the Two's Complement (TC) and Round (RND) control bits. This register is clocked whenever the X or Y input registers are clocked.

The 35-bit accumulator/output register contains the full 32-bit multiplier output which is sign extended or zero-filled based on the TC control bit. The accumulator can also be preloaded from an external source through the bidirectional P port. The operation of the accumulator is controlled by the signals ACC (Accumulator), SUB/$\overline{\text{ADD}}$ (Subtraction/Addition), and PREL (Preload). Each of the input registers and output register has independent clocks.

The Am29L510 is a low-power version of the Am29510. The Am29L510 consumes only one-half the power of its standard power counterpart while maintaining nearly two-thirds the speed.

BLOCK DIAGRAM



RELATED PRODUCTS

Part No.	Description
Am29526/527	High-speed Sine function generator
Am29528/529	High-speed Cosine function generator
Am29540	Programmable FFT address sequencer
Am29520/21	Multilevel pipeline registers

Am29516/17 Family

16 x 16-Bit Parallel Multipliers

DISTINCTIVE CHARACTERISTICS

- High speed 16 x 16 parallel multiplier
- Two's complement, unsigned or mixed operands
- Full product multiplexed at output
- Am29516 pin and functionally compatible with TRW MPY-16HJ – Am29517 optimized for microprogramming, single clock with register enables
- Improved speed: 38ns clock multiply (A devices)
- Reduced power dissipation: 2W (L devices)
- TTL I/O-single +5V supply

GENERAL DESCRIPTION

The Am29516 and Am29517 are high-speed parallel 16 x 16-bit multipliers utilizing internal ECL logic to generate a 32-bit product. Two 17-bit input registers are provided for the X and Y operands and their associated mode controls X_M and Y_M. These mode controls are used to specify each operand as either two's complement or unsigned numbers. When one operand is two's complement and the other is unsigned, the result is two's complement.

At the output of the multiplier array, a format adjust control (FA) allows the user to select either a full 32-bit product or a left-shifted 31-bit product suitable for two's complement only.

Two 16-bit output registers are provided to hold the most and least significant halves of the product (MSP and LSP) as defined by FA. For asynchronous output, these registers may be made transparent by taking the feed through control (FT) High. A round control (RND) allows the rounding of the MSP; this control is registered, and is entered whenever either input register is clocked.

The two halves of the product may be routed to a 16-bit 3-state output port (P) via a multiplexer, and in addition, the LSP is connected to the Y-input port through a separate three-state buffer.

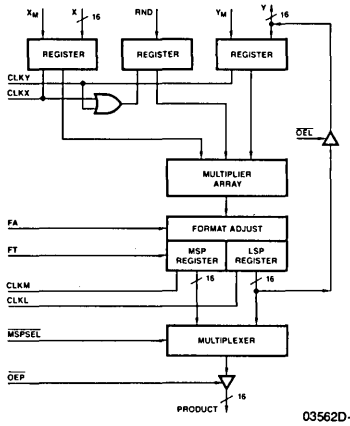
The Am29516 X, Y, MSP and LSP registers have independent clocks (CLKX, CLKY, CLKM, CLKL). The output multiplexer control ($\overline{\text{MSPSEL}}$) uses a pin which is a supply ground in the TRW MPY16HJ. When this control is LOW, the function is that of the MPY16HJ, thus allowing full compatibility.

The Am29517 differs in that it has a single clock input (CLK) and three register enables ($\overline{\text{ENX}}$, $\overline{\text{ENY}}$, $\overline{\text{ENP}}$) for the two input registers and the entire product, respectively. This facilitates the use of the part in microprogrammed systems. In both parts data is entered into the registers on the positive edge of the clock.

The Am29516A and Am29517A are higher speed versions of the Am29516 and Am29517, respectively, offering greater than 40% speed improvement while the Am29L516 and Am29L517 low-power versions consume only one-half the power of their standard power counterparts.

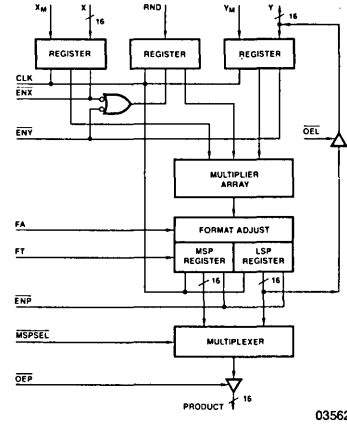
BLOCK DIAGRAMS

Am29516/29516A/29L516



03562D-1

Am29517/29517A/29L517



03562D-2

RELATED PRODUCTS

Part No.	Description
Am29501	Multiport pipelined processor
Am29526/27	Sine function generator
Am29528/29	Cosine function generator
Am29520/21	Pipeline register
Am29540	Address generator

Order # 03562D

Am29520 • Am29521

Multilevel Pipeline Registers

DISTINCTIVE CHARACTERISTICS

- Four 8-bit high speed registers
- Dual two-level or single four-level push-only stack operation
- All registers available at multiplexed output
- Hold, transfer and load instructions
- Provides temporary address or data storage
- 24-pin 0.3" package

GENERAL DESCRIPTION

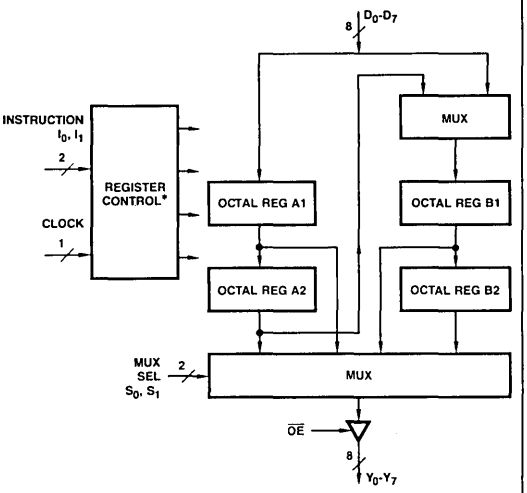
The Am29520 and Am29521 each contain four 8-bit positive edge-triggered registers. These may be operated as a dual 2-level pipeline or as a single 4-level pipeline. A single 8-bit input is provided and all four registers are available at the 8-bit, 3-state output.

The Am29520 and Am29521 differ only in the way data is loaded into and between the registers in dual 2-level operation. This difference is illustrated in Figure 1. In the Am29520 when data is entered into the first level (I=2 or I=1) the existing data in the first level is moved to the second level. In the Am29521 these instructions simply cause the data in the first level to be overwritten. Transfer of data to the second level is achieved using the 4-level shift instruction (I=0). This transfer also causes the first level to change. In either part I=3 is a NO-OP.

RELATED PRODUCTS

Part No.	Description
Am29540	FFT Address Sequencer
Am29116	16-bit Bipolar Microprocessor
Am2925	System Clock Generator and Driver
Am29517	16 x 16-bit High Speed Multiplier
Am29510	16 x 16-bit Multiply Accumulator
Am6108	8-bit Microprocessor Compatible A/D Converter
Am9128-70	2K x 8 Static RAM
Am21L47-55	4K x 1 Static RAM

LOGIC DIAGRAM



03569A-1

*Multilevel Pipeline Register

Am29526 • Am29527
Am29528 • Am29529

High Speed Sine, Cosine Generators

DISTINCTIVE CHARACTERISTICS

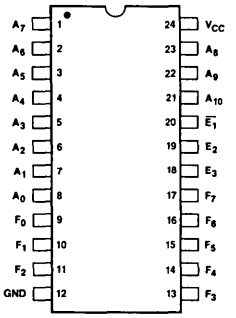
- Provides values for sine/cosine functions in $\pi/2048$ increments
- Outputs are 16-bit two's complement fractions
- Fast generation time of 50ns max Com'l
- S/LS compatible
- Three-state outputs
- IMOX™ processing

RELATED PRODUCTS

Part No.	Description
Am29516/17	16 x 16-Bit High Speed Multipliers
Am29510	16 x 16-Bit Multiply Accumulator
Am29540	FFT Address Sequencer
Am29825	High Performance 8-Bit Register

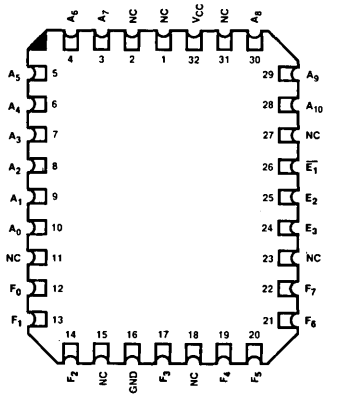
CONNECTION DIAGRAMS – Top Views

DIP



Chip-Pak™

ABL-006



ABL-007

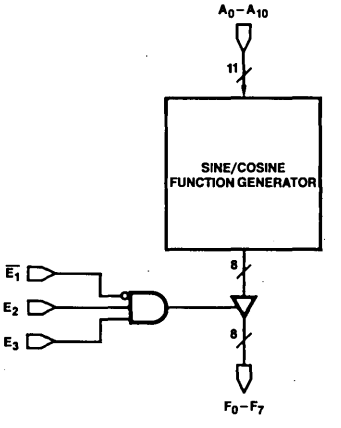
FUNCTIONAL DESCRIPTION

The Am29526/27 and Am29528/29 provide high speed generation of sine and cosine functions over the range $0 \leq \theta < \pi$ in increments of $\pi/2048$. θ is determined by an 11-bit input word. Each device provides an 8-bit output and two are used to give the full 16-bit value. The Am29526 and Am29527 generate the MS and LS bytes respectively for the sine function. Similarly, the Am29528 and Am29529 generate the cosine functions.

The outputs are fractional two's complement numbers with the radix point located immediately to the right of the sign bit (in between the bits weighted -2^0 and 2^{-1}). As this format does not allow for the representation of $+1$ the functions generated are $-\sin\theta$ and $-\cos\theta$. In this way the output values are restricted to the range $-1 \leq f(\theta) < +1$ which is representable. The outputs are three-state with one active Low enable and two active High enable.

While providing general purpose sine and cosine function capability, the Am29526/27/28/29 satisfy the requirements of the Am29540 FFT Address Sequencer.

BLOCK DIAGRAM



ABL-008

IMOX and Chip-Pak are trademarks of Advanced Micro Devices, Inc.

Refer to Page 13-1 for Essential Information on Military Devices

Am29540

Programmable FFT Address Sequencer

DISTINCTIVE CHARACTERISTICS

- Generates data and coefficient addresses
- Programmable transform length 2 to 65,536 points
- Radix-2 or Radix-4
- Decimation in frequency (DIF) or decimation in time (DIT) FFT algorithms supported
- In-place or non-in-place transformation
- 40-pin DIP package
- 5 volt single supply

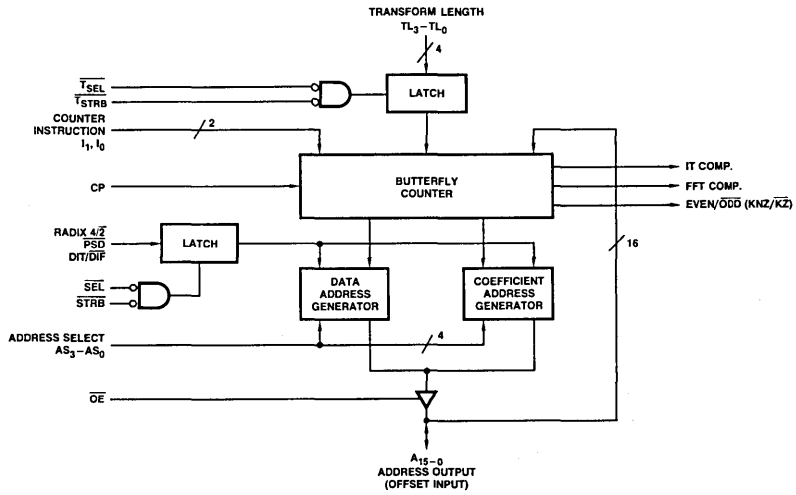
GENERAL DESCRIPTION

The Am29540 Fast Fourier Transform Address Sequencer generates all the data (RAM) and coefficient (ROM) addresses necessary to perform the repetitive butterfly operations of the FFT. Decimation in time and decimation in frequency algorithms are supported (control DIT/$\overline{\text{DIF}}$) in radix-2 or radix-4 (RADIX 4/2). A radix-2 real valued input (RVI) transform is also supported. For radix-2 operation the transform length is programmable in powers of 2 from 2 to 65,536 points. In radix-4 the range is 4 to 65,536 in powers of 4.

Address sequences can be selected to be compatible with data which may or may not have been pre-scrambled ("bit-reversed"). If the data has been pre-scrambled the control $\overline{\text{PSD}}$ must be LOW to select the correct sequence. If the data is not pre-scrambled and an in-place transform is performed, the output data will necessarily be in bit-reversed order. If this is not desirable, alternate addresses are available for a non-in-place, non-bit-reversing algorithm.

The butterfly counter operates on the positive clock edge and responds to four instructions. COUNT causes the counter to increment to the next butterfly. RESET causes the counter to initialize for the specified transform length. RESET/LOAD causes the counter to initialize and a data address offset to be loaded into the part via the bi-directional 3-state ADDRESS port. This offset is effectively OR-ed onto the higher significance bits of the address which are unused for the selected transform length. A HOLD instruction is also provided. Three status lines are provided. EVEN/$\overline{\text{ODD}}$ (KNZ/$\overline{\text{KZ}}$) controls the alternation of read and write memories for non-in-place transforms and determines the butterfly structure in the RVI transform. The flag has the function KNZ/$\overline{\text{KZ}}$ when RVI data addresses are selected (AS = 12 to 15). Iteration complete (IT COMP) flags the bottom of a "column" of butterflies and is used in conjunction with block floating point schemes. FFT COMP identifies the last butterfly of the transform.

LOGIC DIAGRAM



FFT Address Sequencer

RELATED PRODUCTS

Am29501 – Mulitport pipelined processor (Byte-Slice™)
Am29516/17 – 16 x 16 parallel multiplier
Am29520/21 – Multilevel pipeline register
Am29526/27/28/29 – High-speed, sine/cosine generators
Am29825 – High-performance, 8-bit register

Order # 03567D

Am29323

32-Bit Parallel Multiplier

ADVANCED INFORMATION

DISTINCTIVE CHARACTERISTICS

- **32-Bit Three-Bus Architecture**
  - The device has two 32-bit input ports and one 32-bit output port with maximum multiply time of 80ns
- **Single Clock with Register Enables**
  - The Am29323 is controlled by one clock with individual register enables
- **Supports Multiprecision Multiplication**
  - The device has dual 32-bit registers on each data input port to perform multiprecision multiplication
- **Registers can be made transparent**
  - Input and output registers can be made transparent independently to eliminate unwanted pipeline delay
- **Supports Two's Complement, Unsigned or Mixed Numbers**
- **Data Integrity Through Master-Slave Mode and Parity Check/Generate**
  - Parity check/generate catches inter-device connection errors and master/slave mode provides complete function check

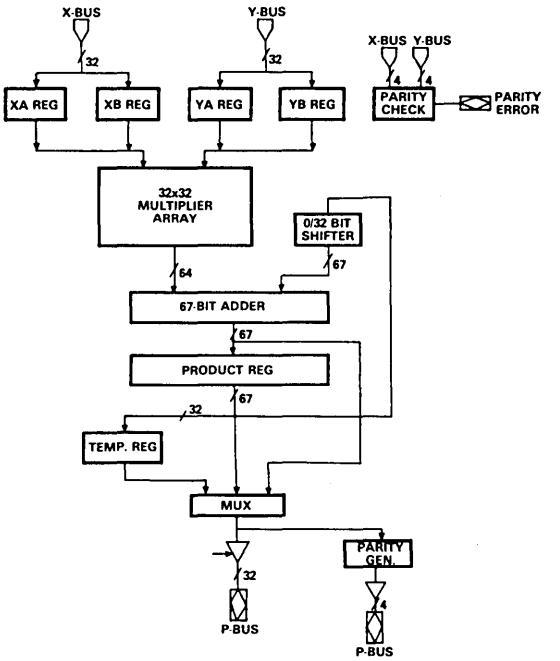
GENERAL DESCRIPTION

The Am29323 is a high-speed 32 x 32-Bit Parallel Multiplier with 67-Bit Accumulator. The part is designed to maximize system level performance by providing a 32-bit three bus architecture and a single clock with register enables.

The Am29323 further enhances the system throughput by providing individual register feedthrough controls, byte parity checking on both input ports and generation on the output port, and dual input registers on each data input bus to support multiprecision multiplication. The Am29323 can manage a wide variety of data types, including two's complement, unsigned, or mixed mode input formats. A 64 x 64-bit multiplication can be performed in seven clock cycles, including input and output. Additional features provided are a format adjust control allowing for standard output or left shifted output suitable for fractional two's complement arithmetic, rounding, and master/slave operation.

The Am29323 is designed with the IMOX* process, which allows internal ECL circuits with TTL-compatible I/O. The device is housed in a 168-lead pin-grid-array package.

SIMPLIFIED BLOCK DIAGRAM



BD005250

*IMOX is a trademark of Advanced Micro Devices, Inc.

This document contains information on a product under development at Advanced Micro Devices, Inc. The information is intended to

Order # 05763B

Am29325

32-Bit Floating Point Processor

PRELIMINARY

DISTINCTIVE CHARACTERISTICS

- Single VLSI device performs high-speed floating-point arithmetic
  - Floating-point addition, subtraction and multiplication in a single clock cycle
  - Internal architecture supports sum-of-products, Newton-Raphson division
- 32-bit, 3-bus flow-through architecture
  - Programmable I/O allows interface to 32- and 16-bit systems
- IEEE and DEC formats
  - Performs conversions between formats
  - Performs integer ⟷ floating point conversions
- Six flags indicate operation status
- Register enables eliminate clock skew
- Input and output registers can be made transparent independently

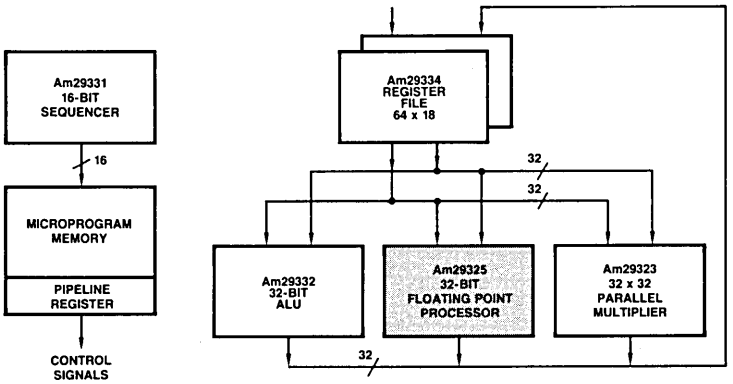
GENERAL DESCRIPTION

The Am29325 is a high-speed floating-point processor unit. It performs 32-bit single-precision floating-point addition, subtraction, and multiplication operations in a single LSI integrated circuit, using the format specified by the proposed IEEE floating-point standard P754. The DEC single-precision floating-point format is also supported. Operations for conversion between 32-bit integer format and floating-point format are available, as are operations for converting between the IEEE and DEC floating-point formats. Any operation can be performed in a single clock cycle. Six flags – invalid operation, inexact result, zero, not-a-number, overflow, and underflow – monitor the status of operations.

The Am29325 has a 3-bus, 32-bit architecture, with two input buses and one output bus. This configuration provides high I/O bandwidth, allows access to all buses and affords a high degree of flexibility when connecting this device in a system. All buses are registered, with each register having a clock enable. Input and output registers may be made transparent independently. Two other I/O configurations, a 32-bit, 2-bus architecture and a 16-bit, 3-bus architecture, are user-selectable, easing interface with a wide variety of systems. Thirty-two-bit internal feedforward data paths support accumulation operations, including sum-of-products and Newton-Raphson division.

Fabricated with the high-speed IMOX™ bipolar process, the Am29325 is powered by a single 5-volt supply. The device is housed in a 144-pin pin-grid-array package.

Am29300 FAMILY HIGH PERFORMANCE SYSTEM BLOCK DIAGRAM



RELATED PRODUCTS

- Am29323 – 32 x 32 Parallel Multiplier
- Am29332 – 32-Bit ALU
- Am29331 – 16-Bit Sequencer
- Am29334 – 64 x 18 Four-Port Dual-Access Register File

Order # 05621B

Am29331

16-Bit Microprogram Sequencer

ADVANCED INFORMATION

DISTINCTIVE CHARACTERISTICS

- **16-Bits Address Up to 64K Words**

  Supports 80–90ns microcycle time for a 32-bit high performance system when used with the other members of the Am29300 Family.

- **Real Time Interrupt Support**

  Micro-TRAP and Interrupts are handled transparently at any microinstruction boundary.

- **Built-In Conditional Test Logic**

  Generates inequality evaluation branch conditions from four ALU status bits. Has eight external tests plus a polarity input.

- **Break-Point Logic**

  Built-in address comparator allows break-points in the microcode for debugging and statistics collection.

- **Master/Slave Error Checking**

  Two sequencers can operate in parallel as a Master and a Slave. The Slave generates a fault flag for unequal results.

- **33-Level Stack**

  Provides support for interrupts, loops and subroutine nesting. It can be accessed through the D-bus to support diagnostics.

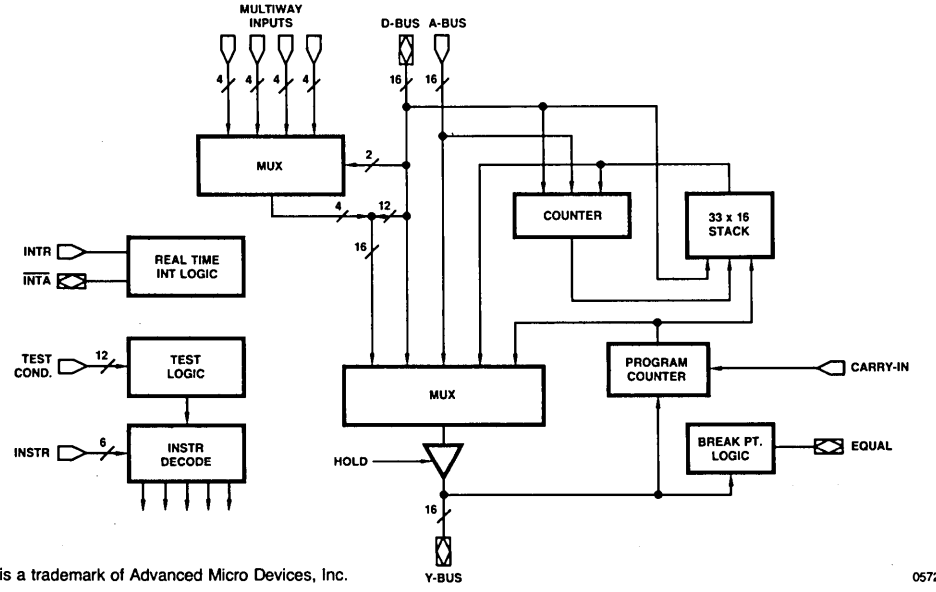
GENERAL DESCRIPTION

The Am29331 is a 16-bit wide high-speed single chip sequencer designed to control the execution sequence of microinstruction stored in the microprogram memory. The instruction set is designed to resemble high-level language constructs, thereby bringing high-level language programming to the micro level.

The Am29331 is interruptible at any microinstruction boundary to support real-time interrupts. Interrupts are handled transparently to the microprogrammer as an unexpected procedure call. Traps are also handled transparently at any microinstruction boundary. This feature allows re-execution of a prior microinstruction. Two separate buses are provided to bring a branch address directly into the chip from two sources to avoid slow turn-on and turn-off times for different sources connected to the data input bus. Four sets of multiway inputs are also provided to avoid slow turn-on and turn-off times for different branch address sources. This feature allows implementation of table look-up or use of external conditions as part of a branch address. The thirty-three deep stack provides the ability to support interrupts, loops and subroutine nesting. The stack can be read through the D-bus to support diagnostics or to implement multi-tasking at the micro-architecture level. The master/slave mode provides a complete function check capability for the device.

The Am29331 is designed with the IMOX™ process which allows internal ECL circuits with TTL-compatible I/O. It is housed in a 120-lead pin-grid-array package.

