

Transform Processing on a Reconfigurable Data Path Processor ¹

K. Joe Hass David F. Cox
jhass@mrc.unm.edu *dcox@mrc.unm.edu*
NASA Institute of Advanced Microelectronics
Microelectronics Research Center
University of New Mexico
801 University Blvd. SE, Suite 206
Albuquerque, New Mexico 87106

Abstract – A Reconfigurable Data Path Processor is capable of high performance transform processing. The Fast Fourier Transform is used to investigate how the RDPP architecture influences its ability to execute these algorithms. Directed graphs are used to visualize the algorithm, which is then implemented in the unique programming language of the RDPP.

1 Introduction

Previous efforts resulted in the development of a reconfigurable data path processor, consisting of a number of identical data path elements interconnected with a wide crossbar bus. Each data path element is capable of performing simple logic functions as well as a multiply-accumulate (MAC) step, making the data path processor well suited to digital signal processing applications. This paper describes the implementation of transform algorithms, such as the Fast Fourier Transform (FFT) on the reconfigurable processor. Conventional software descriptions of these algorithms are converted to directed graphs in order to visualize and explore the flow of data. These graphs then suggest possible configurations of the reconfigurable processor. A behavioral VHDL model of the processor is used to evaluate architectural tradeoffs, while gate level models of a data path element were created to examine design issues at a lower level. These models have suggested alternate configurations of the processor that are better suited to transform algorithms. A comparison of these architectures is presented, along with estimates of processing performance for several algorithms of interest.

2 The Fast Fourier Transform

The Fourier Transform is used to convert some function of time, $h(t)$, to its equivalent representation, $H(f)$, in the frequency domain. In practice, the transform is typically performed on a set of data points, h_n , obtained by sampling $h(t)$ at equally spaced time intervals. The

¹This research was supported by NASA under Space Engineering Research Grant NAGW-3293.

Discrete Fourier Transform, or DFT, is used in this case:

$$H_n \equiv \sum_{k=0}^{N-1} h_k e^{j2\pi kn/N} \text{ where } N \text{ is the number of samples}$$

The DFT transforms N complex numbers (the sampled data points) into N complex numbers that represent the same data in the frequency domain. Each H_n computation requires the summation of N product terms, so the complete transform requires $O(N^2)$ operations. For large sample sizes this brute-force calculation is not practical.

More efficient methods for calculating the DFT have been investigated for nearly 200 years, starting with Gauss in 1805. In 1942, Danielson and Lanczos showed that a DFT of length N can be performed by adding the results of two DFTs of length $N/2$, where one of the DFTs uses the even numbered elements from the original data set and the other DFT uses the odd numbered elements. This can be repeated, dividing the original data set into smaller and smaller pieces until the DFT of only two samples is performed. This concept was popularized in the context of electronic computers in 1965 by J. W. Cooley and J. W. Tukey and is commonly known as the *Cooley-Tukey Fast Fourier Transform*.

The basic building block of the FFT is the *butterfly* operator, shown in Figure 1. It performs these two computations:

$$H(0) = h(0) + h(1) \quad (1)$$

$$H(1) = W^k(h(0) - h(1)) \quad (2)$$

Since all of the input and output values are complex numbers, this butterfly actually requires ten real operations: three additions, three subtractions and four multiplications. The W^k ‘twiddle factors’ are themselves complex numbers of the form

$$W^k = e^{-j2\pi k/N} = \cos(2\pi k/N) - j \sin(2\pi k/N)$$

The butterfly is usually drawn without explicitly showing the addition or subtraction, and it is understood that whenever two or more arrows meet some addition or subtraction is implied.

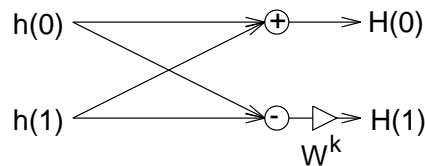


Figure 1: Radix-2 FFT Butterfly

The butterfly operator can be used to compute the FFT of any set of data points where N is of the form $N = 2^m$. Figure 2 illustrates this process for $N = 8$. In the first stage, the butterfly operates on data points $h(0)$ and $h(4)$, then on points $h(1)$ and $h(5)$, and so on until the data is exhausted. The second stage processes the results from the first stage, now working with data points that are only two locations apart. Finally, adjacent data elements are combined in the third stage to produce the transform outputs, $H(n)$.

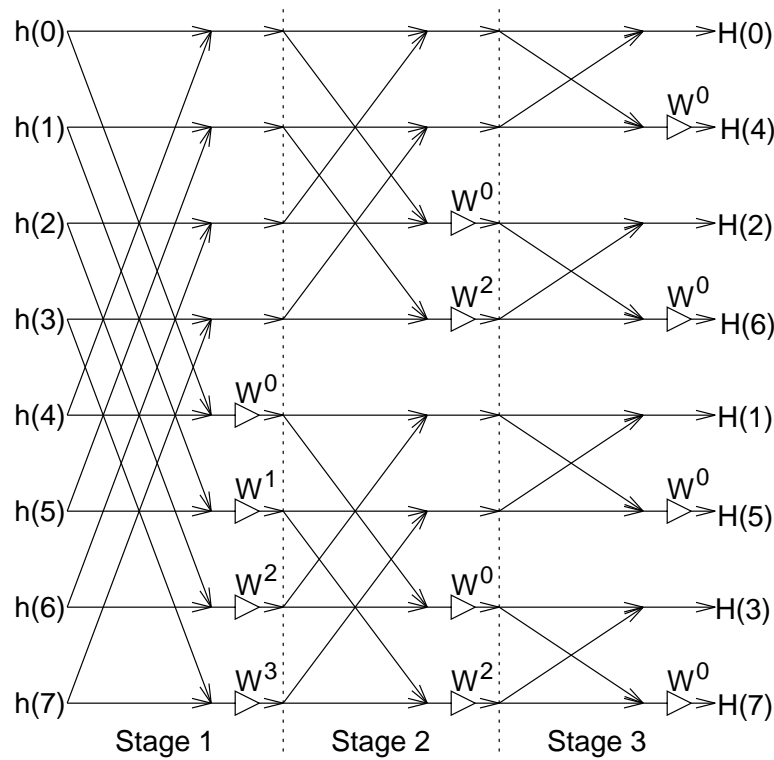


Figure 2: FFT for 8 data points

The butterfly operator discussed above is called a radix-2 butterfly because it processes 2 data elements in parallel. Butterfly operators for higher radices can be derived, such as the radix-4 butterfly shown in Figure 3. The radix-4 butterfly replaces four radix-2 butterflies, reducing the number of multiplies by 25%. However, each transform result is based on a sum and difference term derived from all four inputs so the number of additions is only reduced by 8.3%. Of particular significance for hardware FFT processors is the fact that an FFT based on the radix-4 butterfly needs only $m = \log_4 N$ stages which can reduce the processor clock rate and data memory bandwidth requirements. Radix-8 and radix-16 butterflies have also been used, although they are more complicated and require a multiplication inside the butterfly as well as the twiddle factor scaling at the outputs.

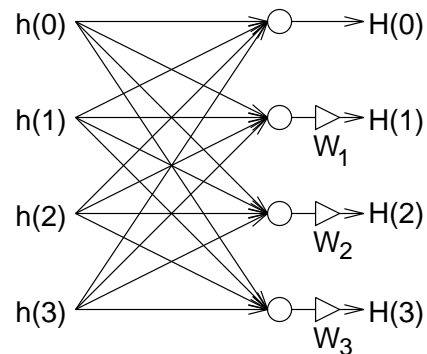


Figure 3: Radix-4 FFT Butterfly

Butterflies with different radices can be combined for any given value of N . For example, a 1024-point FFT can be performed using two passes with a radix-16 butterfly followed by a single pass with a radix-4 butterfly because $1024 = 16 \times 16 \times 4$. If the data rate for this FFT was 1×10^6 samples per second, then the bandwidth for the data into the FFT processor must be three times as high, or 3×10^6 complex data words per second, in order to process data sets as fast as they arrive. A processor that had only a radix-2 butterfly would need ten passes, with a data bandwidth of 10×10^6 complex words per second, to process this data in real time.

3 The FFT in a Programmable Parallel Processor

The butterfly operator represents a much different computational problem than typical FIR and IIR digital filters. The FFT algorithm is not a simple recursive multiply-and-accumulate process, it processes data in relatively small blocks, and the fact that all data values are complex numbers doesn't make matters any easier. However, there is a great deal of parallelism that can be exploited if appropriate hardware resources are available. This can be seen in the directed graph of the radix-4 algorithm shown in Figure 4, which was adapted from a FORTRAN program. The graph can be used to visualize how this algorithm could be mapped into a highly parallel hardware implementation.