SIMPLIFIED BLOCK DIAGRAM



IMOX is a trademark of Advanced Micro Devices, Inc.

05729B-1

Am29332

32-Bit Arithmetic Logic Unit

ADVANCED INFORMATION

DISTINCTIVE CHARACTERISTICS

- **Single Chip, 32-Bit ALU**
  Supports 80–90ns microcycle time for the 32-bit data path. It is a combinatorial ALU with equal cycle time for all instructions.
- **Flow-through Architecture**
  A combinatorial ALU with two input data ports and one output data port allows implementation of either parallel or pipelined architectures.
- **64-Bit In, 32-Bit Out Funnel Shifter**
  This unique functional block allows n-bit shift-up, shift-down, 32-bit barrel shift or 32-bit field extract.
- **Supports All Data Types**
  It supports one-, two-, three- and four-byte data for all operations and variable-length fields for logical operations.
- **Multiply and Divide Support**
  Built-in hardware to support two-bit-at-a-time modified Booth's algorithm and one-bit-at-a-time division algorithm.
- **Extensive Error Checking**
  Parity check and generate provides data transmission check and master/slave mode provides complete function checking.

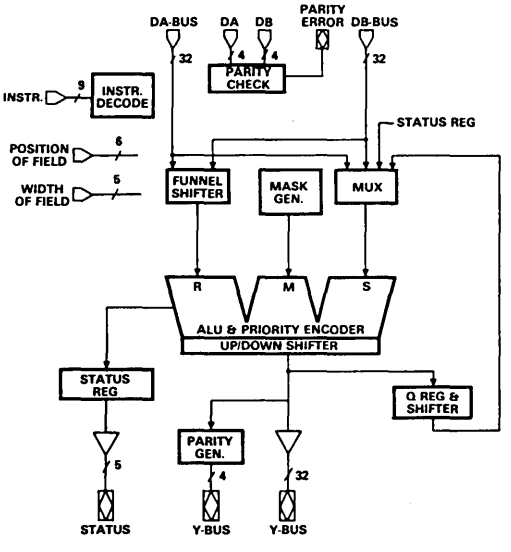
GENERAL DESCRIPTION

The Am29332 is a 32-bit wide non-cascadable Arithmetic Logic Unit (ALU) with integration of functions that normally don't cascade, such as barrel shifters, priority encoders and mask generators. Two input data ports and one output data port provide flow-through architecture and allow the designer to implement his/her architecture with any degree of pipelining and no built-in penalties for branching. Also, the simplicity of a three-bus ALU allows easy implementation of parallel or reconfigurable architectures. The register file is off-chip to allow unlimited expansion and regular addressability.

The Am29332 supports one-, two-, three- and four-byte data for arithmetic and logic operations. It also supports multiprecision arithmetic and shift operations. For logical operations, it can support variable-length fields up to 32 bits. When fewer than four bytes are selected, unselected bits are passed to the destination without modification. The device also supports two-bit-at-a-time modified Booth's algorithm for high-speed multiplication and one-bit-at-a-time division. Both signed and unsigned integers for all byte aligned data types mentioned above are supported.

The Am29332 is designed to support 80–90 ns microcycle time. The device is packaged in a 168-lead pin-grid-array package.

SIMPLIFIED BLOCK DIAGRAM



BD005240

Order # 05730B

Am29334

Four-Port, Dual-Access Register File

ADVANCED INFORMATION

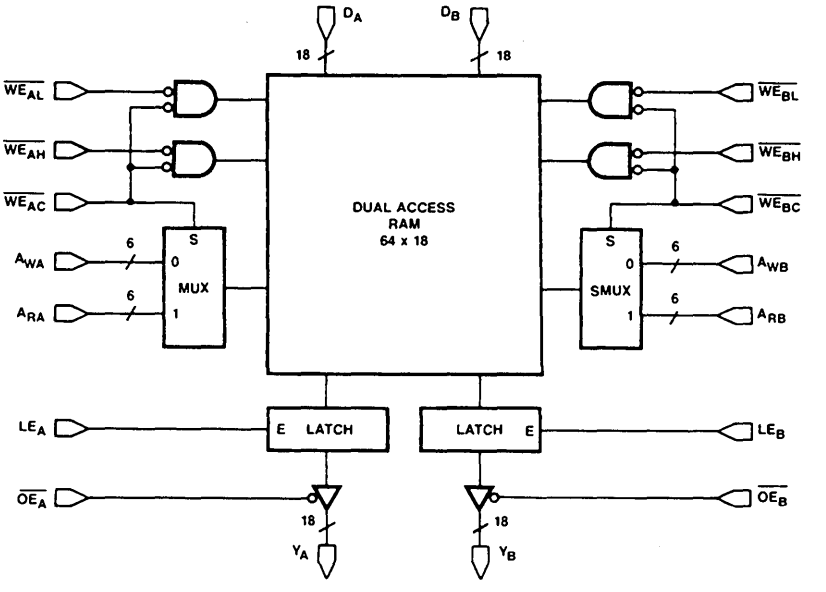
DISTINCTIVE CHARACTERISTICS

- **Fast**
  With an access time of 20ns, the Am29334 supports 80–90ns microcycle time when used with the Am29300 Family for 32-bit systems.
- **64 x 18 Bits Wide Register File**
  The Am29334 is a high-performance, high-speed, dual-access RAM with two READ ports and two WRITE ports.
- **Cascadable**
  The Am29334 is cascadable to support either wider word widths, deeper register files, or both.
- **Simplified Timing Control**
  Control for write enable timing and for on-chip read/write multiplexer are derived from a single-phase clock input.
- **Byte Parity Storage**
  Width of 18 bits facilitates byte parity storage for each port and provides consistency with the Am29332 32-bit ALU.
- **Byte Write Capability**
  Individual byte-write enables allows byte or full word write.

GENERAL DESCRIPTION

The Am29334 is a 64-word deep and 18-bit wide dual-access register file designed to support other members of the Am29300 Family by providing high-speed storage. It has two write and two read ports for data and four 6-bit address ports. Two address ports are associated with each pair of read and write data ports, one to read data and the other to write. The device is capable of performing two reads and two writes in one cycle. The 18-bit wide register file allows storage of byte parity to support parity check and generate in the Am29332 32-bit ALU. Independent control for each read and write data port allows the Am29334 to be used as a high-speed shared memory or as a mailbox for a multiprocessor system. The device is designed with an access time of 20ns. It is housed in a 120 lead-pin-grid-array package.

BLOCK DIAGRAM



BD003022

Order # 05731B

Am2910A

Microprogram Controller

DISTINCTIVE CHARACTERISTICS

- **Twelve Bits Wide**
  Addresses up to 4096 words of microcode with one chip. All internal elements are a full 12 bits wide.
- **Internal Loop Counter**
  Pre-settable 12-bit down-counter for repeating instructions and counting loop iterations.
- **Four Address Sources**
  Microprogram Address may be selected from microprogram counter, branch address bus, 9-level push/pop stack, or internal holding register.
- **Sixteen Powerful Microinstructions**
  Executes 16 sequence control instructions, most of which are conditional on external condition input, state of internal loop counter, or both.
- **Output Enable Controls for Three Branch Address Sources**
  Built-in decoder function to enable external devices onto branch address bus. Eliminates external decoder.
- **Fast**
  The Am2910A supports 100ns cycle times and is 25 – 30% faster than the Am2910.

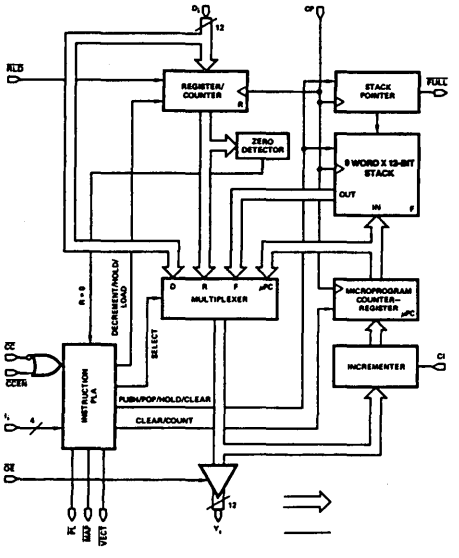
GENERAL DESCRIPTION

The Am2910A Microprogram controller is an address sequencer intended for controlling the sequence of execution of microinstructions stored in microprogram memory. Besides the capability of sequential access, it provides conditional branching to any microinstruction within its 4096-microword range. A last-in, first-out stack provides microsubroutine return linkage and looping capability; there are nine levels of nesting of microsubroutines. Microinstruction loop count control is provided with a count capacity of 4096.

During each microinstruction, the microprogram controller provides a 12-bit address from one of four sources: 1) the microprogram address register (μPC), which usually contains an address one greater than the previous address; 2) an external (direct) input (D); 3) a register/counter (R) retaining data loaded during a previous microinstruction; or 4) a nine-deep last-in, first-out stack (F).

The Am2910A is a speed improved plug-in replacement of the Am2910 featuring AMD's ion-implanted micro-oxide (IMOX) processing and offering 25 – 30% speed improvement. The Am2910A also features a nine-word deep stack versus the five-deep stack of the Am2910.

BLOCK DIAGRAM



BDR02320

Am29C10A

CMOS Microprogram Controller

PRELIMINARY

DISTINCTIVE CHARACTERISTICS

- **Low power**
  The CMOS Am29C10A supports 125 ns cycle times at 20% the power of the equivalent bipolar Am2910A.
- **Twelve bits wide**
  Addresses up to 4096 words of microcode with one chip. All internal elements are a full 12 bits wide.
- **Internal loop counter**
  Pre-settable 12-bit down-counter for repeating instructions and counting loop iterations.
- **Four address sources**
  Microprogram address may be selected from microprogram counter, branch address bus, 9-level push/pop stack, or internal holding register.
- **Sixteen powerful microinstructions**
  Executes 16 sequence control instructions, most of which are conditional on external condition input, state of internal loop counter, or both.
- **Output Enable controls three branch-address sources**
  Built-in decoder function to enable external devices onto branch address bus. Eliminates external decoder.

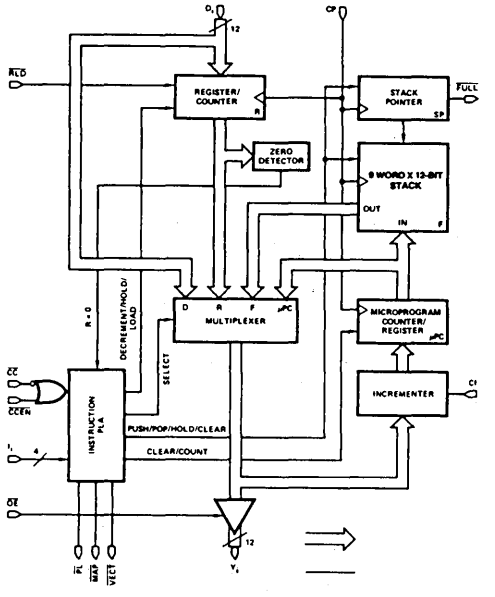
GENERAL DESCRIPTION

The Am29C10A Microprogram Controller is an address sequencer intended for controlling the sequence of execution of microinstructions stored in microprogram memory. Besides the capability of sequential access, it provides conditional branching to any microinstruction within its 4096-microword range. A last-in, first-out stack provides microsubroutine return linkage and looping capability; there are nine levels of nesting of microsubroutines. Microinstruction loop count control is provided with a count capacity of 4096.

During each microinstruction, the Microprogram Controller provides a 12-bit address from one of four sources: 1) the Microprogram Address Counter/Register (μPC), which usually contains an address one greater than the previous address; 2) an external (Direct) input (D); 3) a Register/counter (R) retaining data loaded during a previous microinstruction; or 4) a nine-deep last-in, first-out stack/File (F).

The Am29C10A is a CMOS plug-in replacement of the Am2910A. The Am29C10A-10 is a 10 MHz version and the Am29C10A-20 is a 20 MHz version.

BLOCK DIAGRAM



BDR02321

RELATED PRODUCTS

Part No.	Description
Am29C101	16-Bit CMOS Microprocessor Slice
Am2914	Vectored Interrupt Controller
Am2918	Pipeline Register
Am2922	Condition Code MUX
Am25LS377	Status Register
Am27S35	Registered PROM
Am29818	SSR Diagnostics/Pipeline Register

Am29112

A High-Performance 8-Bit Slice Microprogram Sequencer

ADVANCED INFORMATION

DISTINCTIVE CHARACTERISTICS

- **Expandable**
  8-bit Slice, cascadable up to 16-bits
- **Deep stack**
  A 33 deep on-chip stack is used for subroutine linkage, interrupt handling and loop control.
- **Hold feature**
  A hold pin facilitates multiple sequencer implementations.
- **Interruptible at the microprogram level**
  Two kinds of interrupts: maskable and unmaskable.
- **Powerful loop control**
  When cascaded, two counters can act as a single 16-bit counter or two independent 8-bit counters.
- **Powerful addressing modes**
  Features direct, multiway, multiway relative and program counter relative addressing.

GENERAL DESCRIPTION

The Am29112 is a high performance interruptible microprogram sequencer intended for use in very high speed microprogrammed machines and optimized for the new state-of-the-art ALU's and other processing components.

The Am29112 is designed to operate in 10MHz microprogrammed systems.

It has an instruction set featuring relative and multiway branching, a rich variety of looping constructs, and provision for loading and unloading the on-chip stack.

Interrupts are accepted at the microcycle level and serviced in a manner completely transparent to the interrupted microcode.

APPLICATION NOTES REFERENCE

- Microprogrammed CPU using Am29116
- An intelligent fast disk controller
- Am29116 architecture speeds pixel manipulation in interactive bit-mapped graphics

BLOCK DIAGRAM

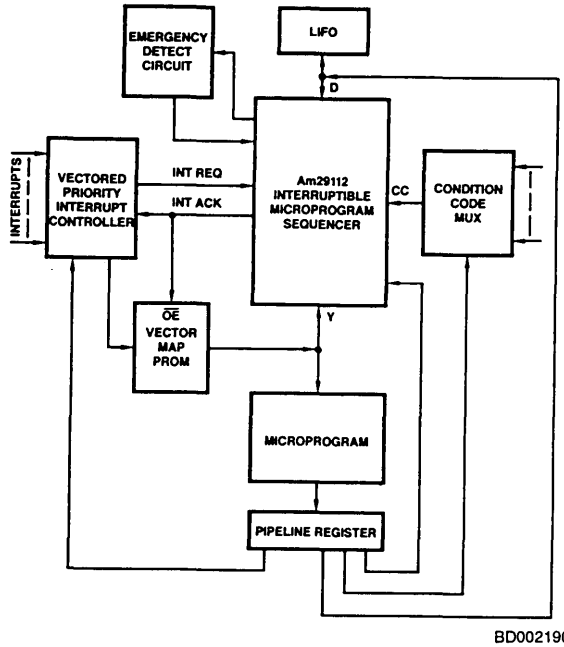


Figure 1. Am29112 in a Single Pipelined System.

Am29116A/29L116A

A High-Performance 16-Bit Bipolar Microprocessor

ADVANCED INFORMATION

DISTINCTIVE CHARACTERISTICS

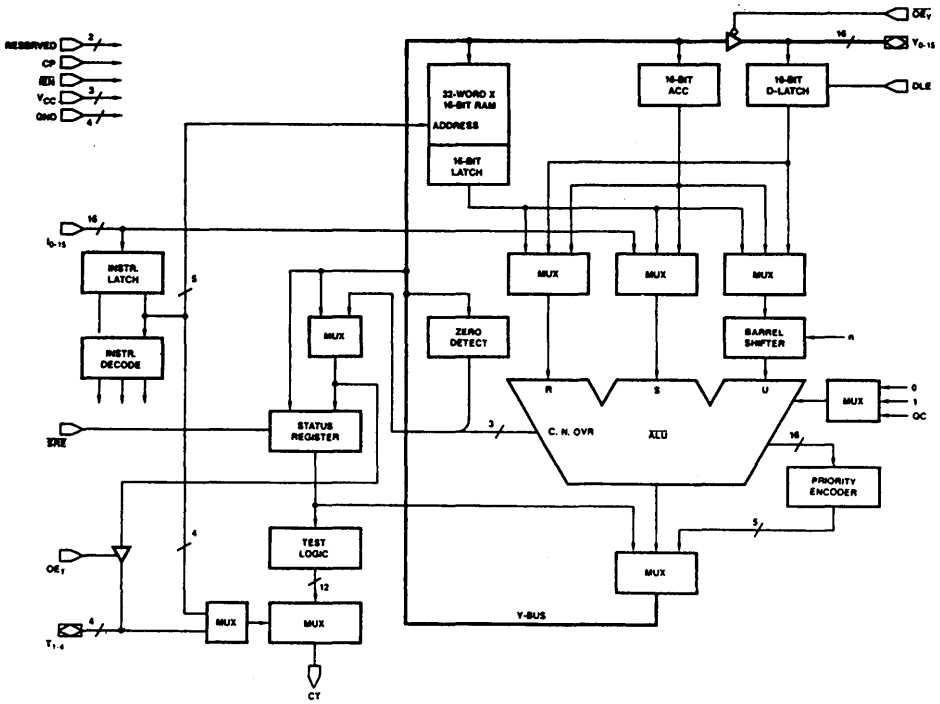
- **Second Generation of the Am29116 16-Bit Microprocessor**
  Internal ECL circuitry and second generation of Ion implanted Oxide-Isolated (IMOX™) process technology, are combined to provide a faster version of the Am29116 and a lower power version of the 29116
- **Plug-In Replacement for the Am29116**
  The Am29116A and Am29L116A are pin-for-pin replacements for the Am29116
- **Improved Speed**
  The 29116A has 25 – 30% speed improvement on the critical paths relative to Am29116
- **Reduced Power**
  The 29L11A operates at reduced power requirements and compatible performace relative to the 29116

GENERAL DESCRIPTION

The Am29116A/L116A microprogrammable 16-bit bipolar microprocessors fabricated using the second generation of AMD's IMOX process technology. The architectures of the Am29116A/L116A are identical to the Am29116's, optimized for high-performance peripheral controllers such as graphics controllers, disk controllers, communication controllers, front-end concentrators and modems. The Am29116A is also extremely suitable for high-speed, general-purpose 16-bit CPU applications when combined with the Am29517A 16 x 16 multiplier.

BLOCK DIAGRAM

Am29116A



BD001960

Am29PL141

Fuse Programmable Controller (FPC)

PRELIMINARY

DISTINCTIVE CHARACTERISTICS

- Implements complex fuse programmable state machines
- 64 words of 32-bit-wide microprogram memory
- Serial Shadow Register (SSR™) diagnostics on chip (programmable option)
- 20 MHz clock rate, 28-pin DIP
- 29 high-level microinstructions
  - Conditional branching
  - Conditional looping
  - Conditional subroutine call
  - Multiway branch
- 16 outputs, 7 conditional inputs

GENERAL DESCRIPTION

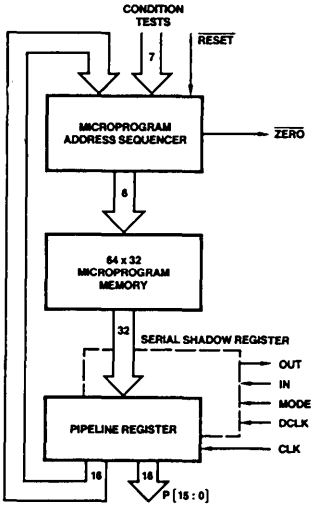
The Am29PL141 is a single-chip Fuse Programmable Controller (FPC) which allows implementation of complex state machines and controllers by programming the appropriate sequence of microinstructions. A repertoire of jumps, loops, and subroutine calls, which can be conditionally executed based on the test inputs, provides the designer with powerful control flow primitives.

The Am29PL141 FPC also allows distribution of intelligent control throughout the system. It off-loads the central controller by distributing FPCs as the control for various self-contained functional units, such as register file/ALU, I/O, interrupt, diagnostic, and bus control units.

A microprogram address sequencer is the heart of the FPC. It provides the microprogram address to an internal 64-word by 32-bit PROM. The fuse programming algorithm is almost identical to that used for AMD's Programmable Array Logic family.

As an option, the Am29PL141 may be programmed to have on chip SSR diagnostics capability. Microinstructions can be serially shifted in, executed, and the results shifted out to facilitate system diagnostics.

BLOCK DIAGRAM



BDR02340

RELATED PRODUCTS

Part No.	Description
Am2914	Vectored Priority Interrupt Controller
Am29100	Controller Family Products