The column of triangles in the center of the graph represent the data inputs. Input values $x(0)$ through $x(3)$ are the real portion of the data points while $y(0)$ through $y(3)$ are the imaginary portion. In other words, $h(n) = x(n) + j y(n)$. Similarly, $\cos(a)$ through $\cos(c)$ are the real portion of the complex twiddle factors while $\sin(a)$ through $\sin(c)$ are the imaginary portion. Ellipses represent an intermediate computation and rectangles represent a computation that produces an output value. Processing proceeds from the top of the graph to the bottom, with each row corresponding to a sequential time step in the algorithm.

A reconfigurable data path architecture that may be well suited to FFT algorithms has been described. This architecture was later modified slightly to improve the utilization of its data path elements. Basically, the data path consists of a number of data path elements (DPE) interconnected by a global bus. Each DPE has a single output port that drives a portion of the global bus and a set of input ports that can read the output of any other DPE. If there are n DPEs and the width of their output ports is m bits, then the global bus width is $n \times m$ bits.

A simplified block diagram for a single DPE is shown in Figure 5. The global bus enters the DPE at the top of the diagram and feeds four $m:1$ multiplexers (labeled MUXA through MUXD). Each multiplexer has $n \times m$ inputs and m outputs. Two of the multiplexers are connected to data storage registers (DREG1 and DREG2) that can hold data values for later use. Two more multiplexers (SEL) select two of the data inputs for the multiplier/accumulator (MAC). The other two MAC inputs come from the logic units (LOGIC), which can perform simple Boolean operations on their inputs. The MAC computes the product of two of its inputs and adds that result to the other two inputs. The output register may be loaded with the MAC output (unshifted, shifted right, or shifted left) or may hold its current value. The data path elements are designed to complete all of these operations in a single clock period.