SSR is a trademark of Advanced Micro Devices, Inc.

```
DEVICE DSP_PAL_U72 (pal22V10)

PIN
        /CCLK    = 1      /BUSY    = 22
        /FLY0    = 2      /BHEN    = 21
        /BYTEL   = 3      HACK     = 20
        /BYTEH   = 4      /BPRO    = 19
        HREQ     = 5      /BREQ    = 18
        /Q       = 6      /CDRI    = 17
        /BPRN    = 7      /CDRR    = 16
        /BSY     = 8      /CDLI    = 15
                          /CDLR    = 14;

BEGIN
    BEGIN
        BREQ      :=   HREQ;

        /HACK     :=   /BUSY;

        BUSY      :=   BREQ * BPRN * /BSY          +
                       BREQ * BUSY;
    END;

    BEGIN
        BPRO       =   /HREQ * BPRN;

        BHEN       =   BUSY;

        CDLR       =   /Q * HREQ * /FLY0            +
                       /Q * BYTEH * /FLY0           +
                       /Q * BYTEL * /FLY0;

        CDLI       =   /Q * HREQ * FLY0             +
                       /Q * BYTEH * FLY0            +
                       /Q * BYTEL * FLY0;

        CDRR       =   Q * HREQ * /FLY0             +
                       Q * BYTEH * /FLY0            +
                       Q * BYTEL * /FLY0;

        CDRI       =   Q * HREQ * FLY0              +
                       Q * BYTEH * FLY0             +
                       Q * BYTEL * FLY0;
    END;
END.
```

```
DEVICE DSP_PAL_U73 (pal22V10)

PIN
        /BHEN    = 1      /WE8     = 21
        /BYTEL   = 2      /WE7     = 20
        /BYTEH   = 3      /WE6     = 19
        HREQ     = 4      /WE5     = 18
        /IOW     = 5      /WE4     = 17
        WEI      = 6      /WE3     = 16
        WER      = 7      /WE2     = 15
        /FLY0    = 8      /WE1     = 14
        /Q       = 9      /DMAH    = 22
        /IOR     = 10;

BEGIN
        WE1  =  Q * /WER                              +
                /Q * HREQ * IOW * /FLY0               +
                /Q * /FLY0 * IOW * BYTEH;

        WE2  =  Q * /WER                              +
                /Q * HREQ * IOW * /FLY0               +
                /Q * /FLY0 * IOW * BYTEH * BHEN       +
                /Q * IOW * /FLY0 * BYTEL;

        WE3  =  Q * /WEI                              +
                /Q * HREQ * IOW * FLY0                +
                /Q * FLY0 *IOW * BYTEH;

        WE4  =  Q * /WEI                              +
                /Q * HREQ * IOW * FLY0                +
                /Q * FLY0 * IOW * BYTEH * BHEN        +
                /Q * IOW * FLY0 * BYTEL;

        WE5  =  /Q * /WER                             +
                Q * HREQ * IOW * /FLY0                +
                Q * /FLY0 * IOW * BYTEH;

        WE6  =  /Q * /WER                             +
                Q * HREQ * IOW * /FLY0                +
                Q * /FLY0 * IOW * BYTEH * BHEN        +
                Q * IOW * /FLY0 * BYTEL;

        WE7  =  /Q * /WEI                             +
                Q * HREQ * IOW * FLY0                 +
                Q * FLY0 * IOW * BYTEH;

        WE8  =  /Q * /WEI                             +
                Q * HREQ * IOW * FLY0                 +
                Q * FLY0 * IOW * BYTEH * BHEN         +
                Q * IOW * FLY0 * BYTEL;

        DMAH =  IOW * HREQ                            +
                IOR * HREQ;

END.
```

```
DEVICE DSP_PAL_U74 (pal22V10)

PIN
        /IOR     = 1      /BYTEH   = 23
        /EOUT    = 2      /BYTEL   = 22
        /A0      = 3      /INIT    = 21
        /A1      = 4      /LOADL   = 20
        /A2      = 5      /LOADH   = 19
        /A4      = 6      CD3      = 18
        /IOW     = 7      CD2      = 17
        HREQ     = 8      /POLL    = 16
        /BHEN    = 9      INSTR    = 15
        /HINIT   = 10     /ADDRL   = 14
        /ADSTB   = 11;

BEGIN
        /INSTR   =  A4                                              +
                    /EOUT                                           +
                    /IOW                                            +
                    /A2                                             +
                    /A1                                             +
                    /A0;

        INIT     =  HINIT                                           +
                    /A4 * EOUT * IOW * A2 * A1 * /A0;

        ADDRL    =  /A4 * EOUT * IOW * A2 * /A1 * A0;

        LOADH    =  /A4 * EOUT * IOW * A2 * /A1 * /A0;

        LOADL    =  /A4 * EOUT * IOW * /A2 * A1 * A0                +
                    /A4 * EOUT * IOW * BHEN * A2 * /A1 * /A0;

        BYTEH    =  /A4 * EOUT * IOW * /A2 * A1 * /A0               +
                    /A4 * EOUT * IOR * /A2 * A1 * /A0;

        BYTEL    =  /A4 * EOUT * IOW * /A2 * /A1 * A0               +
                    /A4 * EOUT * IOR * /A2 * /A1 * A0;

        POLL     =  /A4 * EOUT * IOR * A2 * A1 * A0;

        /CD2     =  /A4 * EOUT * IOW * /BHEN * /A2 * A1 * /A0       +
                    /A4 * EOUT * IOR * /BHEN * /A2 * A1 * /A0       +
                    /A4 * EOUT * IOW * /BHEN * A2 * /A1 * /A0;

        /CD3     =  /A4 * EOUT * IOW * A0                           +
                    /A4 * EOUT * IOR * /A2 * /A1 * A0               +
                    /A4 * EOUT * IOW * BHEN * /A2 * A1 * /A0        +
                    /A4 * EOUT * IOR * BHEN * /A2 * A1 * /A0        +
                    /A4 * EOUT * IOW * BHEN * A2 * /A1 * /A0        +
                    A4 * EOUT * IOW                                 +
                    A4 * EOUT * IOR                                 +
                    HREQ * /ADSTB;

END.
```

```
TITLE        AmDSP Digital Signal Processor Definitions
;
;
 WORD 128
; ================================================================
; 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
; 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 9 9 9 9
; 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6
; ================================================================
; S S R R R R R R R R R R R R R R R R R R R R R R R R R R
; R R A A A A R R R S S S A A A A A A B B B B B B M M M M D D D W
; 1 0 L L L L 0 0 0 0 0 0 1 1 2 2 3 3 1 1 2 2 3 3 I I I I I I I E
;     U U U U P P P P P P 1 0 1 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0
;     3 2 1 0 2 1 0 2 1 0                         3 2 1 0 2 1 0
;
;                                Real ALU
; ================================================================
; 9 9 9 9 9 9 8 8 8 8 8 8 8 8 8 8 7 7 7 7 7 7 7 7 7 7 6 6 6 6 6 6
; 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4
; ================================================================
; S S I I I I I I I I I I I I I I I I I I I I I I I I I I I I
; I I A A A A R R R S S S A A A A A A B B B B B B M M M M D D D W
; 1 0 L L L L 0 0 0 0 0 0 1 1 2 2 3 3 1 1 2 2 3 3 I I I I I I I E
;     U U U U P P P P P P 1 0 1 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0
;     3 2 1 0 2 1 0 2 1 0                         3 2 1 0 2 1 0
;
;                              Imaginary ALU
; ================================================================
; 6 6 6 6 5 5 5 5 5 5 5 5 5 5 4 4 4 4 4 4 4 4 4 4 4 3 3 3 3 3 3 3 3
; 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2
; ================================================================
; D R F M O 2 X X E S C 1 5 A A A A A A A A A A A A A A A A A A A A
; P N A S E C S S N I F 1 4 G G G G G G G G G G G G G G G G G G G G
;   D   P L   E E X N A 6 0 1 1 1 1 1 1 1 1 1 9 8 7 6 5 4 3 2 1 0
;       L     L L   C D O 0 8 7 6 5 4 3 2 1 0
;       S     1 0   O R E E
;       P           S
;
;          Multiplier                          Address Generator
; ================================================================
; 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
; 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
; ================================================================
; B       5 5 5 5 I C C C C C C C B B B B B B B B B B B B 1 1 1 1
; U       2 2 2 2 N K K C C C C C R R R R R R R R R R R R 0 0 0 0
; F       0 0 0 0 T S S P S S S S 1 1 9 8 7 6 5 4 3 2 1 0 I I I I
; C       I I S S R E E O E E E E 1 0                     3 2 1 0
; D       1 0 1 0   L L L L L L L
;                   1 0   3 2 1 0
;
;                                Program Sequence
; ================================================================
EJECT
;                  ADDRESS GENERATOR OVERLAY
;
```

```
;                                                D      5 5 5 5 5 5
;                                                I      4 4 4 4 4 4
;                                                T 4 P 0 0 0 0 0 0
;                                                / / S I I S S S S
;                                                F 2 D 1 0 3 2 1 0
;================================================================================
;                                      1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
;                                      1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
;                                      6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
;                              S D I   I I I I I I I I I I I I I I I I
;                              R L E   1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
;                              E E N   5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
;
;                                      AG18-00
EJECT
;
; * * * * * * * * * * * * * * * * * * * * * * * *
;
; Am29501 INSTRUCTION SET
;
; * * * * * * * * * * * * * * * * * * * * * * * *
;
; Am29501 ALU FUNCTIONS (R FUNCTION S)
;
; CONDITIONAL CARRY OUTPUT
;
ADD.CC:          EQU     H#0
SUBS.CC:         EQU     H#4
MOVE.CC:         EQU     H#8
SUBR.CC:         EQU     H#C
;
;INHIBIT CARRY OUTPUT
;
ADD.NC:          EQU     H#1
SUBS.NC:         EQU     H#5
MOVE.NC:         EQU     H#9
SUBR.NC:         EQU     H#D
;
;FORCE CARRY OUTPUT
;
ADD.FC:          EQU     H#2
SUBS.FC:         EQU     H#6
MOVE.FC:         EQU     H#A
SUBR.FC:         EQU     H#E
;
;LOGICAL (NO CARRY USED TO DISTINGUISH FROM 29116 LOGICALS)
;
XOR.NC:          EQU     H#3
AND.NC:          EQU     H#7
INV.NC:          EQU     H#B
OR.NC:           EQU     H#F
;
; Am29501 SOURCE OPERANDS   R, S
;
A1:              EQU     Q#0
```

```
A2:                EQU     Q#1
A3:                EQU     Q#2
B1:                EQU     Q#3
B2:                EQU     Q#4
B3:                EQU     Q#5
SIGN.EXT:          EQU     Q#6
ZERO:              EQU     Q#7
MSP:               EQU     Q#6
LSP:               EQU     Q#7
EJECT
;
; PORT SELECTS
;
M.EQ.A1:           EQU     H#0
M.EQ.A2:           EQU     H#1
M.EQ.A3:           EQU     H#2
M.EQ.B1:           EQU     H#3
M.EQ.B2:           EQU     H#4
M.EQ.B3:           EQU     H#5
M.EQ.AU:           EQU     H#6
M.EQ.DI:           EQU     H#7
MIO.IN:            EQU     H#8
;
D.EQ.A2:           EQU     Q#0
D.EQ.A3:           EQU     Q#1
D.EQ.B2:           EQU     Q#2
D.EQ.B3:           EQU     Q#3
DIO.IN:            EQU     Q#4
;
; Am29501 Register Operations
;
A1.EQ.MP:          EQU     B#00
A1.EQ.DI:          EQU     B#01
A1.EQ.B3:          EQU     B#10
A1.HOLD:           EQU     B#11
A2.EQ.LP:          EQU     B#00
A2.EQ.AU:          EQU     B#01
A2.EQ.A1:          EQU     B#10
A2.HOLD:           EQU     B#11
A3.EQ.MP:          EQU     B#00
A3.EQ.AU:          EQU     B#01
A3.EQ.A2:          EQU     B#10
A3.HOLD:           EQU     B#11
B1.EQ.MP:          EQU     B#00
B1.EQ.DI:          EQU     B#01
B1.EQ.A3:          EQU     B#10
B1.HOLD:           EQU     B#11
B2.EQ.LP:          EQU     B#00
B2.EQ.AU:          EQU     B#01
B2.EQ.B1:          EQU     B#10
B2.HOLD:           EQU     B#11
B3.EQ.MP:          EQU     B#00
B3.EQ.AU:          EQU     B#01
B3.EQ.B2:          EQU     B#10
B3.HOLD:           EQU     B#11
```

```
EJECT
;
;REAL ALU INSTRUCTIONS
;
R.ADD: DEF      2X,ADD.CC,3VQ#0,3VQ#0,2VB#11,2VB#11,2VB#11,2VB#11,2VB#11,
/               2VB#11,4VH#8,3VQ#4,97X
R.SUBS: DEF     2X,SUBS.CC,3VQ#0,3VQ#0,2VB#11,2VB#11,2VB#11,2VB#11,2VB#11,
/               2VB#11,4VH#8,3VQ#4,97X
R.MOVE: DEF     2X,MOVE.NC,3VQ#0,3X,2VB#11,2VB#11,2VB#11,2VB#11,2VB#11,
/               2VB#11,4VH#8,3VQ#4,97X
R.SUBR: DEF     2X,SUBR.CC,3VQ#0,3VQ#0,2VB#11,2VB#11,2VB#11,2VB#11,2VB#11,
/               2VB#11,4VH#8,3VQ#4,97X
R.XOR: DEF      2X,XOR.NC,3VQ#0,3VQ#0,2VB#11,2VB#11,2VB#11,2VB#11,2VB#11,
/               2VB#11,4VH#8,3VQ#4,97X
R.AND: DEF      2X,AND.NC,3VQ#0,3VQ#0,2VB#11,2VB#11,2VB#11,2VB#11,2VB#11,
/               2VB#11,4VH#8,3VQ#4,97X
R.INV: DEF      2X,INV.NC,3VQ#0,3VQ#0,2VB#11,2VB#11,2VB#11,2VB#11,2VB#11,
/               2VB#11,4VH#8,3VQ#4,97X
R.OR: DEF       2X,OR.NC,3VQ#0,3VQ#0,2VB#11,2VB#11,2VB#11,2VB#11,2VB#11,
/               2VB#11,4VH#8,3VQ#4,97X
;
;IMAGINARY ALU INSTRUCTIONS
;
I.ADD: DEF      34X,ADD.CC,3VQ#0,3VQ#0,2VB#11,2VB#11,2VB#11,2VB#11,2VB#11,
/               2VB#11,4VH#8,3VQ#4,65X
I.SUBS: DEF     34X,SUBS.CC,3VQ#0,3VQ#0,2VB#11,2VB#11,2VB#11,2VB#11,2VB#11,
/               2VB#11,4VH#8,3VQ#4,65X
I.MOVE: DEF     34X,MOVE.NC,3VQ#0,3X,2VB#11,2VB#11,2VB#11,2VB#11,2VB#11,
/               2VB#11,4VH#8,3VQ#4,65X
I.SUBR: DEF     34X,SUBR.CC,3VQ#0,3VQ#0,2VB#11,2VB#11,2VB#11,2VB#11,2VB#11,
/               2VB#11,4VH#8,3VQ#4,65X
I.XOR: DEF      34X,XOR.NC,3VQ#0,3VQ#0,2VB#11,2VB#11,2VB#11,2VB#11,2VB#11,
/               2VB#11,4VH#8,3VQ#4,65X
I.AND: DEF      34X,AND.NC,3VQ#0,3VQ#0,2VB#11,2VB#11,2VB#11,2VB#11,2VB#11,
/               2VB#11,4VH#8,3VQ#4,65X
I.INV: DEF      34X,INV.NC,3VQ#0,3VQ#0,2VB#11,2VB#11,2VB#11,2VB#11,2VB#11,
/               2VB#11,4VH#8,3VQ#4,65X
I.OR: DEF       34X,OR.NC,3VQ#0,3VQ#0,2VB#11,2VB#11,2VB#11,2VB#11,2VB#11,
/               2VB#11,4VH#8,3VQ#4,65X
EJECT
;
; * * * * * * * * * * * * * * * * * * * * * * * *
; Am29517 MULTIPLIER
; * * * * * * * * * * * * * * * * * * * * * * * *
;
MX.RALU:        EQU     H#0     ;MULTIPLIER IS REAL ALU MIO
MX.HOLD:        EQU     H#2     ;USE PREVIOUS MULTIPLIER
MX.IMAG:        EQU     H#4     ;MULTIPLIER IS IMAG MEM
MX.REAL:        EQU     H#8     ;MULTIPLIER IS REAL MEM
MX.CONST:       EQU     H#C     ;MULTIPLIER IS COEF ROM
MX.COS:         EQU     H#C     ;MULTIPLIER IS FFT COS ROM
MX.SIN:         EQU     H#D     ;MULTIPLIER IS FFT SIN ROM
MXY.2C:         EQU     B#1     ;MULTIPLIERS ARE TWO'S COMP
MXY.MAG:        EQU     MXY.2C*         ;UNSIGNED MAGNITUDE
MY.OE:          EQU     B#0     ;ENABLE LSP ON Y
```

```
MY.IN:          EQU     MY.OE*          ;ENABLE Y INPUT
MP.FRAC:        EQU     B#0     ;FRACTIONAL MULTIPLY (FA)
MP.INTG:        EQU     MP.FRAC*        ;INTEGER MULTIPLY
MP.ROUND:       EQU     B#1     ;ROUNDED MSP
MP.TRUNC:       EQU     MP.ROUND*       ;TRUNCATED MSP
MP.MSP:         EQU     B#0     ;MOST SIGNIFICANT PRODUCT OUT
MP.LSP:         EQU     MP.MSP*         ;LEAST SIGNIFICANT PRODUCT OUT
MP.OE:          EQU     B#0     ;OUTPUT ENABLE
;
; MULTIPLIER INSTRUCTIONS
;
MSPROD: DEF     65X,1VB#1,1VB#0,MP.MSP  ;MOST SIGNIFICANT PRODUCT
/               ,1VB#1,1VB#1,4VH#2
/               ,22X,1VB#1,31X
LSPROD: DEF     65X,1VB#1,1VB#0,MP.LSP  ;LEAST SIGNIFICANT PRODUCT
/               ,1VB#1,1VB#1,4VH#2
/               ,22X,1VB#1,31X
;
; * * * * * * * * * * * * * * * * * * * * * * * *
; Am25S10 SHIFTER
; * * * * * * * * * * * * * * * * * * * * * * * *
;
NO.SHIFT:       EQU     B#11            ;LSB CONNECTED TO I(-3)
SHIFT.R1:       EQU     B#10
SHIFT.R2:       EQU     B#01
SHIFT.R3:       EQU     B#00
;
SHIFT.OE:       EQU     B#0
EJECT
;
; * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;
;
;       INDEX TO Am29116 INSTRUCTIONS - [i] REFERS TO ALLOWED MNEMONICS GROUP
;
;               SINGLE OPERAND [1], [2], [3], [4]
;               TWO OPERAND    [5], [6], [7], [8]
;               SHIFT          [9], [10], [11]
;               ROTATE         [12], [13], [14]
;               BIT-ORIENTED   [15], [16], [17]
;               ROTATE & MERGE [18]
;               ROTATE & COMPR [19]
;               PRIORITIZE     [20], [21], [22], [23], [24], [25]
;               CYCLIC REDUNDANCY CHECKS
;               NOOP
;               STATUS         [26], [27]
;               TEST STATUS    [CT]
;
; ************************************************
; GENERAL MNEMONICS
; ************************************************
;
; ****************************
; BYTE - WORD MODE SELECT [M]
; ****************************
```

```
;
B:              EQU             1B#0    ; BYTE MODE
W:              EQU             1B#1    ; WORD MODE
;
;
; ************************************************
; 32 RAM REGISTERS [R]
; ************************************************
;
R0:             EQU             5D#0%   ; 00000
R1:             EQU             5D#1%   ;
R2:             EQU             5D#2%   ;
R3:             EQU             5D#3%   ;
R4:             EQU             5D#4%   ;
R5:             EQU             5D#5%   ;
R6:             EQU             5D#6%   ;
R7:             EQU             5D#7%   ;
R8:             EQU             5D#8%   ;
R9:             EQU             5D#9%   ;
R10:            EQU             5D#10%  ;
R11:            EQU             5D#11%  ;
R12:            EQU             5D#12%  ;
R13:            EQU             5D#13%  ;
R14:            EQU             5D#14%  ;
R15:            EQU             5D#15%  ;
R16:            EQU             5D#16%  ;
R17:            EQU             5D#17%  ;
R18:            EQU             5D#18%  ;
R19:            EQU             5D#19%  ;
R20:            EQU             5D#20%  ;
R21:            EQU             5D#21%  ;
R22:            EQU             5D#22%  ;
R23:            EQU             5D#23%  ;
R24:            EQU             5D#24%  ;
R25:            EQU             5D#25%  ;
R26:            EQU             5D#26%  ;
R27:            EQU             5D#27%  ;
R28:            EQU             5D#28%  ;
R29:            EQU             5D#29%  ;
R30:            EQU             5D#30%  ;
R31:            EQU             5D#31%  ;
EJECT
; * * * * * * * * * * * * * * * * *
; Am29116 CONTROL LINES
; 16-Bit Bipolar Microprocessor
; * * * * * * * * * * * * * * * * *
;
OEYEN:          EQU             B#0     ; Y BUS ENABLE
OEYDIS:         EQU             B#1
;
DLE.EN:         EQU             B#1     ; DATA LATCH ENABLE
DLE.DIS:        EQU             B#0
;
OETEN:          EQU             B#1     ; T BUS ENABLE
OETDIS:         EQU             B#0
```

```
;
SRE.EN:         EQU             B#0     ; STATUS REGISTER ENABLE
SRE.DIS:        EQU             B#1
;
IEN:            EQU             B#0     ; INSTRUCTION ENABLE
IDIS:           EQU             B#1
;
; * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;
;
EJECT
; ******************************
; SINGLE OPERAND INSTRUCTIONS
; ******************************
;
; OPCODES [1]
;
MOVE:           EQU             H#C
COMP:           EQU             H#D     ; COMPLEMENT
INC:            EQU             H#E     ; INCREMENT
NEG:            EQU             H#F     ; NEGATE
;
; SOURCE-DESTINATION SELECT [2]
;
SORA:           EQU             H#0     ; RAM   ACC
SORY:           EQU             H#2     ; RAM   Y BUS
SORS:           EQU             H#3     ; RAM   STATUS
SOAR:           EQU             H#4     ; ACC   RAM
SODR:           EQU             H#6     ; D     RAM
SOIR:           EQU             H#7     ; I     RAM
SOZR:           EQU             H#8     ; 0     RAM
SOZER:          EQU             H#9     ; D(0E) RAM
SOSER:          EQU             H#A     ; D(SE) RAM
SORR:           EQU             H#B     ; RAM   RAM
;
; ************************************************************
;
SOR: DEF        74X,1VB#1,B#01,3VB#010,         ;SINGLE OPERAND RAM
/               1V,  B#10,4V,    4V,        5V%, 4X,2VB#11,2VB#11,24X
;
;               MODE,QUAD,OPCODE,SOURCE-DEST,REGISTER
;                [M]       [1]     [2]        [R]
;
; ************************************************************
;
; SOURCE (R/S) [3]
;
SOA:            EQU             H#4     ; ACC
SOD:            EQU             H#6     ; D
SOI:            EQU             H#7     ; I
SOZ:            EQU             H#8     ; 0
SOZE:           EQU             H#9     ; D(0E)
SOSE:           EQU             H#A     ; D(SE)
;
; DESTINATION [4]
```

```
;
NRY:            EQU             D#0     ; Y BUS
NRA:            EQU             D#1     ; ACC
NRS:            EQU             D#4     ; STATUS
NRAS:           EQU             D#5     ; ACC,STATUS
;
; ****************************************************************
SONR: DEF       74X,1VB#1,B#01,3VB#010,         ; SINGLE OPERAND NON-RAM
/               1V,  B#11,4V,    4V,    5V%,  4X,2VB#11,2VB#11,24X
;
;               MODE,QUAD,OPCODE,SOURCE,DESTINATION
;                [M]       [1]    [3]      [4]
; ****************************************************************
EJECT
; ***********************************
; TWO OPERAND INSTRUCTIONS
; ***********************************
;
; OPCODES [5]
;
SUBR:           EQU             H#0     ; S minus R
SUBRC:          EQU             H#1     ; S minus R with carry
SUBS:           EQU             H#2     ; R minus S
SUBSC:          EQU             H#3     ; R minus S with carry
ADD:            EQU             H#4     ; R plus S
ADDC:           EQU             H#5     ; R plus S with carry
AND:            EQU             H#6     ; R . S
NAND:           EQU             H#7     ; R . S
EXOR:           EQU             H#8     ; R   S
NOR:            EQU             H#9     ; R + S
OR:             EQU             H#A     ; R + S
EXNOR:          EQU             H#B     ; R   S
;
;
; SOURCE-DESTINATION [6]                  ; R     S       DEST
;
TORAA:          EQU             H#0     ; RAM   ACC     ACC
TORIA:          EQU             H#2     ; RAM   I       ACC
TODRA:          EQU             H#3     ; D     RAM     ACC
TORAY:          EQU             H#8     ; RAM   ACC     Y BUS
TORIY:          EQU             H#A     ; RAM   I       Y BUS
TODRY:          EQU             H#B     ; D     RAM     Y BUS
TORAR:          EQU             H#C     ; RAM   ACC     RAM
TORIR:          EQU             H#E     ; RAM   I       RAM
TODRR:          EQU             H#F     ; D     RAM     RAM
;
; ****************************************************************
TOR1: DEF       74X,1VB#1,B#01,3VB#010,         ; TWO OPERAND RAM (1)
/               1V,  B#00,4V,          4V,    5V%,  4X,2VB#11,2VB#11,24X
;
;               MODE,QUAD,SOURCE-DEST,OPCODE,REGISTER
;                [M]          [6]     [5]     [R]
; ****************************************************************
;
; THE [i] IN THE COMMENT BELOW THE VARIABLE-FIELD REFERS TO THE ALLOWED
```

```
; MNEMONIC GROUP.  EXAMPLE: MODE REFERS VIA [M] TO THE BYTE-WORD SELECT.
; EXAMPLE: THE ALLOWED OPCODE SUBSTITUTIONS IN TOR1 COME FROM GROUP [5]
; WHILE THE ALLOWED SOURCE-DESTINATIONS COME FROM GROUP [6].
;
EJECT
;
;
; SOURCE-DESTINATION [7]                          R     S       DEST
;
TODAR:          EQU             H#1     ; D     ACC     RAM
TOAIR:          EQU             H#2     ; ACC   I       RAM
TODIR:          EQU             H#5     ; D     I       RAM
;
; ************************************************************
TOR2: DEF       74X,1VB#1,B#01,3VB#010,         ; TWO OPERAND RAM (2)
/               1V,  B#10,4V,          4V,    5V%, 4X,2VB#11,2VB#11,24X
;
;               MODE,QUAD,SOURCE-DEST,OPCODE,REGISTER
;               [M]           [7]       [5]    [R]
; ************************************************************
;
; SOURCE [8]                                    R     S
;
TODA:           EQU             H#1     ; D     ACC
TOAI:           EQU             H#2     ; ACC   I
TODI:           EQU             H#5     ; D     I
; ************************************************************
TONR: DEF       74X,1VB#1,B#01,3VB#010,         ; TWO OPERAND NON-RAM
/               1V,   B#11,4V,    4V,    5V%, 4X,2VB#11,2VB#11,24X
;
;               MODE, QUAD,SOURCE,OPCODE,DESTINATION
;               [M]          [8]    [5]     [4]
; ************************************************************
EJECT
; ************************************************
; SHIFT INSTRUCTIONS
; ************************************************
;
; DIRECTION AND INPUT [9]
;
SHUPZ:          EQU             H#0     ; UP 0
SHUP1:          EQU             H#1     ; UP 1
SHUPL:          EQU             H#2     ; UP QLINK
SHDNZ:          EQU             H#4     ; DOWN 0
SHDN1:          EQU             H#5     ; DOWN 1
SHDNL:          EQU             H#6     ; DOWN QLINK
SHDNC:          EQU             H#7     ; DOWN QC
SHDNOV:         EQU             H#8     ; DOWN QN QOVR
;
;
; SOURCE [10]
;
SHRR:           EQU             H#6     ; RAM   RAM
SHDR:           EQU             H#7     ; D     RAM
;
```

```
;
; ******************************************************
SHFTR: DEF      74X,1VB#1,B#01,3VB#010,          ; SHIFT RAM
/               1V,  B#10,4V,          4V,     5V%,  4X,2VB#11,2VB#11,24X
;
;               MODE,QUAD,SOURCE,DIRECT-INPT,REGISTER
;               [M]        [10]       [9]      [R]
; ******************************************************
;
;
; SOURCE [11]
;
SHA:            EQU             H#6      ; ACC
SHD:            EQU             H#7      ; D
;
;
; ******************************************************
SHFTNR: DEF     74X,1VB#1,B#01,3VB#010,  ; SHIFT NON-RAM
/               1V,  B#11,4V,     4V,     5V%,  4X,2VB#11,2VB#11,24X
;
;               MODE,QUAD,SOURCE,DIRECT-INP,DESTINATION
;               [M]          [11]   [9]         [4](NRY; NRA ONLY)
; ******************************************************
;
EJECT
; ******************************************************
;ROTATE INSTRUCTIONS
; ******************************************************
;
; SOURCE-DESTINATION [12]
;
RTRA:           EQU             H#C      ; RAM   ACC
RTRY:           EQU             H#E      ; RAM   Y BUS
RTRR:           EQU             H#F      ; RAM   RAM
;
;
; ******************************************************
ROTR1: DEF      74X,1VB#1,B#01,3VB#010,          ; ROTATE RAM (1)
/               1V,  B#00,4V,4V,         5V%,  4X,2VB#11,2VB#11,24X
;
;               MODE,QUAD,N,SOURCE-DEST,REGISTER
;               [M]      [N]     [12]     [R]
; ******************************************************
;
; SOURCE-DESTINATION [13]
;
RTAR:           EQU             H#0      ; ACC   RAM
RTDR:           EQU             H#1      ; D     RAM
;
;
; ******************************************************
ROTR2: DEF      74X,1VB#1,B#01,3VB#010,          ; ROTATE RAM (2)
/               1V,  B#01,4V,4V,         5V%,  4X,2VB#11,2VB#11,24X
;
;               MODE,QUAD,N,SOURCE-DEST,REGISTER
```

```
;                [M]      [N]    [13]      [R]
; ***************************************************
;
; SOURCE DESTINATION [14]
;
RTDY:            EQU              D#24     ; D      Y BUS
RTDA:            EQU              D#25     ; D      ACC
RTAY:            EQU              D#28     ; ACC    Y BUS
RTAA:            EQU              D#29     ; ACC    ACC
;
;
; ***************************************************
ROTNR: DEF       74X,1VB#1,B#01,3VB#010,          ; ROTATE NON-RAM
/                1V,  B#11,4V,H#C,       5V%,  4X,2VB#11,2VB#11,24X
;
;                MODE,QUAD,N,FIXED CODE,DESTINATION
;                [M]      [N]                [14]
; ***************************************************
EJECT
; ***************************************************
; BIT ORIENTED INSTRUCTIONS
; ***************************************************
;
; OPCODES [15]
;
SETNR:           EQU              H#D      ; SET RAM, BIT N
RSTNR:           EQU              H#E      ; RESET RAM, BIT N
TSTNR:           EQU              H#F      ; TEST RAM, BIT N
;
;
; ***************************************************
BOR1: DEF        74X,1VB#1,B#01,3VB#010,          ; BIT ORIENTED RAM (1)
/                1V,  B#11,4V,4V,    5V%,  4X,2VB#11,2VB#11,24X
;
;                MODE,QUAD,N,OPCODE,REGISTER
;                [M]      [N]  [15]    [R]
; ***************************************************
;
;
; OPCODES [16]
;
LD2NR:           EQU              H#C      ; 2^N ---> RAM
LDC2NR:          EQU              H#D      ; 2^N ---> RAM
A2NR:            EQU              H#E      ; RAM + 2^N --> RAM
S2NR:            EQU              H#F      ; RAM - 2^N --> RAM
;
;
; ***************************************************
BOR2: DEF        74X,1VB#1,B#01,3VB#010,          ; BIT ORIENTED RAM (2)
/                1V,  B#10,4V,4V,    5V%,  4X,2VB#11,2VB#11,24X
;
;                MODE,QUAD,N,OPCODE,REGISTER
;                [M]      [N]  [16]    [R]
; ***************************************************
EJECT
```

```
;
;
;
;
;
; OPCODES [17]
;
TSTNA:          EQU             D#0     ; TEST ACC, BIT N
RSTNA:          EQU             D#1     ; RESET ACC, BIT N
SETNA:          EQU             D#2     ; SET ACC, BIT N
A2NA:           EQU             D#4     ; ACC + 2"N ←← ACC
S2NA:           EQU             D#5     ; ACC - 2"N ←←ACC
LD2NA:          EQU             H#6     ; 2"N ←← ACC
LDC2NA:         EQU             D#7     ; 2"N ←← ACC
TSTND:          EQU             D#16    ; TEST D, BIT N
RSTND:          EQU             D#17    ; RESET D, BIT N
SETND:          EQU             D#18    ; SET D, BIT N
A2NDY:          EQU             D#20    ; D + 2"N ←← Y BUS
S2NDY:          EQU             D#21    ; D - 2"N ←← Y BUS
LD2NY:          EQU             D#22    ; 2"N ←← Y BUS
LDC2NY:         EQU             D#23    ; 2"N ←← Y BUS
;
;
; ***********************************************************
BONR: DEF       74X,1VB#1,B#01,3VB#010,         ; BIT ORIENTED NON-RAM
/               1V,  B#11,4V,B#1100,    5V%,  4X,2VB#11,2VB#11,24X
;
;               MODE,QUAD,N,FIXED CODE,OPCODE
;               [M]     [N]             [17]
; ***********************************************************
EJECT
; ***************************************************
; ROTATE AND MERGE
; ***************************************************
;
; SOURCE-DEST SELECT [U,S,MASK-DEST] [18]
;
;                                       ROT   NON-ROT  MASK-DEST
MDAI:           EQU             H#7     ; D     ACC     I
MDAR:           EQU             H#8     ; D     ACC     RAM
MDRI:           EQU             H#9     ; D     RAM     I
MDRA:           EQU             H#A     ; D     RAM     ACC
MARI:           EQU             H#C     ; ACC   RAM     I
MRAI:           EQU             H#E     ; RAM   ACC     I
;
;
; ***********************************************************
ROTM: DEF       74X,1VB#1,B#01,3VB#010,         ;ROTATE AND MERGE
/               1V,  B#01,4V,4V,        5V%,  4X,2VB#11,2VB#11,24X
;
;               MODE,QUAD,N,SOURCE-DEST,REGISTER
;               [M]     [N]    [18]       [R]
; ***********************************************************
;
;
```

```
;
;
;
; ************************************************
; ROTATE AND COMPARE
; ************************************************
;
; ROT.SRC(U)-NON ROT.SRC(S)/DEST-MASK(S)[19]
;
CDAI:           EQU             H#2     ; D     ACC     I
CDRI:           EQU             H#3     ; D     RAM     I
CDRA:           EQU             H#4     ; D     RAM     ACC
CRAI:           EQU             H#5     ; RAM   ACC     I
;
;
; *******************************************
ROTC: DEF       74X,1VB#1,B#01,3VB#010,         ; ROTATE AND COMPARE
/               1V,  B#01,4V,4V,            5V%, 4X,2VB#11,2VB#11,24X
;
;               MODE,QUAD,N,SOURCE-DEST-MASK,REGISTER
;               [M]     [N]       [19]        [R]
; *******************************************
EJECT
;
; ************************************************
; PRIORITIZE
; ************************************************
;
; SOURCE [20]
;
PRT1A:          EQU             H#7     ; ACC
PR1D:           EQU             H#9     ; D
;
;
; DESTINATION [21]
;
PR1A:           EQU             H#8     ; ACC
PR1Y:           EQU             H#A     ; Y BUS
PR1R:           EQU             H#B     ; RAM
;
; *******************************************
PRT1: DEF       74X,1VB#1,B#01,3VB#010,         ; RAM ADDR MASK(S)
/               1V,  B#10,4V,         4V,   5V%, 4X,2VB#11,2VB#11,24X
;
;               MODE,QUAD,DESTINATION,SOURCE,REG-MASK
;               [M]         [21]    [20]    [R]
; *******************************************
;
;
; DESTINATION [23]
;
PR2A:           EQU             H#0     ; ACC
PR2Y:           EQU             H#2     ; Y BUS
;
; MASK (S) [22]
```

```
;
PRA:            EQU             H#8     ; ACC
PRZ:            EQU             H#A     ; 0
PRI:            EQU             H#B     ; I
;
;
; ***********************************************
PRT2: DEF       74X,1VB#1,B#01,3VB#010,         ; PRIORITIZE RAM
/               1V,  B#10,4V,  4V,  5V%, 4X,2VB#11,2VB#11,24X
;
;               MODE,QUAD,MASK,DEST,REG=SOURCE
;               [M]       [22] [23]    [R]
; ***********************************************
EJECT
;
; SOURCE (R) [24]
;
PR3R:           EQU             H#3     ; RAM
PR3A:           EQU             H#4     ; ACC
PR3D:           EQU             H#6     ; D
;
;
; ***********************************************
PRT3: DEF       74X,1VB#1,B#01,3VB#010,         ; PRIORITIZE RAM
/               1V,  B#10,4V,  4V,    5V%, 4X,2VB#11,2VB#11,24X
;
;               MODE,QUAD,MASK,SOURCE,REG=DEST
;               [M]       [22]  [24]    [R]
; ***********************************************
;
;
; SOURCE (R) [25]
;
PRTA:           EQU             H#4     ; ACC
PRTD:           EQU             H#6     ; D
;
;
; ***********************************************
PRTNR: DEF      74X,1VB#1,B#01,3VB#010, ; PRIORITIZE NON-RAM
/               1V,  B#11,4V,  4V,    5V%, 4X,2VB#11,2VB#11,24X
;
;               MODE,QUAD,MASK,SOURCE,DESTINATION
;               [M]       [22]  [25]    [4](NRY,NRA ONLY)
; ***********************************************
EJECT
;
;
; ***********************************************
; CYCLIC REDUNDANCY CHECK
; ***********************************************
;
;
; *****************************************
CRCF: DEF 74X,1VB#1,B#01,3VB#010,
/       B#11001100011,5V%,4X,2VB#11,2VB#11,24X  ; FORWARD
```

```
; ******************************************
;
;
; ******************************************
CRCR: DEF 74X,1VB#1,B#01,3VB#010,
/       B#11001101001,5V%, 4X,2VB#11,2VB#11,24X ; REVERSE
; ******************************************
;
;
;
;
; ******************************************
;
; NOOP
;
; ******************************************
NOOP: DEF 74X,1VB#1,B#01,3VB#010,H#7140, 4X,2VB#11,2VB#11,24X; NO OPERATION
; ******************************************
;
EJECT
; ************************************************
; STATUS
; ************************************************
;
; OPCODE [26]
;
SONZC:          EQU             5D#3%   ; SET OVR,N,C,Z
SL:             EQU             5D#5%   ; SET LINK
SF1:            EQU             5D#6%   ; SET FLAG 1
SF2:            EQU             5D#9%   ; SET FLAG 2
SF3:            EQU             5D#10%  ; SET FLAG 3
;
;
; ************************************************
SETST: DEF 74X,1VB#1,B#01,3VB#010,
/       B#011,H#BA,5V%, 4X,2VB#11,2VB#11,24X    ; SET STATUS
;
;                                       OPCODE
;                                        [26]
; ************************************************
;
;
;
; OPCODE [27]
;
RONCZ:          EQU             D#3%    ; RESET OVR,N,C,Z
RL:             EQU             D#5%    ; RESET LINK
RF1:            EQU             D#6%    ; RESET FLAG 1
RF2:            EQU             D#9%    ; RESET FLAG 2
RF3:            EQU             D#10%   ; RESET FLAG 3
;
;************************************************
RSTST: DEF 74X,1VB#1,B#01,3VB#010,
/       B#011,H#AA,5V%, 4X,2VB#11,2VB#11,24X            ; RESET STATUS
```

```
;
;                                        OPCODE
;                                        [27]
; ***************************************************
EJECT
;
;
; ***************************************************
SVSTR: DEF      74X,1VB#1,B#01,3VB#010,          ; SAVE STATUS-RAM
/               1V,  B#10,H#7A, 5V%, 4X,2VB#11,2VB#11,24X
;
;               MODE,QUAD,FIXED,RAM ADDRESS/DEST
;               [M]                      [R]
; ***************************************************
;
;
;***************************************************
SVSTNR: DEF     74X,1VB#1,B#01,3VB#010,          ; SAVE STATUS NON-RAM
/               1V, B#11,H#7A, 5V%, 4X,2VB#11,2VB#11,24X
;
;               MODE,QUAD,FIXED,DESTINATION
;               [M]              [4](NRY,NRA ONLY)
; ***************************************************
;
;
; ************************************************
; TEST STATUS
; ************************************************
;
; OPCODE (CT)
;
TNOZ:           EQU             D#0     ; TEST (N OVR) + Z
TNO:            EQU             D#2     ; TEST N OVR
TZ:             EQU             D#4     ; TEST Z
TOVR:           EQU             D#6     ; TEST OVR
TLOW:           EQU             D#8     ; TEST LOW
TC:             EQU             D#10    ; TEST C
TZC:            EQU             D#12    ; TEST Z + C
TN:             EQU             D#14    ; TEST N
TL:             EQU             D#16    ; TEST LINK
TF1:            EQU             D#18    ; TEST FLAG 1
TF2:            EQU             D#20    ; TEST FLAG 2
TF3:            EQU             D#22    ; TEST FLAG 3
;
; ************************************************
TEST: DEF 74X,1VB#1,B#01,3VB#010
/       ,B#011,H#9A,5V%, 4X,2VB#11,2VB#11,24X   ; TEST STATUS
;
; ************************************************
;
IMMED: DEF      74X,1VB#1,B#01010,16V$,4X,2VB#11,2VB#11,24X
EJECT
;
; * * * * * * * * * * * * * * * * * * * * * * * *
; Am29540 FFT ADDRESS GENERATOR
```

```
; * * * * * * * * * * * * * * * * * * * * * * *
;
ADR.HOLD:       EQU     B#00
ADR.RST:        EQU     B#01
ADR.LOAD:       EQU     B#10
ADR.INC:        EQU     B#11
;
RADIX.2:        EQU     B#0
RADIX.4:        EQU     RADIX.2*
PSD:            EQU     B#0     ;PRESCRAMBLED DATA
NORM.ORD:       EQU     PSD*    ;NORMAL ORDER
DIT:            EQU     B#1
DIF:            EQU     DIT*
ADR.OE:         EQU     B#0
;
ADR1:           EQU     H#0     ;DATA ADDRESS 1 FOR RADIX 2/4
ADR2:           EQU     H#1     ;DATA ADDRESS 2 FOR RADIX 2/4
ADR3:           EQU     H#2     ;DATA ADDRESS 3 FOR RADIX 4
ADR4:           EQU     H#3     ;DATA ADDRESS 4 FOR RADIX 4
ALT.ADR1:       EQU     H#4     ;ALTERNATE ADDRESS 1 FOR RADIX 2/4
ALT.ADR2:       EQU     H#5     ;ALTERNATE ADDRESS 2 FOR RADIX 2/4
ALT.ADR3:       EQU     H#6     ;ALTERNATE ADDRESS 3 FOR RADIX 4
ALT.ADR4:       EQU     H#7     ;ALTERNATE ADDRESS 4 FOR RADIX 4
CONST1:         EQU     H#8     ;CONSTANT ADDRESS 1 FOR RADIX 2/4 OR SHADING
CONST2:         EQU     H#9     ;CONSTANT ADDRESS 2 FOR RADIX 4
CONST3:         EQU     H#A     ;CONSTANT ADDRESS 3 FOR RADIX 4
CONST1.S:       EQU     H#B     ;INVERTED CONSTANT ADDRESS 1 FOR SHADING
RVI.ADR1:       EQU     H#C     ;REAL VALUE INPUT ADDRESS 1
RVI.ADR2:       EQU     H#D     ;REAL VALUE INPUT ADDRESS 2
RVI.ADR3:       EQU     H#E     ;REAL VALUE INPUT ADDRESS 3
RVI.ADR4:       EQU     H#F     ;REAL VALUE INPUT ADDRESS 4
;
; * * * * * * * * * * * * * * * * * * * * * * *
; Am29520 PIPELINE REGISTERS
; * * * * * * * * * * * * * * * * * * * * * * *
;
ADP.SHFT:       EQU     B#00    ;SHIFT ADDR THROUGH PIPELINE
ADP.LDB:        EQU     B#01    ;LOAD & SHIFT SECOND HALF OF PIPELINE
ADP.LDA:        EQU     B#10    ;LOAD & SHIFT FIRST HALF OF PIPELINE
ADP.HOLD:       EQU     B#11    ;NO OP
;
ADP.A1:         EQU     B#11    ;REG A1 TO OUTPUT
ADP.A2:         EQU     B#10    ;REG A2 TO OUTPUT
ADP.B1:         EQU     B#01    ;REG B1 TO OUTPUT
ADP.B2:         EQU     B#00    ;REG B2 TO OUTPUT
;
ADP.OE: EQU     B#0     ;OUTPUT ENABLE
;
; ADDRESS GENERATOR INSTRUCTIONS
;
ADG.HOLD: DEF   74X,1VB#1,B#10001,3X,1VB#1,1VB#0,1VB#1,ADR.HOLD,4VH#0,8X,
/               2VB#11,2VB#11,24X
ADG.RST: DEF    74X,1VB#1,B#10001,3X,1VB#1,1VB#0,1VB#1,ADR.RST,4VH#0,8X,
/               2VB#11,2VB#11,24X
ADG.LOAD: DEF   74X,1VB#1,B#10001,3X,1VB#1,1VB#0,1VB#1,ADR.LOAD,4VH#0,8X,
```

```
/                2VB#11,2VB#11,24X
ADG.INC: DEF     74X,1VB#1,B#10001,3X,1VB#1,1VB#0,1VB#1,ADR.INC,4VH#0,8X,
/                2VB#11,2VB#11,24X
EJECT
;
;
; * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;
; Am2910 MICROPROGRAM CONTROLLER INSTRUCTION SET [20]
;
; * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;
;
; CONDITION CODE MULTIPLEXER
;
IF.HIGH:         EQU     B#0      ;CONDITION PREFIX
IF.LOW:          EQU     IF.HIGH*
UNCOND:          EQU     H#0      ;UNCONDITIONAL BRANCH
NEW.PROC:        EQU     H#1      ;NEW PROCESS COMMAND FROM HOST
INT.ACK:         EQU     H#2      ;INTERRUPT ACK FROM HOST
FFT.ITC:         EQU     H#3      ;Am29540 ITERATION COMPLETE
FFT.DONE:        EQU     H#4      ;Am29540 FFT COMPLETE
ADR.CT:          EQU     H#5      ;Am29116 CONDITION TEST
RALU.S:          EQU     H#8      ;REAL ALU SIGN
RALU.OV:         EQU     H#9      ;REAL ALU OVERFLOW
RALU.Z:          EQU     H#A      ;REAL ALU ZERO
RALU.C:          EQU     H#B      ;REAL ALU CARRY
IALU.S:          EQU     H#C      ;IMAG ALU
IALU.OV:         EQU     H#D
IALU.Z:          EQU     H#E
IALU.C:          EQU     H#F
;
JZ:      DEF     124X,H#0                          ; RESET STACK, MICROPC, ADDRESS
CJS:     DEF     107X,1VB#0,4V,12V$H#FFF,H#1       ; COND JUMP SUBROUTINE
JSR:     DEF     107X,B#0,UNCOND,12V$H#FFF,H#1     ; UNCOND JUMP SUBROUTINE
JMAP:    DEF     112X,12V$H#FFF,H#2                ; UNCOND JUMP TO MEMORY MAP (Di)
CJP:     DEF     107X,1VB#0,4V,12V$H#FFF,H#3       ; COND JUMP PIPELINE
JMP:     DEF     107X,B#0,UNCOND,12V$H#FFF,H#3     ; UNCOND JUMP PIPELINE
PUSH:    DEF     107X,1VB#0,4V,12V$,H#4            ; PUSH STACK, LOAD REG MAYBE
JSRP:    DEF     112X,12V$H#FFF,H#5                ; JUMP SUB FROM REG (F) OR PIPE(T)
CJV:     DEF     107X,1VB#0,4V,12V$H#FFF,H#6       ; COND JUMP TO VECTOR INTER (Di)
JRP:     DEF     112X,12V$H#FFF,H#7                ; JUMP TO REG (F) OR PIPE (T)
RFCT:    DEF     112X,12V$H#FFF,H#8                ; DO LOOP REPEAT UNTIL CTR=0-STACK
RPCT:    DEF     112X,12V$H#FFF,H#9                ; DO LOOP UNTIL CTR=0 - PIPE
CRTN:    DEF     107X,1VB#0,4V,12V$H#FFF,H#A       ; COND RETURN, POP STACK (T)
RTN:     DEF     107X,B#0,UNCOND,12V$H#FFF,H#A     ; UNCOND RETURN
CJPP:    DEF     107X,1VB#0,4V,12V$H#FFF,H#B       ; COND JUMP PIPELINE, POP STACK
LDCT:    DEF     112X,12V$,H#C                     ; LOAD REGISTER, CONTINUE
LOOP:    DEF     124X,H#D                          ; DO LOOP UNTIL TEST=T - STACK
CONT:    DEF     124X,H#E                          ; CONTINUE
TWB:     DEF     107X,1VB#0,4V,12V$H#FFF,H#F       ; THREE WAY BRANCH
;
EJECT
;
; * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

```
;
; Am2925 CYCLE LENGTH SELECT                          [21]
; System Clock Generator and Driver
;
; * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;
; THE FOLLOWING ARE THE CYCLE LENGTH CODES  (PRELIM)
;                                                          EXAMPLE CYCLE (1 OF 4)
CLA:    EQU     Q#0     ; 3 CLOCK PERIODS                  100ns AT 30MHz
CLB:    EQU     Q#1     ; 4                                160ns AT 25MHz
CLC:    EQU     Q#5     ; 5                                200ns AT 25MHz
CLD:    EQU     Q#7     ; 6                                200ns AT 30MHz
CLE:    EQU     Q#3     ; 7                                280ns AT 25MHz
CLF:    EQU     Q#2     ; 8                                320ns AT 25MHz
CLG:    EQU     Q#6     ; 9                                300ns AT 30MHz
CLH:    EQU     Q#4     ; 10 CLOCK PERIODS                 322ns AT 31MHz
;                                       (max crystal frequency is 311MHz)
;
; OTHER CONTROL LINES FOR THE Am2925
; INCOMPLETELY DEFINED AT PRESENT (IN THIS FILE)
;
FIRST.25:       EQU     B#1             ;
LAST.25:        EQU     B#0             ;
;
HALT:           EQU     B#00            ;
NOHALT:         EQU     B#00            ;
;
SINGLSTP:       EQU     B#00            ;
RUN:            EQU     B#00            ;
;
WAITREQ:        EQU     B#0             ;
NOWAITRQ:       EQU     B#1             ;
;
READY:          EQU     B#0             ;
NOTREADY:       EQU     B#1             ;
;
INITIALIZE:     EQU     B#0             ;
NO.INIT:        EQU     B#1             ;
;
;
EJECT
;
; MISCELLANEOUS CONTROLS FOR THE DSP
;
WE:             EQU     B#0     ;MEMORY WRITE ENABLE
NWE:            EQU     WE*     ;NO WRITE
;
RD.MEM: DEF     31X,NWE,31X,NWE,64X
WR.CMPX: DEF    31X,WE,31X,WE,64X
WR.REAL: DEF    31X,WE,31X,NWE,64X
WR.IMAG: DEF    31X,NWE,31X,WE,64X
;
SEL.116:        EQU     B#01
SEL.540:        EQU     B#10
NO.ADDR:        EQU     B#11
```

```
;
BUFCD:          EQU     B#1     ;BUFFER CHIP DISABLE
BUFEN:          EQU     BUFCD*
;
CF.LOAD:        EQU     B#0     ;ENABLE NEW ROM ADDRESS
CF.HOLD:        EQU     CF.LOAD*
;
; DATA PRESCALING
;
DIV.BY.1:       DEF     NO.SHIFT,30X,NO.SHIFT,94X
DIV.BY.2:       DEF     SHIFT.R1,30X,SHIFT.R1,94X
DIV.BY.4:       DEF     SHIFT.R2,30X,SHIFT.R2,94X
DIV.BY.8:       DEF     SHIFT.R3,30X,SHIFT.R3,94X
;
SP:             EQU     B#1     ;SINGLE PRECISION (16 BITS)
DP:             EQU     B#0     ;DOUBLE PRECISION (32 BITS=IMAG:REAL)
;
INTRRUPT: DEF   104X,B#1,23X    ;GENERATE EXTERNAL INTERRUPT
;
;FIELD POSITIONS
;
MISC: DEF       64X,1VB#1,39X,1VB#0,2VB#00,21X
;
NO.OP: DEF      NO.SHIFT,MOVE.NC,A1,A1  ;DO NOTHING
/               ,A1.HOLD,A2.HOLD,A3.HOLD,B1.HOLD,B2.HOLD,B3.HOLD
/               ,MIO.IN,DIO.IN,NWE
/               ,NO.SHIFT,MOVE.NC,A1,A1
/               ,A1.HOLD,A2.HOLD,A3.HOLD,B1.HOLD,B2.HOLD,B3.HOLD
/               ,MIO.IN,DIO.IN,NWE
/               ,SP,MP.TRUNC,MP.FRAC,MP.MSP
/               ,MY.IN,MXY.2C,MX.CONST
/               ,1X,SEL.540,B#111
/               ,3X,DIT,RADIX.2,NORM.ORD,ADR.HOLD,ADR1
/               ,8X,ADP.HOLD,ADP.A1,24X
;
RD.CMD: DEF     NO.SHIFT,MOVE.CC,A1,A1  ;READ MODE INTO 540 AND 29116
/               ,A1.HOLD,A2.HOLD,A3.HOLD,B1.HOLD,B2.HOLD,B3.HOLD
/               ,MIO.IN,DIO.IN,NWE
/               ,NO.SHIFT,MOVE.CC,A1,A1
/               ,A1.HOLD,A2.HOLD,A3.HOLD,B1.HOLD,B2.HOLD,B3.HOLD
/               ,MIO.IN,DIO.IN,NWE
/               ,SP,MP.TRUNC,MP.FRAC,MP.MSP
/               ,MY.IN,MXY.2C,MX.CONST
/               ,1X,NO.ADDR,B#101
/               ,3X,DIT,RADIX.2,NORM.ORD,ADR.HOLD,ADR1
/               ,8X,ADP.HOLD,ADP.A1,24X
;
MAKE.ONE: DEF   NO.SHIFT,MOVE.FC,A1,A1  ;FORCE CARRY INTO IMAG ALU
/               ,A1.HOLD,A2.HOLD,A3.HOLD,B1.HOLD,B2.HOLD,B3.HOLD
/               ,MIO.IN,DIO.IN,NWE
/               ,NO.SHIFT,MOVE.CC,ZERO,A1       ; 0 + CRY = 1
/               ,A1.HOLD,A2.EQ.AU,A3.HOLD,B1.HOLD,B2.HOLD,B3.HOLD
/               ,MIO.IN,DIO.IN,NWE
/               ,DP,MP.TRUNC,MP.FRAC,MP.MSP
/               ,MY.IN,MXY.2C,MX.CONST
/               ,54X
;
END
```

```
TITLE   AMDSP DIGITAL SIGNAL PROCESSOR
; * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;
; IDLE LOOP WAITING FOR PROCESS INSTRUCTION
;
START::
NO.OP & JMP $ + 1                       ; JUMP INSTR FOR JAMMING
;
NO.OP & CJP IF.LOW,NEW.PROC,$           ; WAIT FOR INSTR STROBE
;
NO.OP & CJP IF.HIGH,NEW.PROC,$          ; WAIT FOR STROBE TO LATCH DATA
;
RD.CMD & JMAP                           ; VECTOR TO COMMANDED PROCESS
;
; * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;
; SET INTERRUPT FLAG AND WAIT FOR AN ACKNOWLEDGE
;
FINISH:
                                        ; WAIT FOR ACKNOWLEDGE
INTRRUPT & NO.OP & CONT                 ; NO INTR WHILE DEBUGGING
/; & CJP IF.HIGH,INT.ACK,$
;
NO.OP & JMP START
EJECT
;
; * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;
; FFT PROGRAM
; SIZE HANDLED BY Am29540
;
FFT::
;                                       *** RESET 29540 AND DO NOTHING
ADG.RST & RD.MEM & DIV.BY.2
/& R.MOVE & I.MOVE & MSPROD & MISC
/& CONT
;                                       *** FILL PIPELINE BEFORE WRITING
ADG.HOLD , DIT, RADIX.2, NORM.ORD, ADR2, ADP.LDB
/& RD.MEM & DIV.BY.2
/& R.MOVE & I.MOVE & MSPROD & MISC
/& CONT
;                                       *** READ B OPERAND & COEFFICIENT
ADG.HOLD CF.LOAD, DIT, RADIX.2, NORM.ORD, CONST1, ADP.HOLD, ADP.B1
/& RD.MEM & DIV.BY.2
/& R.MOVE , , , , B1.EQ.DI, , , , DIO.IN
/& I.MOVE , , , , B1.EQ.DI, , , , DIO.IN
/& MSPROD
/& MISC
/& CONT
;                                       *** REAL*COS
ADG.HOLD CF.HOLD, DIT, RADIX.2, NORM.ORD, ADR1, ADP.LDA
/& RD.MEM & DIV.BY.2
/& R.MOVE , , , , B1.HOLD, , , M.EQ.B1
/& I.MOVE , , , , B1.HOLD, , , MIO.IN
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.COS, BUFEN
```

```
/& MISC
/& CONT
;                                       *** REAL*SIN, READ A OPERAND
ADG.INC CF.HOLD, DIT, RADIX.2, NORM.ORD, , ADP.HOLD, ADP.A1
/& RD.MEM & DIV.BY.2
/& R.MOVE , A1.EQ.DI, , , , , , M.EQ.B1, DIO.IN
/& I.MOVE , A1.EQ.DI, , , B1.HOLD, , , MIO.IN, DIO.IN
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.SIN, BUFEN
/& MISC
/& CONT
;                                       *** IMAG*COS
ADG.HOLD CF.HOLD, DIT, RADIX.2, NORM.ORD, ADR2, ADP.LDB
/& RD.MEM & DIV.BY.2
/& R.ADD A1,MSP, A1.HOLD, , A3.EQ.MP, , B2.EQ.AU, , MIO.IN
/& I.MOVE , A1.HOLD, , , B1.HOLD, , , M.EQ.B1
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.COS, BUFCD
/& MISC
/& CONT
;                                       *** IMAG*SIN & READ B
ADG.HOLD CF.LOAD, DIT, RADIX.2, NORM.ORD, CONST1, ADP.HOLD, ADP.B1
/& RD.MEM & DIV.BY.2
/& R.MOVE , A1.HOLD, , A3.HOLD, B1.EQ.DI, B2.HOLD, , MIO.IN, DIO.IN
/& I.SUBS A1,MSP, A1.HOLD, , A3.EQ.MP, B1.EQ.DI, B2.EQ.AU, , M.EQ.B1, DIO.IN
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.SIN, BUFCD
/& MISC
/& CONT
;                                       *** REAL*COS
ADG.HOLD CF.HOLD, DIT, RADIX.2, NORM.ORD, ADR1, ADP.LDA
/& RD.MEM & DIV.BY.2
/& R.SUBS A1,A3, A1.HOLD, A2.EQ.AU, , B1.HOLD, B2.HOLD, , M.EQ.B1
/& I.ADD B2,MSP, A1.HOLD, , A3.HOLD, B1.HOLD, B2.EQ.AU, B3.EQ.MP, MIO.IN
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.COS, BUFEN
/& MISC
/& CONT
;                                       *** PASS 1 LOOP CAN DO SHADING
IT1.LOOP:
;                                       *** REAL*SIN, READ A
ADG.INC CF.HOLD, DIT, RADIX.2, NORM.ORD, , ADP.HOLD, ADP.A1
/& RD.MEM & DIV.BY.2
/& R.ADD B2,MSP, A1.EQ.DI, A2.HOLD, , , B2.EQ.AU, B3.EQ.MP, M.EQ.B1, DIO.IN
/& I.SUBS A1,B3, A1.EQ.DI, A2.EQ.AU, A3.HOLD, B1.HOLD, B2.HOLD, , MIO.IN, DIO.IN
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.SIN, BUFEN
/& MISC
/& CONT
;                                       *** IMAG*COS, WRITE A - B*w
ADG.HOLD CF.HOLD, DIT, RADIX.2, NORM.ORD, ADR2, ADP.LDB, ADP.B2
/& WR.CMPX
/& R.ADD A1,MSP, A1.HOLD, A2.HOLD, A3.EQ.MP, , B2.EQ.AU, B3.HOLD, , D.EQ.B2
/& I.ADD A2,A3, A1.HOLD, A2.EQ.AU, , B1.HOLD, , , M.EQ.B1, D.EQ.B2
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.COS, BUFCD
/& MISC
/& CONT
;                                       *** IMAG*SIN, READ B
ADG.HOLD CF.LOAD, DIT, RADIX.2, NORM.ORD, CONST1, ADP.HOLD, ADP.B1
/& RD.MEM & DIV.BY.2
```

```
/& R.SUBS A2,B3, A1.HOLD, A2.EQ.AU, A3.HOLD, B1.EQ.DI, B2.HOLD, , , DIO.IN
/& I.SUBS A1,MSP, A1.HOLD, A2.HOLD, A3.EQ.MP, B1.EQ.DI, B2.EQ.AU, ,M.EQ.B1, DIO.IN
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.SIN, BUFCD
/& MISC
/& CONT
;                                       *** REAL*COS, WRITE A + B*w
ADG.HOLD CF.HOLD, DIT, RADIX.2, NORM.ORD, ADR1, ADP.LDA, ADP.A2
/& WR.CMPX
/& R.SUBS A1,A3, , A2.EQ.AU, , B1.HOLD, B2.HOLD, , M.EQ.B1, D.EQ.A2
/& I.ADD B2,MSP, A1.HOLD, , A3.HOLD, B1.HOLD, B2.EQ.AU, B3.EQ.MP, MIO.IN, D.EQ.A2
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.COS, BUFEN
/& MISC
/& CJP IF.LOW,FFT.ITC,IT1.LOOP
;
BTF.LOOP:
;                                       *** REAL*SIN, READ A
ADG.INC CF.HOLD, DIT, RADIX.2, NORM.ORD, , ADP.HOLD, ADP.A1
/& RD.MEM & DIV.BY.2
/& R.ADD B2,MSP, A1.EQ.DI, A2.HOLD, , , B2.EQ.AU, B3.EQ.MP, M.EQ.B1, DIO.IN
/& I.SUBS A1,B3, A1.EQ.DI, A2.EQ.AU, A3.HOLD, B1.HOLD, B2.HOLD, , MIO.IN, DIO.IN
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.SIN, BUFEN
/& MISC
/& CONT
;                                       *** IMAG*COS, WRITE A - B*w
ADG.HOLD CF.HOLD, DIT, RADIX.2, NORM.ORD, ADR2, ADP.LDB, ADP.B2
/& WR.CMPX
/& R.ADD A1,MSP, A1.HOLD, A2.HOLD, A3.EQ.MP, , B2.EQ.AU, B3.HOLD, , D.EQ.B2
/& I.ADD A2,A3, A1.HOLD, A2.EQ.AU, , B1.HOLD, , , M.EQ.B1, D.EQ.B2
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.COS, BUFCD
/& MISC
/& CONT
;                                       *** IMAG*SIN, READ B
ADG.HOLD CF.LOAD, DIT, RADIX.2, NORM.ORD, CONST1, ADP.HOLD, ADP.B1
/& RD.MEM & DIV.BY.2
/& R.SUBS A2,B3, A1.HOLD, A2.EQ.AU, A3.HOLD, B1.EQ.DI, B2.HOLD, , , DIO.IN
/& I.SUBS A1,MSP, A1.HOLD, A2.HOLD, A3.EQ.MP, B1.EQ.DI, B2.EQ.AU, ,M.EQ.B1, DIO.IN
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.SIN, BUFCD
/& MISC
/& CONT
;                                       *** REAL*COS, WRITE A + B*w
ADG.HOLD CF.HOLD, DIT, RADIX.2, NORM.ORD, ADR1, ADP.LDA, ADP.A2
/& WR.CMPX
/& R.SUBS A1,A3, , A2.EQ.AU, , B1.HOLD, B2.HOLD, , M.EQ.B1, D.EQ.A2
/& I.ADD B2,MSP, A1.HOLD, , A3.HOLD, B1.HOLD, B2.EQ.AU, B3.EQ.MP, MIO.IN, D.EQ.A2
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.COS, BUFEN
/& MISC
/& CJP IF.LOW,FFT.DONE,BTF.LOOP
;/& CJP IF.LOW,FFT.ITC,BTF.LOOP
;                                       *** REAL*SIN, READ A
ADG.INC CF.HOLD, DIT, RADIX.2, NORM.ORD, , ADP.HOLD, ADP.A1
/& RD.MEM & DIV.BY.2
/& R.ADD B2,MSP, A1.EQ.DI, A2.HOLD, , , B2.EQ.AU, B3.EQ.MP, M.EQ.B1, DIO.IN
/& I.SUBS A1,B3, A1.EQ.DI, A2.EQ.AU, A3.HOLD, B1.HOLD, B2.HOLD, , MIO.IN, DIO.IN
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.SIN, BUFEN
/& MISC
```

```
/& CONT
;/& RPCT BTF.LOOP+1              ;COUNT PASSES FOR TESTING
;
; FLUSH PIPELINE
;                                          *** IMAG*COS, WRITE A - B*w
ADG.HOLD CF.HOLD, DIT, RADIX.2, NORM.ORD, ADR2, ADP.LDB, ADP.B2
/& WR.CMPX
/& R.ADD A1,MSP, A1.HOLD, A2.HOLD, A3.EQ.MP, , B2.EQ.AU, B3.HOLD, , D.EQ.B2
/& I.ADD A2,A3, A1.HOLD, A2.EQ.AU, , B1.HOLD, , , M.EQ.B1, D.EQ.B2
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.COS, BUFCD
/& MISC
/& CONT
;                                          *** IMAG*SIN
ADG.HOLD , DIT, RADIX.2, NORM.ORD, , ADP.HOLD
/& RD.MEM & DIV.BY.2
/& R.SUBS A2,B3, A1.HOLD, A2.EQ.AU, A3.HOLD, , B2.HOLD
/& I.SUBS A1,MSP, A1.HOLD, A2.HOLD, A3.EQ.MP, , B2.EQ.AU, ,M.EQ.B1
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.SIN, BUFCD
/& MISC
/& CONT
;                                          *** WRITE A + B*w
ADG.HOLD , DIT, RADIX.2, NORM.ORD, , ADP.LDA, ADP.A2
/& WR.CMPX
/& R.SUBS A1,A3, , A2.EQ.AU, , , B2.HOLD, , , D.EQ.A2
/& I.ADD B2,MSP, A1.HOLD, , A3.HOLD, , B2.EQ.AU, B3.EQ.MP, , D.EQ.A2
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C
/& MISC
/& CONT
;
ADG.HOLD , DIT, RADIX.2, NORM.ORD, , ADP.HOLD
/& RD.MEM & DIV.BY.2
/& R.ADD B2,MSP, , A2.HOLD, , , B2.EQ.AU, B3.EQ.MP
/& I.SUBS A1,B3, , A2.EQ.AU, A3.HOLD, , B2.HOLD
/& MSPROD & MISC
/& CONT
;                                          *** WRITE A - B*w
ADG.HOLD , DIT, RADIX.2, NORM.ORD, , , ADP.B2
/& WR.CMPX
/& R.MOVE , , , , , , , , D.EQ.B2
/& I.ADD A2,A3, , A2.EQ.AU, , , , , , D.EQ.B2
/& MSPROD & MISC
/& CONT
;
ADG.HOLD , DIT, RADIX.2, NORM.ORD
/& RD.MEM & DIV.BY.2
/& R.SUBS A2,B3, , A2.EQ.AU
/& I.MOVE , , A2.HOLD
/& MSPROD & MISC
/& CONT
;                                          *** WRITE A + B*w
ADG.HOLD , DIT, RADIX.2, NORM.ORD, , , ADP.A2
/& WR.CMPX
/& R.MOVE , , , , , , , , D.EQ.A2
/& I.MOVE , , , , , , , , D.EQ.A2
/& MSPROD & MISC
```

```
/& JMP FINISH
EJECT
;
; * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;
; INVERSE FFT PROGRAM (PRE SCRAMBLED DATA)
; SIZE HANDLED BY Am29540
;
IFFT::
;                                          *** RESET 29540 AND DO NOTHING
ADG.RST & RD.MEM & DIV.BY.2
/& R.MOVE & I.MOVE & MSPROD & MISC
/& CONT
;                                          *** FILL PIPELINE BEFORE WRITING
ADG.HOLD , DIT, RADIX.2, PSD, ADR2, ADP.LDB
/& RD.MEM & DIV.BY.2
/& R.MOVE & I.MOVE & MSPROD & MISC
/& CONT
;                                          *** READ B OPERAND & COEFFICIENT
ADG.HOLD CF.LOAD, DIT, RADIX.2, PSD, CONST1, ADP.HOLD, ADP.B1
/& RD.MEM & DIV.BY.2
/& R.MOVE , , , , B1.EQ.DI, , , , DIO.IN
/& I.MOVE , , , , B1.EQ.DI, , , , DIO.IN
/& MSPROD
/& MISC
/& CONT
;                                          *** REAL*COS OF COMPLEX MULTIPLY
ADG.HOLD CF.HOLD, DIT, RADIX.2, PSD, ADR1, ADP.LDA
/& RD.MEM & DIV.BY.2
/& R.MOVE , , , , B1.HOLD, , , M.EQ.B1
/& I.MOVE , , , , B1.HOLD, , , MIO.IN
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.COS, BUFEN
/& MISC
/& CONT
;                                          *** REAL*SIN, READ A OPERAND
ADG.INC CF.HOLD, DIT, RADIX.2, PSD, , ADP.HOLD, ADP.A1
/& RD.MEM & DIV.BY.2
/& R.MOVE , A1.EQ.DI, , , , , , M.EQ.B1, DIO.IN
/& I.MOVE , A1.EQ.DI, , , B1.HOLD, , , MIO.IN, DIO.IN
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.SIN, BUFEN
/& MISC
/& CONT
;                                          *** IMAG*COS
ADG.HOLD CF.HOLD, DIT, RADIX.2, PSD, ADR2, ADP.LDB
/& RD.MEM & DIV.BY.2
/& R.ADD A1,MSP, A1.HOLD, , A3.EQ.MP, , B2.EQ.AU
/& I.MOVE , A1.HOLD, , , B1.HOLD, , , M.EQ.B1
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.COS, BUFCD
/& MISC
/& CONT
;                                          *** IMAG*SIN, READ B
ADG.HOLD CF.LOAD, DIT, RADIX.2, PSD, CONST1, ADP.HOLD, ADP.B1
/& RD.MEM & DIV.BY.2
/& R.MOVE , A1.HOLD, , A3.HOLD, B1.EQ.DI, B2.HOLD, , , DIO.IN
/& I.ADD A1,MSP, A1.HOLD, , A3.EQ.MP, B1.EQ.DI, B2.EQ.AU, , M.EQ.B1, DIO.IN
```

```
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.SIN, BUFCD
/& MISC
/& CONT
;                                          *** REAL*COS
ADG.HOLD CF.HOLD, DIT, RADIX.2, PSD, ADR1, ADP.LDA
/& RD.MEM & DIV.BY.2
/& R.SUBS A1,A3, A1.HOLD, A2.EQ.AU, , B1.HOLD, B2.HOLD, , M.EQ.B1
/& I.ADD B2,MSP, A1.HOLD, , A3.HOLD, B1.HOLD, B2.EQ.AU, B3.EQ.MP, MIO.IN
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.COS, BUFEN
/& MISC
/& CONT
;
IBTF.LUP:
;                                          *** REAL*SIN, READ A
ADG.INC CF.HOLD, DIT, RADIX.2, PSD, , ADP.HOLD, ADP.A1
/& RD.MEM & DIV.BY.2
/& R.SUBS B2,MSP, A1.EQ.DI, A2.HOLD, , , B2.EQ.AU, B3.EQ.MP, M.EQ.B1, DIO.IN
/& I.SUBS A1,B3, A1.EQ.DI, A2.EQ.AU, A3.HOLD, B1.HOLD, B2.HOLD, , MIO.IN, DIO.IN
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.SIN, BUFEN
/& MISC
/& CONT
;                                          *** IMAG*COS, WRITE A - B*w
ADG.HOLD CF.HOLD, DIT, RADIX.2, PSD, ADR2, ADP.LDB, ADP.B2
/& WR.CMPX
/& R.ADD A1,MSP, A1.HOLD, A2.HOLD, A3.EQ.MP, , B2.EQ.AU, B3.HOLD, , D.EQ.B2
/& I.SUBS A2,A3, A1.HOLD, A2.EQ.AU, , B1.HOLD, , , M.EQ.B1, D.EQ.B2
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.COS, BUFCD
/& MISC
/& CONT
;                                          *** IMAG*SIN, READ B
ADG.HOLD CF.LOAD, DIT, RADIX.2, PSD, CONST1, ADP.HOLD, ADP.B1
/& RD.MEM & DIV.BY.2
/& R.ADD A2,B3, A1.HOLD, A2.EQ.AU, A3.HOLD, B1.EQ.DI, B2.HOLD, , , DIO.IN
/& I.ADD A1,MSP, A1.HOLD, A2.HOLD, A3.EQ.MP, B1.EQ.DI, B2.EQ.AU, ,M.EQ.B1, DIO.IN
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.SIN, BUFCD
/& MISC
/& CONT
;                                          *** REAL*COS, WRITE A + B*w
ADG.HOLD CF.HOLD, DIT, RADIX.2, PSD, ADR1, ADP.LDA, ADP.A2
/& WR.CMPX
/& R.SUBS A1,A3, , A2.EQ.AU, , B1.HOLD, B2.HOLD, , M.EQ.B1, D.EQ.A2
/& I.ADD B2,MSP, A1.HOLD, , A3.HOLD, B1.HOLD, B2.EQ.AU, B3.EQ.MP, MIO.IN, D.EQ.A2
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.COS, BUFEN
/& MISC
/& CJP IF.LOW,FFT.DONE,IBTF.LUP
;                                          *** REAL*SIN, READ A OPERAND
ADG.INC CF.HOLD, DIT, RADIX.2, PSD, , ADP.HOLD, ADP.A1
/& RD.MEM & DIV.BY.2
/& R.SUBS B2,MSP, A1.EQ.DI, A2.HOLD, , , B2.EQ.AU, B3.EQ.MP, M.EQ.B1, DIO.IN
/& I.SUBS A1,B3, A1.EQ.DI, A2.EQ.AU, A3.HOLD, B1.HOLD, B2.HOLD, , MIO.IN, DIO.IN
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.SIN, BUFEN
/& MISC
/& CONT
;
; FLUSH PIPELINE
```

```
;                                              *** IMAG*COS, WRITE A - B*w
ADG.HOLD CF.HOLD, DIT, RADIX.2, PSD, ADR2, ADP.LDB, ADP.B2
/& WR.CMPX
/& R.ADD A1,MSP, A1.HOLD, A2.HOLD, A3.EQ.MP, , B2.EQ.AU, B3.HOLD, , D.EQ.B2
/& I.SUBS A2,A3, A1.HOLD, A2.EQ.AU, , B1.HOLD, , , M.EQ.B1, D.EQ.B2
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.COS, BUFCD
/& MISC
/& CONT
;                                              *** IMAG*SIN
ADG.HOLD , DIT, RADIX.2, PSD, , ADP.HOLD
/& RD.MEM & DIV.BY.2
/& R.ADD A2,B3, A1.HOLD, A2.EQ.AU, A3.HOLD, , B2.HOLD
/& I.ADD A1,MSP, A1.HOLD, A2.HOLD, A3.EQ.MP, , B2.EQ.AU, ,M.EQ.B1
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C, MX.SIN, BUFCD
/& MISC
/& CONT
;                                              *** WRITE A + B*w
ADG.HOLD , DIT, RADIX.2, PSD, , ADP.LDA, ADP.A2
/& WR.CMPX
/& R.SUBS A1,A3, , A2.EQ.AU, , , B2.HOLD, , , D.EQ.A2
/& I.ADD B2,MSP, A1.HOLD, , A3.HOLD, , B2.EQ.AU, B3.EQ.MP, , D.EQ.A2
/& MSPROD MP.ROUND, MP.FRAC, MY.IN, MXY.2C
/& MISC
/& CONT
;
ADG.HOLD , DIT, RADIX.2, PSD, , ADP.HOLD
/& RD.MEM & DIV.BY.2
/& R.SUBS B2,MSP, , A2.HOLD, , , B2.EQ.AU, B3.EQ.MP
/& I.SUBS A1,B3, , A2.EQ.AU, A3.HOLD, , B2.HOLD
/& MSPROD & MISC
/& CONT
;                                              *** WRITE A - B*w
ADG.HOLD , DIT, RADIX.2, PSD, , , ADP.B2
/& WR.CMPX
/& R.MOVE , , , , , , , , D.EQ.B2
/& I.SUBS A2,A3, , A2.EQ.AU, , , , , , D.EQ.B2
/& MSPROD & MISC
/& CONT
;
ADG.HOLD , DIT, RADIX.2, PSD
/& RD.MEM & DIV.BY.2
/& R.ADD A2,B3, , A2.EQ.AU
/& I.MOVE , , A2.HOLD
/& MSPROD & MISC
/& CONT
;                                              *** WRITE A + B*w
ADG.HOLD , DIT, RADIX.2, PSD, , , ADP.A2
/& WR.CMPX
/& R.MOVE , , , , , , , , D.EQ.A2
/& I.MOVE , , , , , , , , D.EQ.A2
/& MSPROD & MISC
/& JMP FINISH
EJECT
;
MXMULT::
```

```
NO.OP & JMP START        ;TBDL
EJECT
;
; * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;
;       PROCESS VECTORS
        ORG     1024 - 8
;
NO.OP & JMP START                ; INSTRUCTION 0 = NOOP
NO.OP & JMP MXMULT               ;             1 = MATRIX MULTIPLY
NO.OP & JMP FILTER               ;             2 = FILTER
NO.OP & JMP FFT                  ;             3 = FFT
NO.OP & JMP IFFT                 ;             4 = INVERSE FFT
        END
```

```
TITLE IIR Filter using the Am29510 and Am29PL141
;
; Microcode field definitions
;
 WORD 32
;
; P31    - Output enable
; P30-26 - Op code
; P25    - Test polarity
; P24-22 - Test input select
; P21-16 - Instruction data
; P15    - Input data taken flag
; P14    - Input select
; P13    - MAC result select
; P12-11 - Shift register control
; P10-09 - Register select
; P08-06 - Coefficient select
; P05    - MAC add/subtract control
; P04    - MAC pass/accumulate control
; P03    - MAC round/truncate control
; P02    - Output data ready flag
; P01    - Unused
; P00    - Error flag
;
; * * * * * * * * * * * * * * * * * * * * * * * *
;
; Application Definitions
;
; * * * * * * * * * * * * * * * * * * * * * * * *
;
DTACK:   EQU     1B#0     ; Input data taken ack
INSEL:   EQU     1B#0     ; Input select tristate control
MACSEL:  EQU     1B#0     ; MAC input select = INSEL*
HOLD:    EQU     2B#11    ; 29520 instructions
LDA:     EQU     2B#10
LDB:     EQU     2B#01
A1:      EQU     2B#11    ; 29520 register select
A2:      EQU     2B#10
B1:      EQU     2B#01
B2:      EQU     2B#00
W0:      EQU     3Q#0     ; Coefficient select
W1:      EQU     3Q#1
W2:      EQU     3Q#2
W3:      EQU     3Q#3
W4:      EQU     3Q#4
RND:     EQU     1B#0     ; 29510 round control
TRUNC:   EQU     RND*
ACCUM:   EQU     1B#0     ; 29510 accumulate control
PASS:    EQU     ACCUM*
SUB:     EQU     1B#0     ; 29510 subtract control
ADD:     EQU     SUB*
DRDY:    EQU     1B#0     ; Output data ready
;
INRDY:   EQU     3Q#0     ; Test condition input I0
READY:   EQU     B#0      ; Polarity control for ready
OUTACK:  EQU     3Q#1     ; Test condition input I1
TAKEN:   EQU     B#0      ; Polarity control for taken
```

```
;
; Only need a few instructions for this simple design
;
CONTINUE:       DEF     6H#2D:,10D#0%,          ; Continue
/       1V,1V,1V,2V,2V,3V,1V,1V,1V,1V,1X,1VB#0
GO.PL.IF:       DEF     6H#39:,1V,3V%:,6V%:,    ; Go to pipeline if
/       1V,1V,1V,2V,2V,3V,1V,1V,1V,1V,1X,1VB#0
WAIT.TILL:      DEF     6H#3A:,1V,3V%:,6V%:,    ; Wait for test input
/       1V,1V,1V,2V,2V,3V,1V,1V,1V,1V,1X,1VB#0
GO.PL:          DEF     6H#39:,1B#1,3Q#6,6V%:,  ; Go to pipeline
/       1V,1V,1V,2V,2V,3V,1V,1V,1V,1V,1X,1VB#0
STOP:           DEF     6H#3A:,1B#0,3Q#6,6X,    ; Error halt
/       1V,1V,1V,2V,2V,3V,1V,1V,1V,1V,1X,1VB#0
;
END
```

```
TITLE IIR SECOND ORDER FILTER SECTION USING THE Am29510 AND Am29PL141
;
; This program implements the equation:
; y(n) = W(0)*x(n) + W(1)*x(n-1) + W(2)*x(n-2) + W(3)*y(n-1) + W(4)*y(n-2)
; The CC input is grounded for unconditional jumps.
; T0 is connected to input data ready
; T1 is connected to output data taken
;
TRUE:   EQU     B#1
;
; Keep writing data on input lines until valid data is written
; No ops keep data sampling synchronous
;
INIT:
WAIT.TILL READY,INRDY,$+1,
/               DTACK*,INSEL,MACSEL*,LDA,A1,W0,ADD,PASS,RND,DRDY*
CONTINUE        DTACK,INSEL*,MACSEL,HOLD,A1,W0,ADD,PASS,RND,DRDY*
CONTINUE        DTACK*,INSEL*,MACSEL,HOLD,A1,W0,ADD,PASS,RND,DRDY*
CONTINUE        DTACK*,INSEL*,MACSEL,HOLD,A1,W0,ADD,PASS,RND,DRDY*
CONTINUE        DTACK*,INSEL*,MACSEL,HOLD,A1,W0,ADD,PASS,RND,DRDY*
;
; Error if next data sample not ready
GO.PL.IF READY*,INRDY,ERROR,
/               DTACK*,INSEL,MACSEL*,LDA,A2,W1,ADD,PASS,TRUNC,DRDY*
;
; Output W(0)*x(n)
CONTINUE        DTACK,INSEL*,MACSEL,LDB,A2,W1,ADD,PASS,TRUNC,DRDY
;
; Error if data not taken
GO.PL.IF TAKEN*,OUTACK,ERROR,
/               DTACK*,INSEL*,MACSEL,HOLD,A2,W1,ADD,PASS,TRUNC,DRDY*
;
; Add W(1)*x(n-1)
CONTINUE        DTACK*,INSEL*,MACSEL,HOLD,A1,W0,ADD,ACCUM,TRUNC,DRDY*
;
; Add W(3)*y(n-1)
CONTINUE        DTACK*,INSEL*,MACSEL,HOLD,B1,W3,ADD,ACCUM,RND,DRDY*
;
; Do W2*x(n-2), read data sample, error if not ready
FOREVER:
GO.PL.IF READY*,INRDY,ERROR,
/               DTACK,INSEL,MACSEL*,LDA,A2,W2,ADD,PASS,TRUNC,DRDY*
;
; Add W1*x(n-1), output previous filtered sample
CONTINUE        DTACK*,INSEL*,MACSEL,LDB,A2,W1,ADD,ACCUM,TRUNC,DRDY
;
; Add W0*x(n), error if output not taken
GO.PL.IF TAKEN*,OUTACK,ERROR,
/               DTACK*,INSEL*,MACSEL,HOLD,A1,W0,ADD,ACCUM,TRUNC,DRDY*
;
; Add W4*y(n-2)
CONTINUE        DTACK*,INSEL*,MACSEL,HOLD,B2,W4,ADD,ACCUM,TRUNC,DRDY*
;
; Add W3*y(n-1), loop indefinitely
GO.PL   FOREVER
/               DTACK*,INSEL*,MACSEL,HOLD,B1,W3,ADD,ACCUM,RND,DRDY*
;
ERROR:
STOP            DTACK*,INSEL*,MACSEL,HOLD,A1,W0,ADD,ACCUM,TRUNC,DRDY*,TRUE
;
END
```

```
TITLE IIR Filter using the Am29510 and the Am29PL141
;
; Microcode field definitions
;
 WORD 32
;
; P31    - Output enable
; P30-26 - Op code
; P25    - Test polarity
; P24-22 - Test input select
; P21-16 - Instruction data
; P15    - Input data taken flag
; P14    - Input select
; P13    - MAC result select
; P12-09 - Operand address
; P08    - Operand RAM write enable
; P07-04 - Coefficient select
; P03    - MAC add/subtract control
; P02    - MAC pass/accumulate control
; P01    - MAC round/truncate control
; P00    - Output data ready flag

;
; * * * * * * * * * * * * * * * * * * * * * * * *
;
; Application Definitions
;
; * * * * * * * * * * * * * * * * * * * * * * * *
;
DTACK:  EQU     1B#0    ; Input data taken ack
INSEL:  EQU     1B#0    ; Input select tristate control
MACSEL: EQU     1B#0    ; MAC input select = INSEL*
XN:     EQU     4H#1    ; Next data sample x(n+1)
X0:     EQU     4H#0    ; Current data sample x(n)
X1:     EQU     4H#F    ; Previous input x(n-1)
X2:     EQU     4H#E
X3:     EQU     4H#D
X4:     EQU     4H#C
X5:     EQU     4H#B
X6:     EQU     4H#A
Y0:     EQU     4H#8    ; Filtered output y(n)
Y1:     EQU     4H#7    ; Previous output y(n-1)
Y2:     EQU     4H#6
Y3:     EQU     4H#5
Y4:     EQU     4H#4
Y5:     EQU     4H#3
Y6:     EQU     4H#2
```

```
W0:     EQU     4H#0    ; Coefficient select
W1:     EQU     4H#1
W2:     EQU     4H#2
W3:     EQU     4H#3
W4:     EQU     4H#4
W5:     EQU     4H#5
W6:     EQU     4H#6
W7:     EQU     4H#7
W8:     EQU     4H#8
W9:     EQU     4H#9
W10:    EQU     4H#A
W11:    EQU     4H#B
W12:    EQU     4H#C
ZERO:   EQU     4H#F    ; Put in a zero coefficient for NOP
WE:     EQU     1B#0    ; Operand RAM write enable
ADD:    EQU     1B#0    ; 29510 add/subtract control
SUBT:   EQU     ADD*
ACCUM:  EQU     1B#0    ; 29510 accumulate control
PASS:   EQU     ACCUM*
RND:    EQU     1B#0    ; 29510 round control
TRUNC:  EQU     RND*
DRDY:   EQU     1B#0    ; Output data ready
;
INRDY:  EQU     3Q#0    ; Test condition input T0
READY:  EQU     B#0     ; Polarity control for ready
OUTACK: EQU     3Q#1    ; Test condition input T1
TAKEN:  EQU     B#0     ; Polarity control for taken
;
;
CONTINUE:       DEF     6H#2D:,10D#0%,          ; Continue
/       1V,1V,1V,4V,1V,4V,1V,1V,1V,1V
LD.CREG:        DEF     6H#24:,1B#1,3Q#6,6V%:,  ; Load counter
/       1V,1V,1V,4V,1V,4V,1V,1V,1V,1V
LOOP:           DEF     6H#28:,4X,6V%:,         ; Go to label if C<>0
/       1V,1V,1V,4V,1V,4V,1V,1V,1V,1V
WAIT.TILL:      DEF     6H#3A:,1V,3V%:,6V%:,    ; Wait for test input
/       1V,1V,1V,4V,1V,4V,1V,1V,1V,1V
;
END
```

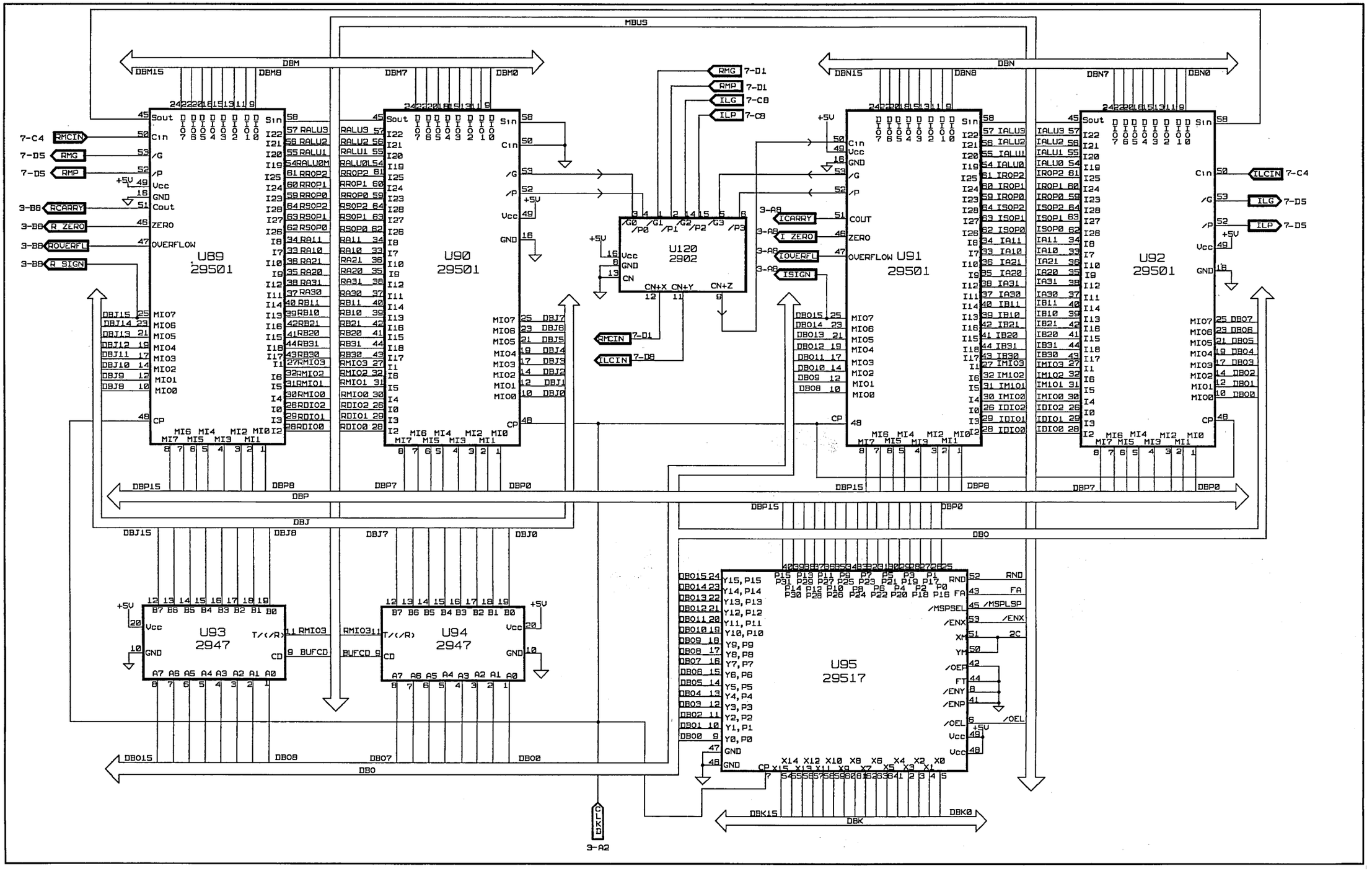
```
TITLE IIR SIXTH ORDER FILTER USING THE Am29510 AND THE AmPL141
;
; This program implements the equation:
; y(n) = W(0)*x(n) + ... + W(6)*x(n-6) + W(7)*y(n-1) + ... + W(12)*y(n-6)
; The CC input is grounded for unconditional jumps.
; T0 is connected to input data ready
; T1 is connected to output data taken
;
TRUE:   EQU     B#1
;
; Make previous operands = 0
;
INIT:
LD.CREG D#15,   DTACK,INSEL*,MACSEL*,XN,WE,W0,ADD,PASS,TRUNC,DRDY*
LOOP $,         DTACK,INSEL*,MACSEL*,XN,WE,W0,ADD,PASS,TRUNC,DRDY*
;
FOREVER:
WAIT.TILL READY,INRDY,$+1,      ; Synchronize to input clock
/               DTACK*,INSEL,MACSEL*,XN,WE,W0,ADD,PASS,TRUNC,DRDY*
; Do W0*x(n)
CONTINUE        DTACK,INSEL*,MACSEL,X0,WE*,W0,ADD,PASS,TRUNC,DRDY*
; Add W1*x(n-1)
CONTINUE        DTACK*,INSEL*,MACSEL,X1,WE*,W1,ADD,ACCUM,TRUNC,DRDY*
; Add W2*x(n-2)
CONTINUE        DTACK*,INSEL*,MACSEL,X2,WE*,W2,ADD,ACCUM,TRUNC,DRDY*
; Add W3*x(n-3)
CONTINUE        DTACK*,INSEL*,MACSEL,X3,WE*,W3,ADD,ACCUM,TRUNC,DRDY*
; Add W4*x(n-4)
CONTINUE        DTACK*,INSEL*,MACSEL,X4,WE*,W4,ADD,ACCUM,TRUNC,DRDY*
; Add W5*x(n-5)
CONTINUE        DTACK*,INSEL*,MACSEL,X5,WE*,W5,ADD,ACCUM,TRUNC,DRDY*
; Add W6*x(n-6)
CONTINUE        DTACK*,INSEL*,MACSEL,X6,WE*,W6,ADD,ACCUM,TRUNC,DRDY*
; Add W7*y(n-1)
CONTINUE        DTACK*,INSEL*,MACSEL,Y1,WE*,W7,ADD,ACCUM,TRUNC,DRDY*
; Add W8*y(n-2)
CONTINUE        DTACK*,INSEL*,MACSEL,Y2,WE*,W8,ADD,ACCUM,TRUNC,DRDY*
; Add W9*y(n-3)
CONTINUE        DTACK*,INSEL*,MACSEL,Y3,WE*,W9,ADD,ACCUM,TRUNC,DRDY*
; Add W10*y(n-4)
CONTINUE        DTACK*,INSEL*,MACSEL,Y4,WE*,W10,ADD,ACCUM,TRUNC,DRDY*
; Add W11*y(n-5)
CONTINUE        DTACK*,INSEL*,MACSEL,Y5,WE*,W11,ADD,ACCUM,TRUNC,DRDY*
; Add W12*y(n-6)
CONTINUE        DTACK*,INSEL*,MACSEL,Y6,WE*,W12,ADD,ACCUM,RND,DRDY*
; Wait for pipeline delay
CONTINUE        DTACK*,INSEL*,MACSEL,Y0,WE*,ZERO,ADD,ACCUM,TRUNC,DRDY*
; Output data, then loop back for next sample
WAIT.TILL TAKEN,OUTACK,FOREVER,
/               DTACK*,INSEL*,MACSEL,Y0,WE,ZERO,ADD,PASS,TRUNC,DRDY
;
END
```

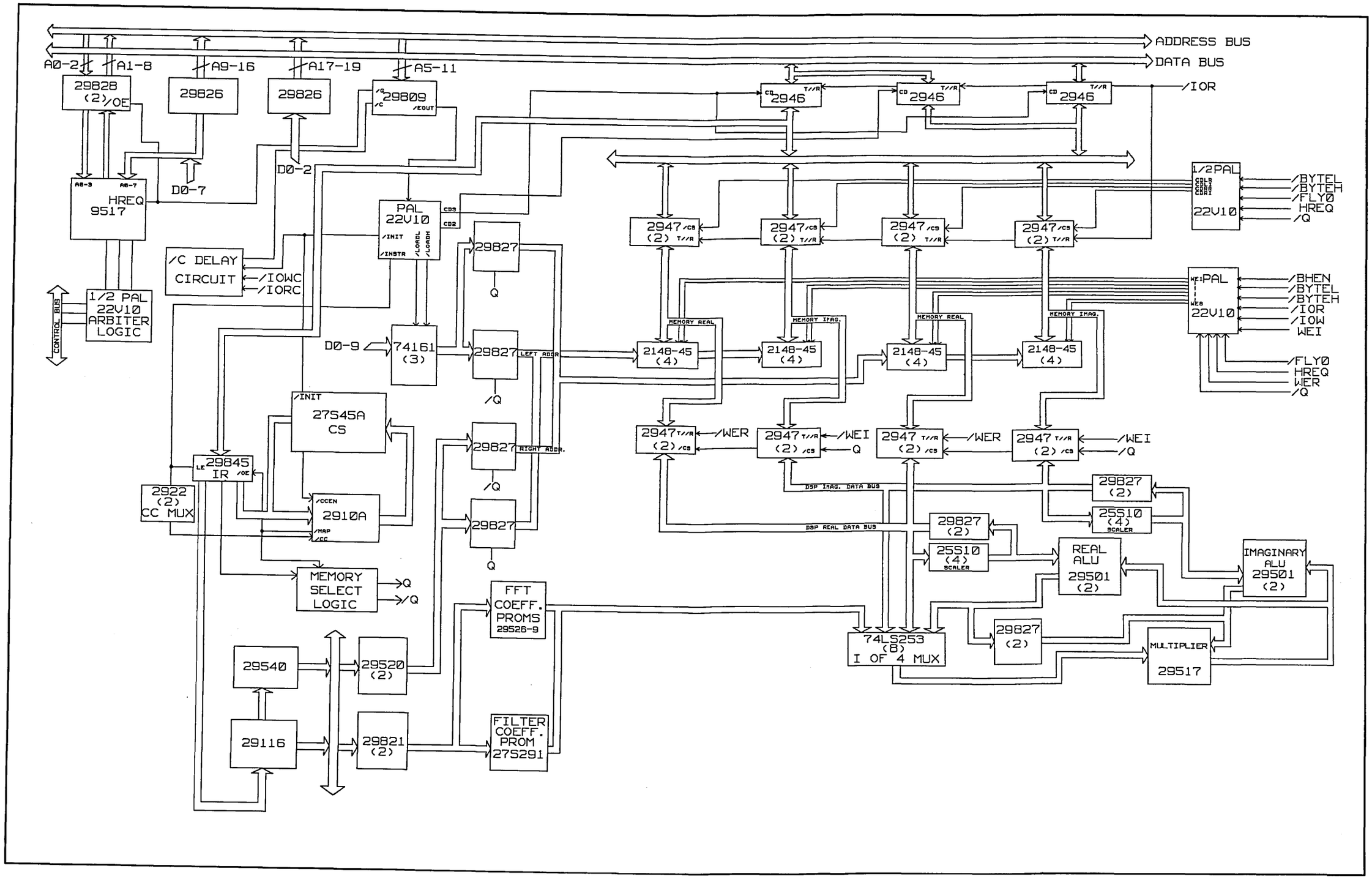
```
TITLE IIR Filter using the Am29510 and the Am29PL141
;
; Microcode field definitions
;
 WORD 32
;
; P31    - Output enable
; P30-26 - Op code
; P25    - Test polarity
; P24-22 - Test input select
; P21-16 - Instruction data
; P15    - Input data taken flag
; P14    - Input select
; P13    - MAC result select
; P12-09 - Operand address
; P08    - Operand RAM write enable
; P07-03 - Coefficient select
; P02    - MAC pass/accumulate control
; P01    - MAC round/truncate control
; P00    - Output data ready flag
;
; * * * * * * * * * * * * * * * * * * * * * * * *
;
; Application Definitions
;
; * * * * * * * * * * * * * * * * * * * * * * * *
;
DTACK:  EQU     1B#0    ; Input data taken ack
INSEL:  EQU     1B#0    ; Input select tristate control
MACSEL: EQU     1B#0    ; MAC input select = INSEL*
XN:     EQU     4H#1    ; Next data sample x(n)
Z0:     EQU     4H#0
Z1:     EQU     4H#F    ; Intermediate result delayed once
Z2:     EQU     4H#E
Z3:     EQU     4H#D
Z4:     EQU     4H#C
Z5:     EQU     4H#B
Z6:     EQU     4H#A
Z7:     EQU     4H#9
Z8:     EQU     4H#8
Z9:     EQU     4H#7
Z10:    EQU     4H#6
Z11:    EQU     4H#5
Z12:    EQU     4H#4
Z13:    EQU     4H#3
Z14:    EQU     4H#2
```

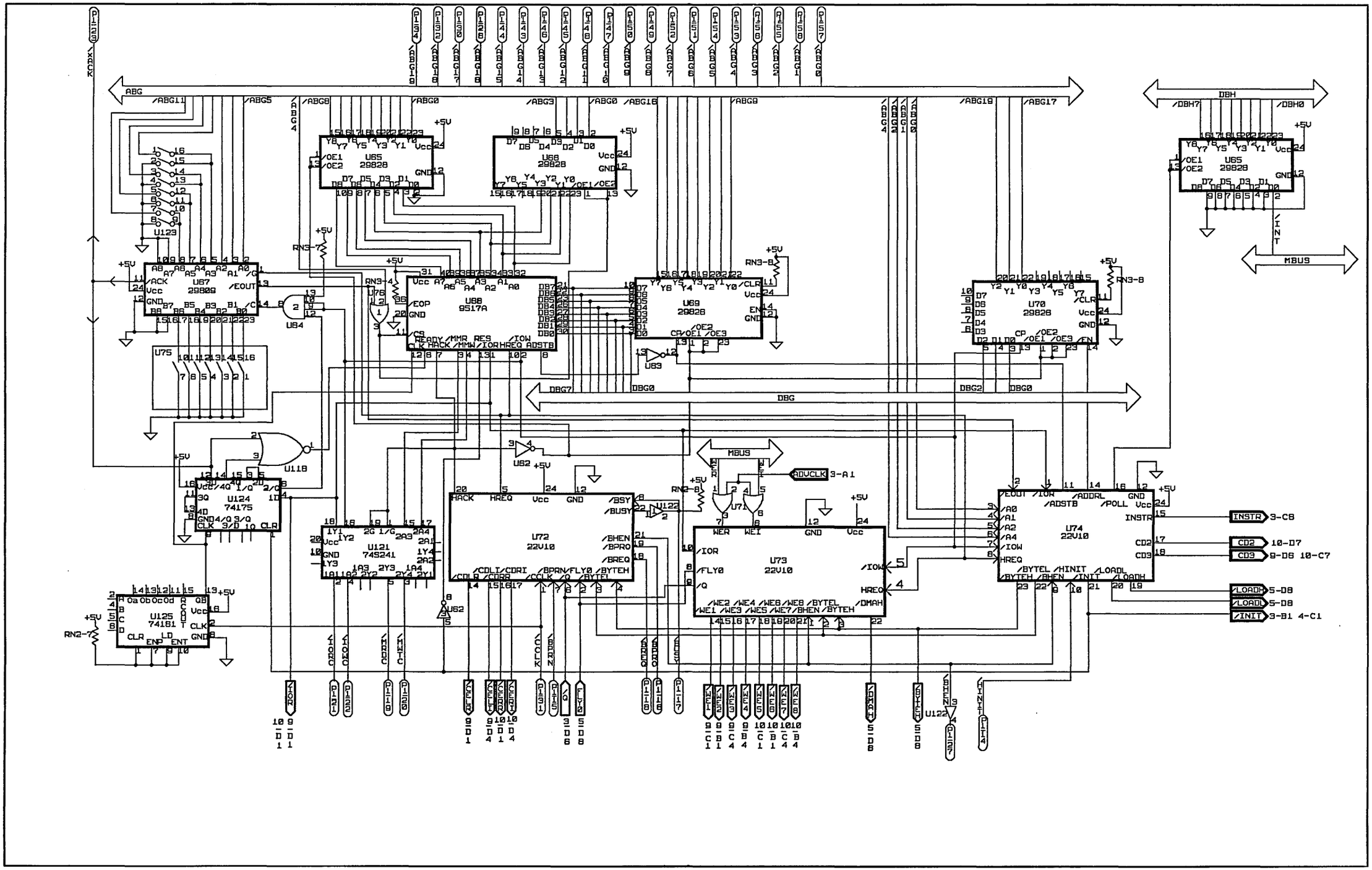
```
W0:       EQU       5D#0%    ; Coefficient select
W1:       EQU       5D#1%
W2:       EQU       5D#2%
W3:       EQU       5D#3%
W4:       EQU       5D#4%
W5:       EQU       5D#5%
W6:       EQU       5D#6%
W7:       EQU       5D#7%
W8:       EQU       5D#8%
W9:       EQU       5D#9%
W10:      EQU       5D#10%
W11:      EQU       5D#11%
W12:      EQU       5D#12%
W13:      EQU       5D#13%
W14:      EQU       5D#14%
W15:      EQU       5D#15%
W16:      EQU       5D#16%
W17:      EQU       5D#17%
W18:      EQU       5D#18%
W19:      EQU       5D#19%
W20:      EQU       5D#20%
W21:      EQU       5D#21%
W22:      EQU       5D#22%
W23:      EQU       5D#23%
W24:      EQU       5D#24%
W25:      EQU       5D#25%
W26:      EQU       5D#26%
W27:      EQU       5D#27%
W28:      EQU       5D#28%
ZERO:     EQU       5D#31%   ; Zero coefficient for NOOP
WE:       EQU       1B#0     ; Operand RAM write enable
RND:      EQU       1B#0     ; 29510 round control
TRUNC:    EQU       RND*
ACCUM:    EQU       1B#0     ; 29510 accumulate control
PASS:     EQU       ACCUM*
DRDY:     EQU       1B#0     ; Output data ready
;
INRDY:    EQU       3Q#0     ; Test condition input T0
READY:    EQU       B#0      ; Polarity control for ready
OUTACK:   EQU       3Q#1     ; Test condition input T1
TAKEN:    EQU       B#0      ; Polarity control for taken
;
;
CONTINUE:           DEF       6H#2D:,10D#0%,          ; Continue
/         1V,1V,1V,4V,1V,5V,1V,1V,1V
LD.CREG:            DEF       6H#24:,1B#1,3Q#6,6V%:,  ; Load counter
/         1V,1V,1V,4V,1V,5V,1V,1V,1V
LOOP:               DEF       6H#28:,4X,6V%:,         ; Go to pipeline if C<>0
/         1V,1V,1V,4V,1V,5V,1V,1V,1V
WAIT.TILL:          DEF       6H#3A:,1V,3V%:,6V%:,    ; Wait for test input
/         1V,1V,1V,4V,1V,5V,1V,1V,1V
;
END
```

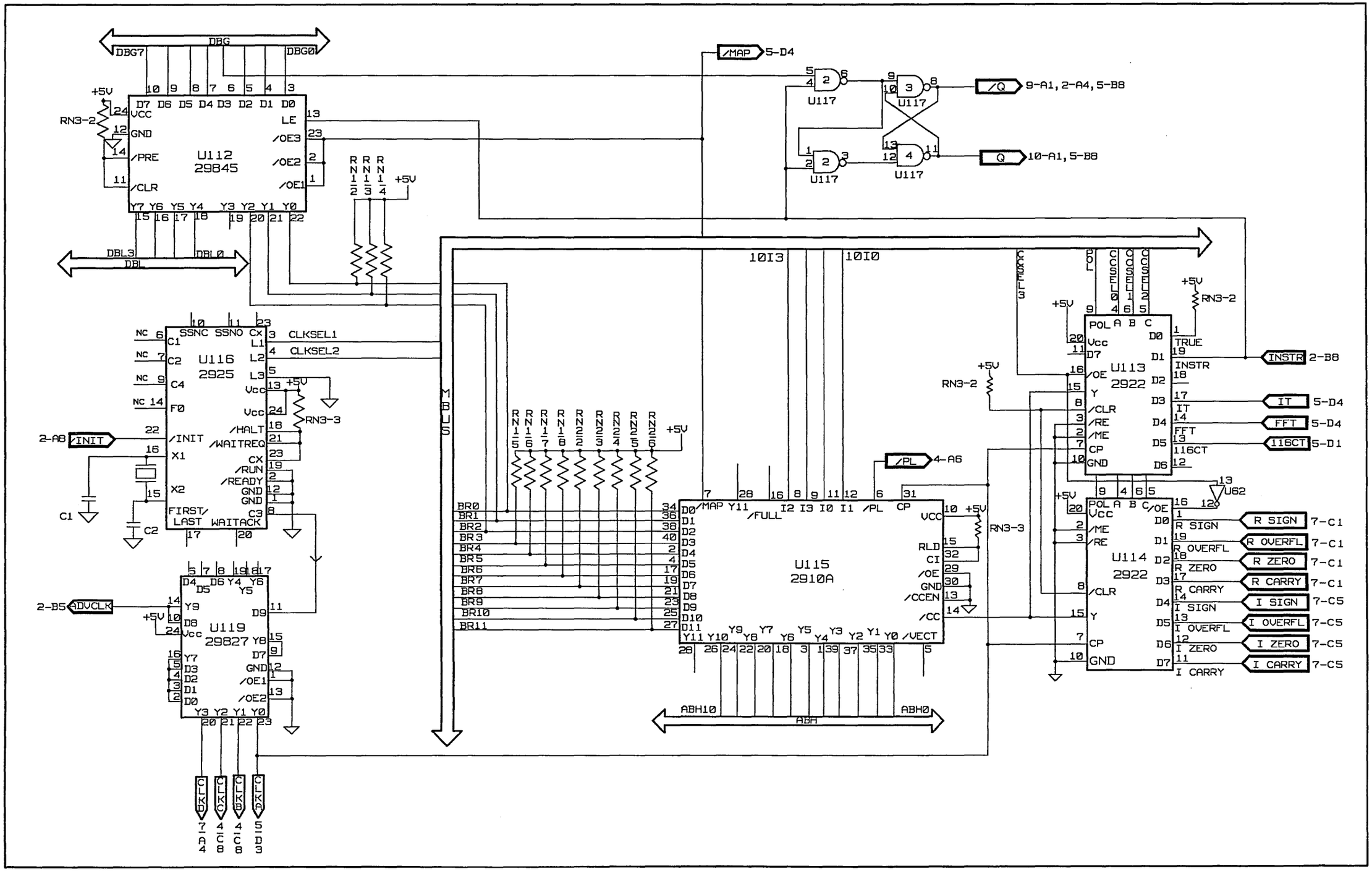
```
TITLE IIR ORDER-15 FILTER USING THE Am29510 AND THE Am29PL141
;
; This program implements the equations:
; z(n) = W(0)*x(n) + W(1)*z(n-1) + ... + W(14)*x(n-14)
; y(n) = z(n) + W(15)*z(n-1) + ... + W(28)*z(n-14)
; The CC input is grounded for unconditional jumps.
; T0 is connected to input data ready
; T1 is connected to output data taken
;
TRUE:   EQU     B#1
;
; Make previous operands = 0
INIT:
LD.CREG D#15,   DTACK,INSEL*,MACSEL*,Z0,WE,W0,PASS,TRUNC,DRDY*
LOOP $,         DTACK,INSEL*,MACSEL*,Z0,WE,W0,PASS,TRUNC,DRDY*
;
FOREVER:
WAIT.TILL READY,INRDY,$+1,      ; Wait for data to start
/               DTACK*,INSEL,MACSEL*,XN,WE,W0,PASS,TRUNC,DRDY*
; Acknowledge input, do W0*x(n)
CONTINUE        DTACK,INSEL*,MACSEL,XN,WE*,W0,PASS,TRUNC,DRDY*
; Add W1*z(n-1)
CONTINUE        DTACK*,INSEL*,MACSEL,Z1,WE*,W1,ACCUM,TRUNC,DRDY*
; Add W2*z(n-2)
CONTINUE        DTACK*,INSEL*,MACSEL,Z2,WE*,W2,ACCUM,TRUNC,DRDY*
; Add W3*z(n-3)
CONTINUE        DTACK*,INSEL*,MACSEL,Z3,WE*,W3,ACCUM,TRUNC,DRDY*
; Add W4*z(n-4)
CONTINUE        DTACK*,INSEL*,MACSEL,Z4,WE*,W4,ACCUM,TRUNC,DRDY*
; Add W5*z(n-5)
CONTINUE        DTACK*,INSEL*,MACSEL,Z5,WE*,W5,ACCUM,TRUNC,DRDY*
; Add W6*z(n-6)
CONTINUE        DTACK*,INSEL*,MACSEL,Z6,WE*,W6,ACCUM,TRUNC,DRDY*
; Add W7*z(n-7)
CONTINUE        DTACK*,INSEL*,MACSEL,Z7,WE*,W7,ACCUM,TRUNC,DRDY*
; Add W8*z(n-8)
CONTINUE        DTACK*,INSEL*,MACSEL,Z8,WE*,W8,ACCUM,TRUNC,DRDY*
; Add W9*z(n-9)
CONTINUE        DTACK*,INSEL*,MACSEL,Z9,WE*,W9,ACCUM,TRUNC,DRDY*
; Add W10*z(n-10)
CONTINUE        DTACK*,INSEL*,MACSEL,Z10,WE*,W10,ACCUM,TRUNC,DRDY*
; Add W11*z(n-11)
CONTINUE        DTACK*,INSEL*,MACSEL,Z11,WE*,W11,ACCUM,TRUNC,DRDY*
; Add W12*z(n-12)
CONTINUE        DTACK*,INSEL*,MACSEL,Z12,WE*,W12,ACCUM,TRUNC,DRDY*
; Add W13*z(n-13)
CONTINUE        DTACK*,INSEL*,MACSEL,Z13,WE*,W13,ACCUM,TRUNC,DRDY*
; Add W14*z(n-14)
CONTINUE        DTACK*,INSEL*,MACSEL,Z14,WE*,W14,ACCUM,TRUNC,DRDY*
;
```

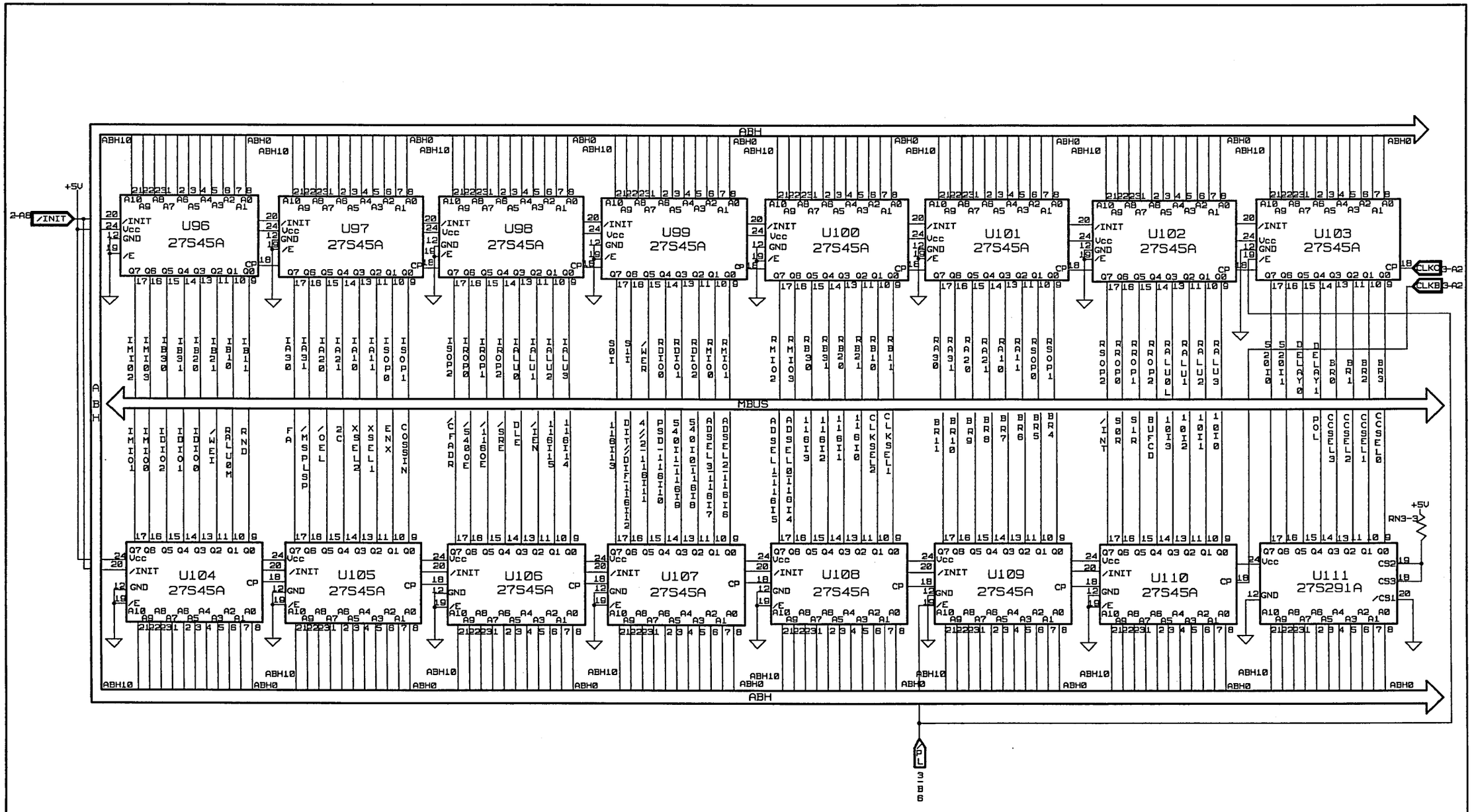
```
;
; Save intermediate result
CONTINUE          DTACK*,INSEL*,MACSEL,Z0,WE*,ZERO,ACCUM,TRUNC,DRDY*
CONTINUE          DTACK*,INSEL*,MACSEL,Z0,WE,ZERO,ACCUM,TRUNC,DRDY*
; Add W15*z(n-1)
CONTINUE          DTACK*,INSEL*,MACSEL,Z1,WE*,W15,ACCUM,TRUNC,DRDY*
; Add W16*z(n-2)
CONTINUE          DTACK*,INSEL*,MACSEL,Z2,WE*,W16,ACCUM,TRUNC,DRDY*
; Add W17*z(n-3)
CONTINUE          DTACK*,INSEL*,MACSEL,Z3,WE*,W17,ACCUM,TRUNC,DRDY*
; Add W18*z(n-4)
CONTINUE          DTACK*,INSEL*,MACSEL,Z4,WE*,W18,ACCUM,TRUNC,DRDY*
; Add W19*z(n-5)
CONTINUE          DTACK*,INSEL*,MACSEL,Z5,WE*,W19,ACCUM,TRUNC,DRDY*
; Add W20*z(n-6)
CONTINUE          DTACK*,INSEL*,MACSEL,Z6,WE*,W20,ACCUM,TRUNC,DRDY*
; Add W21*z(n-7)
CONTINUE          DTACK*,INSEL*,MACSEL,Z7,WE*,W21,ACCUM,TRUNC,DRDY*
; Add W22*z(n-8)
CONTINUE          DTACK*,INSEL*,MACSEL,Z8,WE*,W22,ACCUM,TRUNC,DRDY*
; Add W23*z(n-9)
CONTINUE          DTACK*,INSEL*,MACSEL,Z9,WE*,W23,ACCUM,TRUNC,DRDY*
; Add W24*z(n-10)
CONTINUE          DTACK*,INSEL*,MACSEL,Z10,WE*,W24,ACCUM,TRUNC,DRDY*
; Add W25*z(n-11)
CONTINUE          DTACK*,INSEL*,MACSEL,Z11,WE*,W25,ACCUM,TRUNC,DRDY*
; Add W26*z(n-12)
CONTINUE          DTACK*,INSEL*,MACSEL,Z12,WE*,W26,ACCUM,TRUNC,DRDY*
; Add W27*z(n-13)
CONTINUE          DTACK*,INSEL*,MACSEL,Z13,WE*,W27,ACCUM,TRUNC,DRDY*
; Add W28*z(n-14)
CONTINUE          DTACK*,INSEL*,MACSEL,Z14,WE*,W28,ACCUM,RND,DRDY*
; Wait for pipeline delay
CONTINUE          DTACK*,INSEL*,MACSEL,XN,WE*,ZERO,ACCUM,TRUNC,DRDY*
; Wait for data taken, then loop back for next sample
WAIT.TILL TAKEN,OUTACK,FOREVER
/                 DTACK*,INSEL*,MACSEL,XN,WE*,ZERO,ACCUM,TRUNC,DRDY
;
END
```

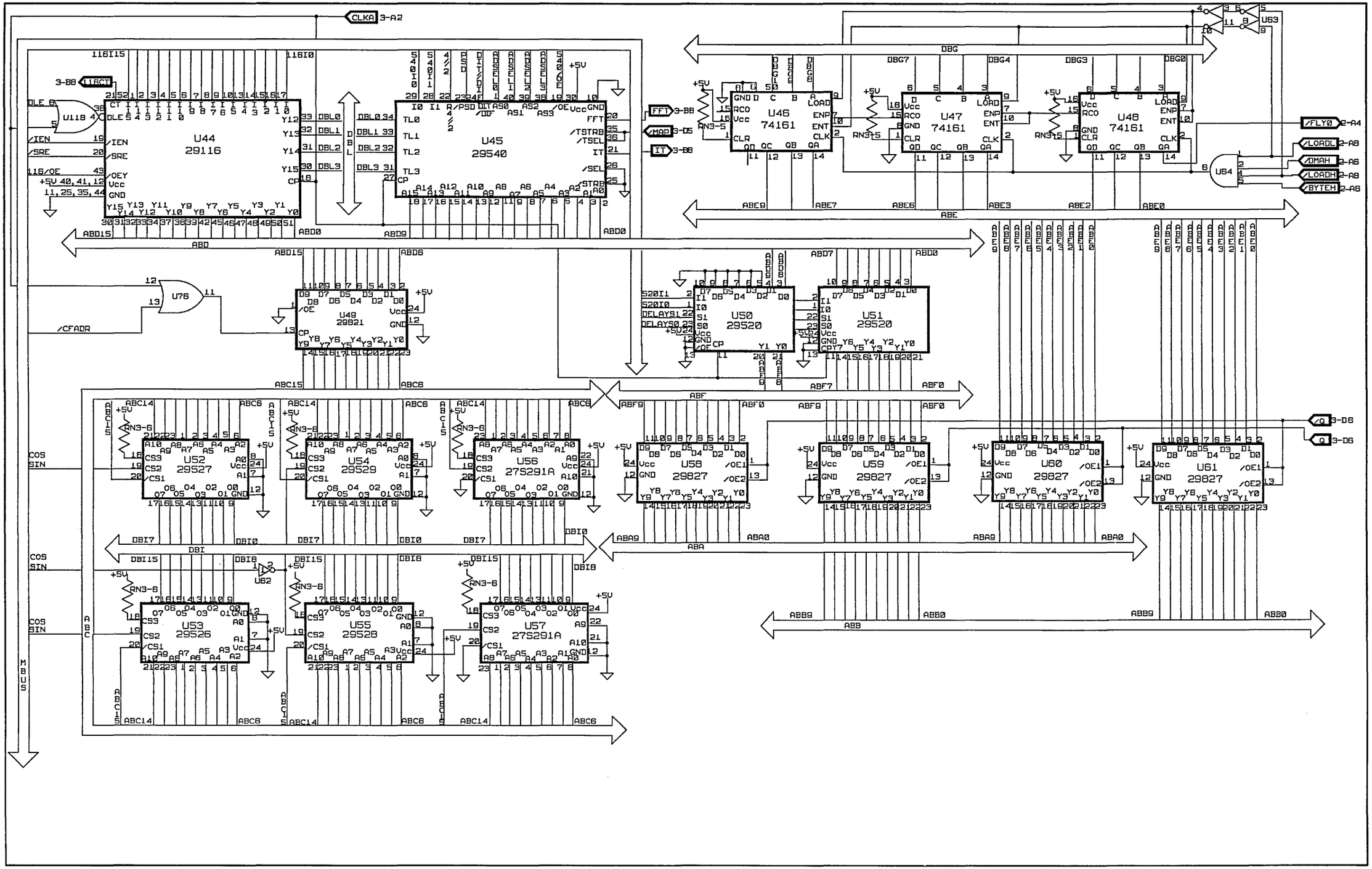

MBUS
DBM
DBM15
DBM8
DBM7
DBM0
DBN
DBN15
DBN8
DBN7
DBN0
U89
29501
U90
29501
U91
29501
U92
29501
U120
2902
U93
2947
U94
2947
U95
29517
RMCIN
RMG
RMP
RCARRY
R ZERO
ROVERFL
R SIGN
ILCIN
ILG
ILP
ICARRY
I ZERO
IOVERFL
ISIGN
DBP
DBJ
DBO
DBK
DBXFC
MIO7
MIO6
MIO5
MIO4
MIO3
MIO2
MIO1
MIO0
RMIO3
RMIO2
RMIO1
RMIO0
BUFCD
RND
FA
/MSPLSP
/ENX
2C
/OEP
/ENY
/ENP
/OEL

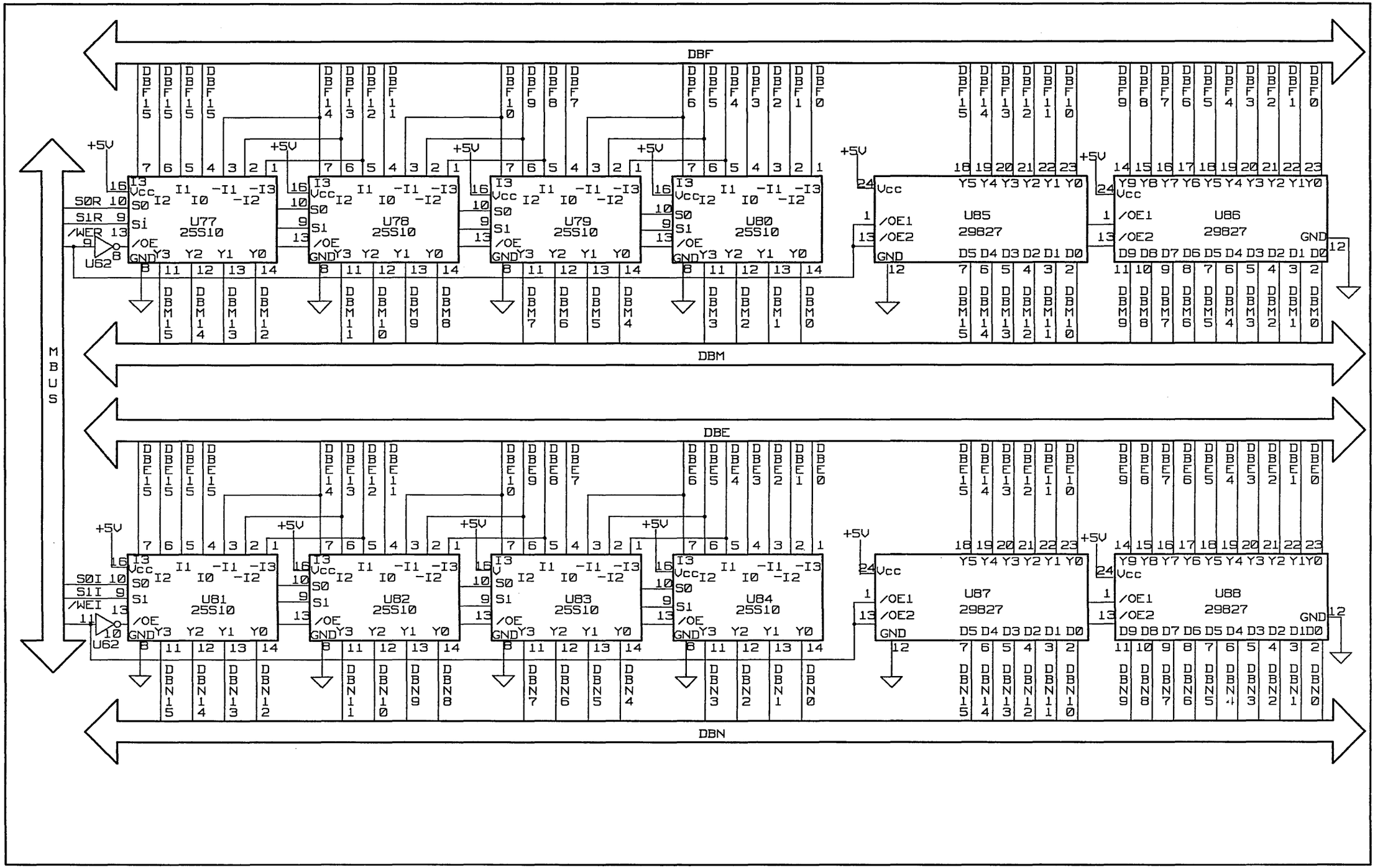

ADDRESS BUS
DATA BUS
A0-2
A1-8
A9-16
A17-19
A5-11
29828 (2) /OE
29826
29826
29809
/EOUT
2946 T//R
2946 T//R
2946 T//R
/IOR
HREQ 9517
D0-7
D0-2
1/2 PAL 22V10 ARBITER LOGIC
CONTROL BUS
/C DELAY CIRCUIT
/IOWC
/IORC
PAL 22V10
/INIT
/INSTR
/LOADL
/LOADH
CD3
CD2
29827
Q
74161 (3)
D0-9
29827
LEFT ADDR.
/Q
/INIT
27S45A CS
29827
RIGHT ADDR.
/Q
29845 IR /OE
2922 (2) CC MUX
/CCEN
2910A
/MAP
/CCC
29827
Q
MEMORY SELECT LOGIC
Q
/Q
FFT COEFF. PROMS 29526-9
29540
29520 (2)
29116
29821 (2)
FILTER COEFF. PROM 27S291
2947 /CS (2) T//R
2947 /CS (2) T//R
2947 /CS (2) T//R
2947 /CS (2) T//R
1/2 PAL 22V10
/BYTEL
/BYTEH
/FLY0
HREQ
/Q
MEMORY REAL
MEMORY IMAG.
MEMORY REAL
MEMORY IMAG.
2148-45 (4)
2148-45 (4)
2148-45 (4)
2148-45 (4)
WEI PAL WE8 22V10
/BHEN
/BYTEL
/BYTEH
/IOR
/IOW
WEI
/FLY0
HREQ
WER
/Q
2947 T//R (2) /CS
/WER
2947 T//R (2) /CS
/WEI
Q
2947 T//R (2) /CS
/WER
2947 T//R (2) /CS
/WEI
/Q
DSP IMAG. DATA BUS
DSP REAL DATA BUS
29827 (2)
25S10 (4) SCALER
29827 (2)
25S10 (4) SCALER
REAL ALU 29501 (2)
IMAGINARY ALU 29501 (2)
74LS253 (8) I OF 4 MUX
29827 (2)
MULTIPLIER 29517

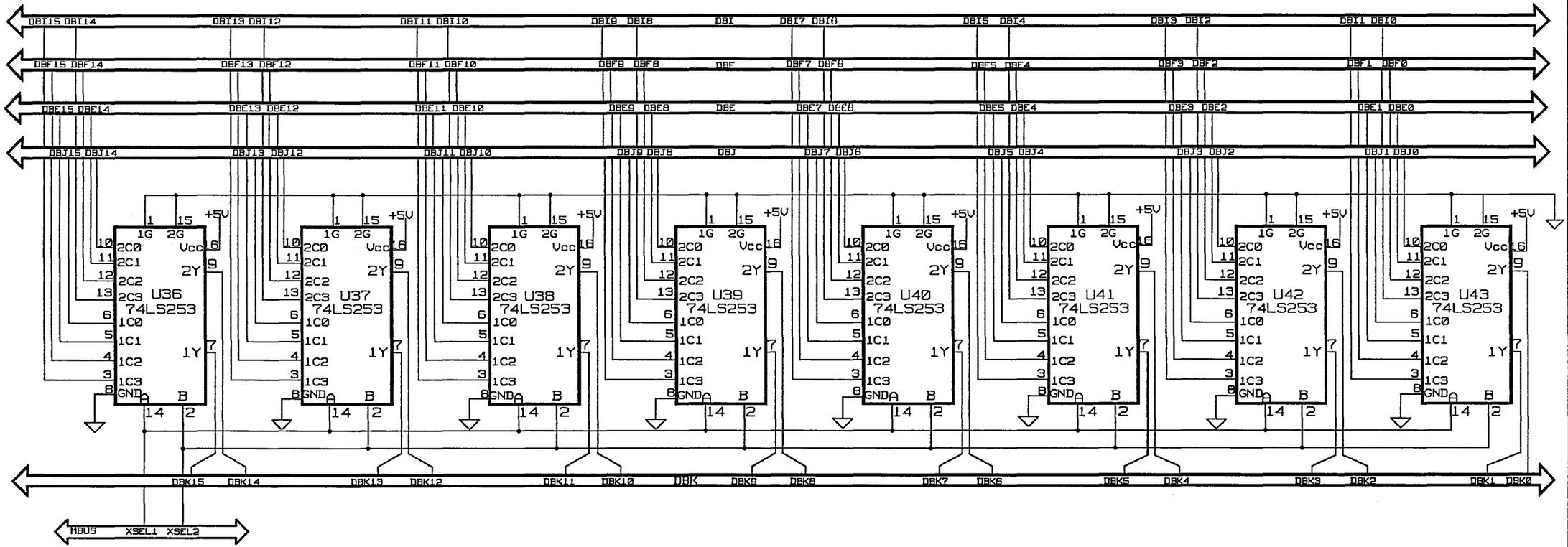

ABG
DBG
DBH
MBUS
U65 29828
U68 29828
U67 29809
U84
U88 9517A
U69 29828
U70 29828
U83
U75
U118
U124 74175
U82
U121 74S241
U72 22V10
U122
U71
U73 22V10
U74 22V10
U125 74181
U62
INSTR 3-C8
CD2 10-D7
CD3 9-D6 10-C7
LOADH 5-D8
LOADL 5-D8
INIT 3-B1 4-C1
ADVCLK 3-A1

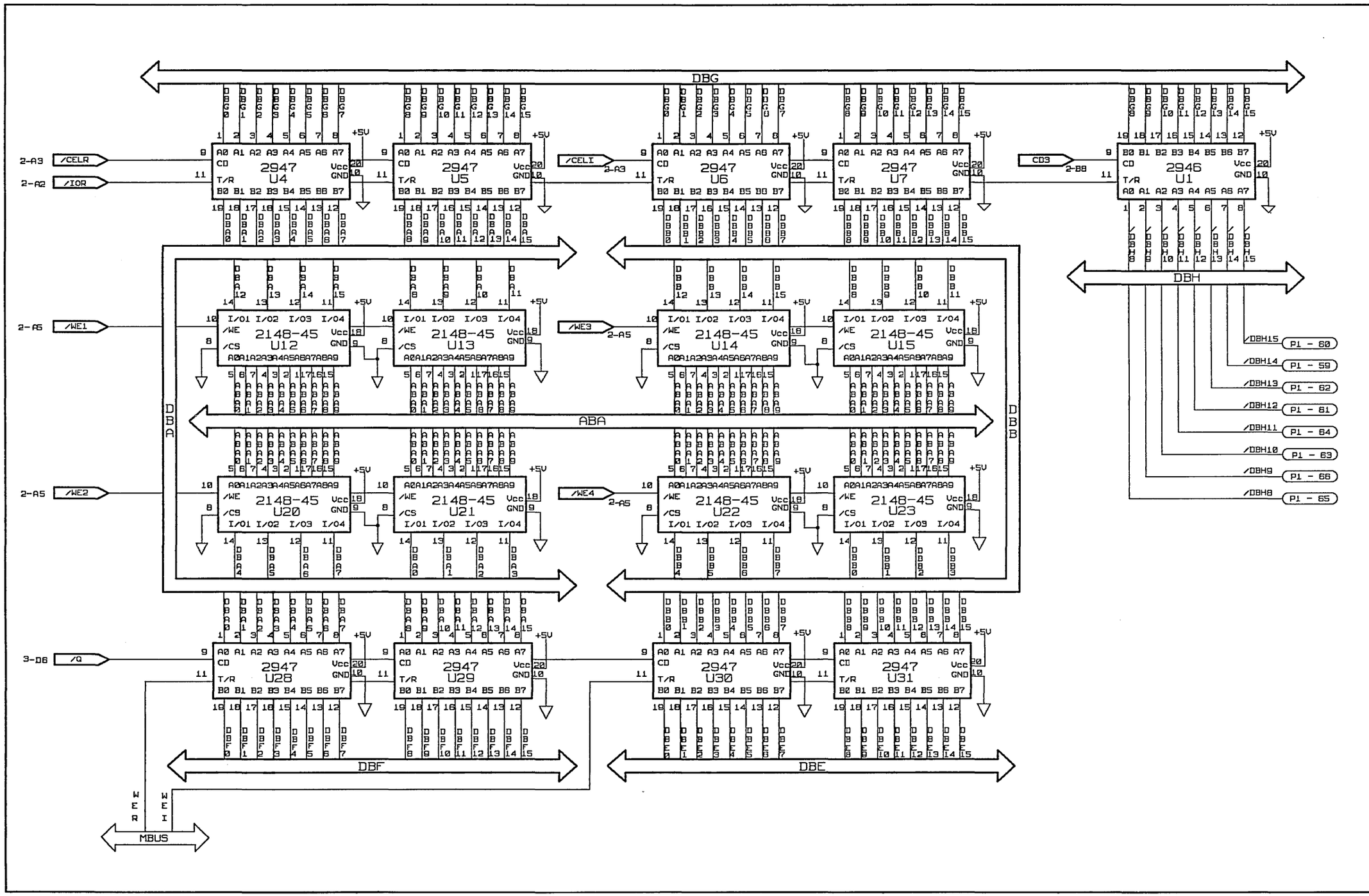

DBG
DBG7
DBG0
DBL
DBL3
DBL0
MBUS
/MAP 5-D4
/Q 9-A1, 2-A4, 5-B8
Q 10-A1, 5-B8
U117
U112
29845
U116
2925
U119
29827
U115
2910A
U113
2922
U114
2922
U62
RN3-2
RN3-3
+5V
C1
C2
2-A8 /INIT
2-B5 ADVCLK
CLKSEL1
CLKSEL2
/PL 4-A6
10I3
10I0
INSTR 2-B8
IT 5-D4
FFT 5-D4
116CT 5-D1
R SIGN 7-C1
R OVERFL 7-C1
R ZERO 7-C1
R CARRY 7-C1
I SIGN 7-C5
I OVERFL 7-C5
I ZERO 7-C5
I CARRY 7-C5
ABH10
ABH0
ABH

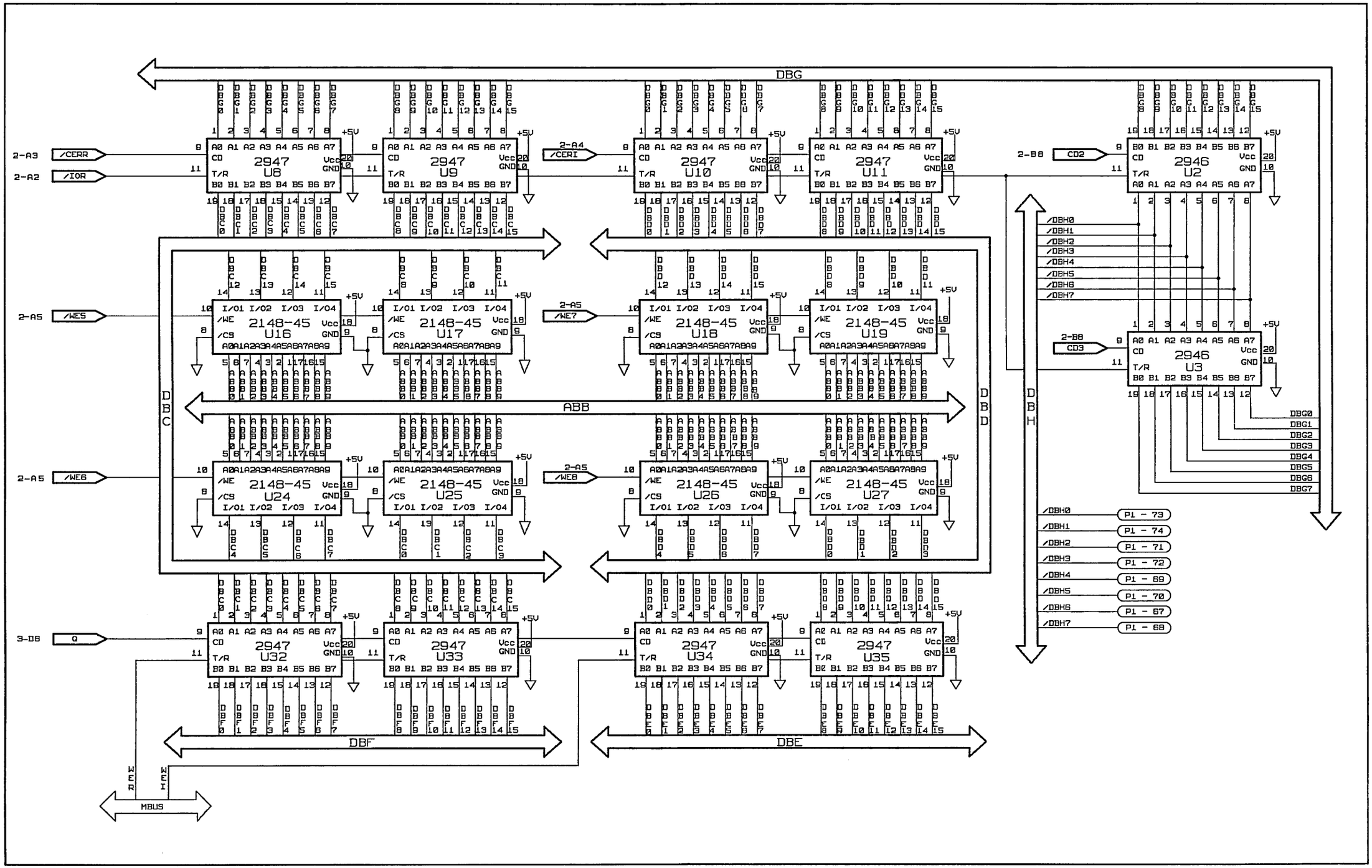

ABH
MBUS
U96 27S45A
U97 27S45A
U98 27S45A
U99 27S45A
U100 27S45A
U101 27S45A
U102 27S45A
U103 27S45A
U104 27S45A
U105 27S45A
U106 27S45A
U107 27S45A
U108 27S45A
U109 27S45A
U110 27S45A
U111 27S291A
/INIT
+5V
CLKA
CLKB
RN3-3


CLKA 3-A2
U44 29116
U45 29540
U46 74161
U47 74161
U48 74161
U63
U64
U118
U76
U49 29821
U50 29520
U51 29520
U52 29527
U53 29526
U54 29529
U55 29528
U56 27S291A
U57 27S291A
U58 29827
U59 29827
U60 29827
U61 29827
U62
/FLY0 2-A4
/LOADL 2-A8
/DMAH 2-A6
/LOADH 2-A8
/BYTEH 2-A8
/Q 3-D6
Q 3-D6
FFT 3-B8
/MAP 3-D5
IT 3-B8
/CFADR
COS
SIN
MBUS
DBG
ABE
ABD
ABC
DBI
ABA
ABB
ABF


DBF
DBM
DBE
DBN
MBUS
+5V
U77
25S10
U78
25S10
U79
25S10
U80
25S10
U85
29827
U86
29827
U81
25S10
U82
25S10
U83
25S10
U84
25S10
U87
29827
U88
29827
U62
S0R
S1R
/WER
S0I
S1I
/WEI
Si
/OE
/OE1
/OE2
Vcc
GND


DBI15 DBI14
DBI13 DBI12
DBI11 DBI10
DBI9 DBI8
DBI
DBI7 DBI6
DBI5 DBI4
DBI3 DBI2
DBI1 DBI0
DBF15 DBF14
DBF13 DBF12
DBF11 DBF10
DBF9 DBF8
DBF
DBF7 DBF6
DBF5 DBF4
DBF3 DBF2
DBF1 DBF0
DBE15 DBE14
DBE13 DBE12
DBE11 DBE10
DBE9 DBE8
DBE
DBE7 DBE6
DBE5 DBE4
DBE3 DBE2
DBE1 DBE0
DBJ15 DBJ14
DBJ13 DBJ12
DBJ11 DBJ10
DBJ9 DBJ8
DBJ
DBJ7 DBJ6
DBJ5 DBJ4
DBJ3 DBJ2
DBJ1 DBJ0
U36
U37
U38
U39
U40
U41
U42
U43
74LS253
1G
2G
2C0
2C1
2C2
2C3
1C0
1C1
1C2
1C3
1Y
2Y
Vcc
GND
A
B
+5V
DBK15 DBK14
DBK13 DBK12
DBK11 DBK10
DBK
DBK9 DBK8
DBK7 DBK6
DBK5 DBK4
DBK3 DBK2
DBK1 DBK0
MBUS
XSEL1
XSEL2


DBG
DBH
DBA
DBB
DBF
DBE
ABA
MBUS
WEI
WER
2947
2946
2148-45
U4
U5
U6
U7
U1
U12
U13
U14
U15
U20
U21
U22
U23
U28
U29
U30
U31
/CELR
/IOR
/CELI
/WE1
/WE3
/WE2
/WE4
/Q
CD3
/DBH15
/DBH14
/DBH13
/DBH12
/DBH11
/DBH10
/DBH9
/DBH8
P1 - 60
P1 - 59
P1 - 62
P1 - 61
P1 - 64
P1 - 63
P1 - 66
P1 - 65
+5V


DBG
2-A3
/CERR
2-A2
/IOR
2-A4
/CERI
2-B8
CD2
U8
U9
U10
U11
U2
2947
2946
2-A5
/WE5
2-A5
/WE7
2-A5
/WE6
2-A5
/WE8
2-B8
CD3
U16
U17
U18
U19
U3
2148-45
ABB
DBC
DBD
DBH
U24
U25
U26
U27
3-D6
Q
U32
U33
U34
U35
DBF
DBE
WEA
WEI
MBUS
/DBH0
/DBH1
/DBH2
/DBH3
/DBH4
/DBH5
/DBH6
/DBH7
DBG0
DBG1
DBG2
DBG3
DBG4
DBG5
DBG6
DBG7
P1 - 73
P1 - 74
P1 - 71
P1 - 72
P1 - 69
P1 - 70
P1 - 67
P1 - 68

ADVANCED MICRO DEVICES, INC.
901 Thompson Place
P.O. Box 3453
Sunnyvale,
California 94088
(408) 732-2400
TWX: 910-339-9280
TELEX: 34-6306
TOLL FREE
(800) 538-8450

Order #04779A

Printed in U.S.A. IH-MU-10M-1/86-0