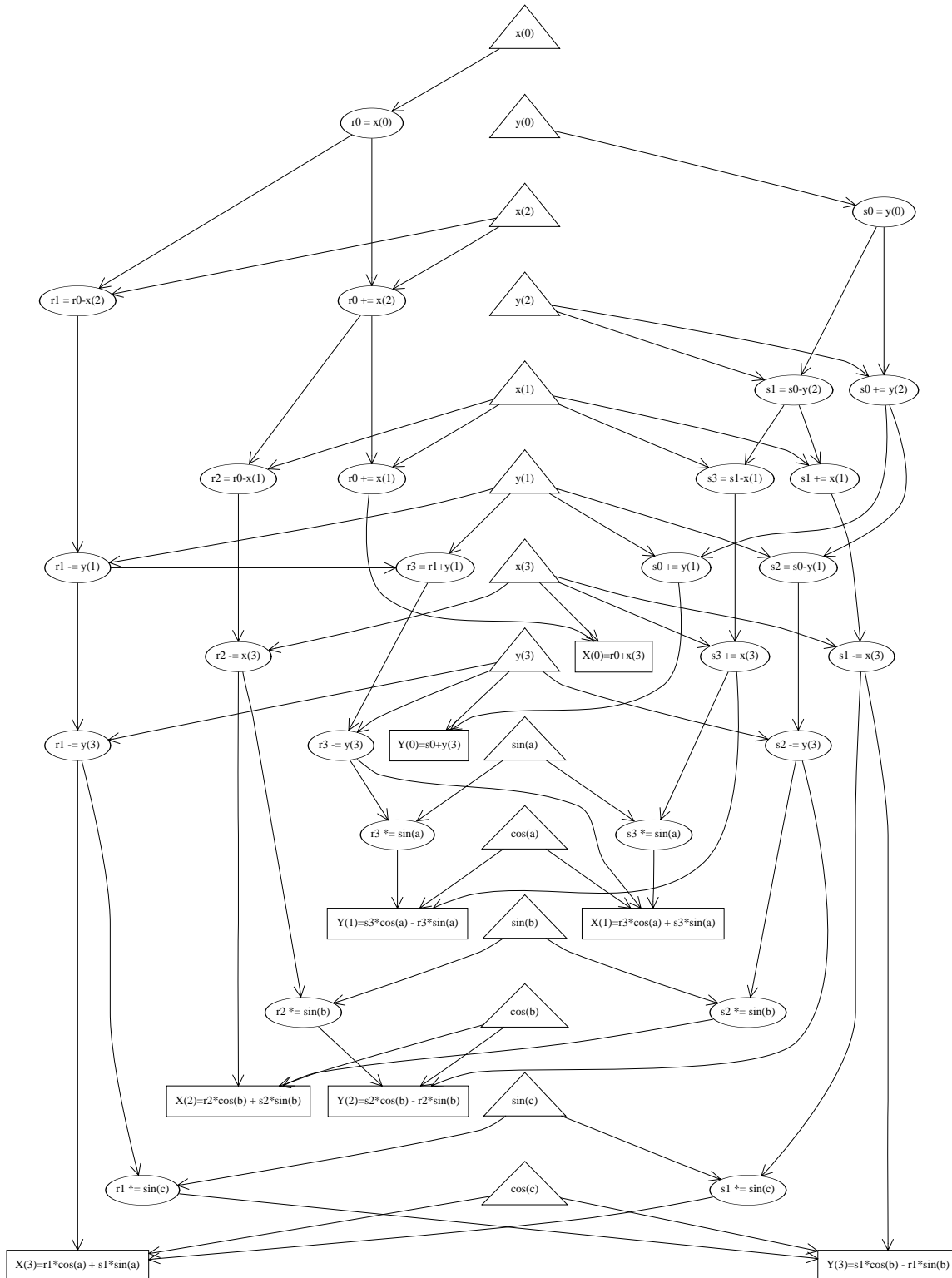


Figure 4: Radix-4 Butterfly Directed Graph

A very wide control store provides the signals that control the operation of every logic block in every DPE, allowing them to be arbitrarily ‘connected’ in parallel, serial or pipeline configurations. The control store can be implemented as on-chip RAM that is preloaded before processing begins. In general, one control store word, or instruction, is executed every clock cycle and a simple looping mechanism allows a group of instructions to be repeated indefinitely.

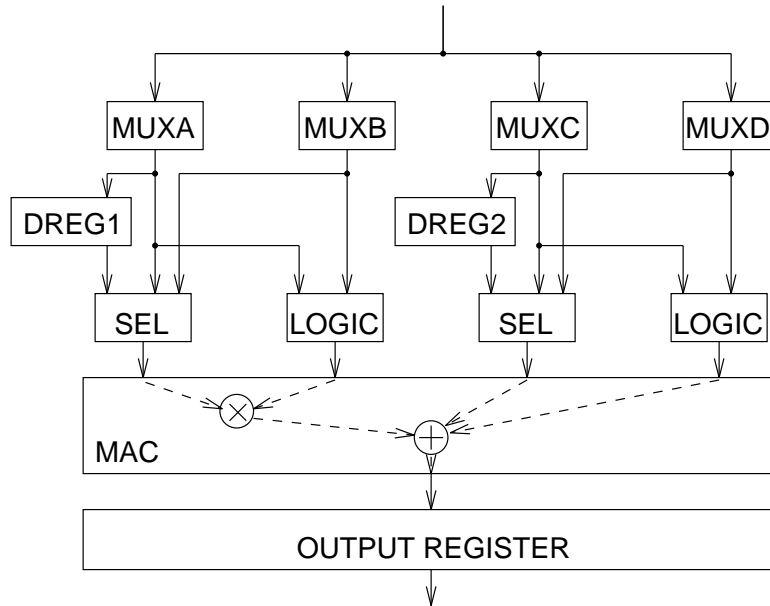


Figure 5: Reconfigurable Data Path Processor Element

In general, the FFT butterfly can be implemented with this reconfigurable data path by treating each row in the directed graph as a single clock period and replacing each ellipse and rectangle in the graph with a data path element. The edges between nodes (represented by arrows in the graph) indicate the connections that must be made through the global bus. This simple mapping can be done only when all edges are constrained to span from one row to the next and can only flow downward. The most efficient utilization of the DPEs occurs when the number of nodes in each row is more or less equal, because the number of DPEs required will be determined by the row with the largest number of nodes. Whenever an edge spans more than one row the situation becomes more complicated. In this case we must store the output result from the generating node in some data register in the receiving node, or the generating node must hold all of the necessary input data so that its output can be recalculated just before it is needed by the receiving node.

The reconfigurable data path processor has been modeled and simulated using VHDL. Figure 6 illustrates the processing flow for the radix-4 FFT butterfly that results when the directed graph is mapped into the reconfigurable processor. Each row in the diagram corresponds to a single processing step, or clock cycle, for the processor. Each column of squares indicates the activity in a single processing element. For clarity, only the eight processing elements used in this computation are shown in the diagram. A white square indicates that the processing element is not used during that time step, a gray square indicates that the

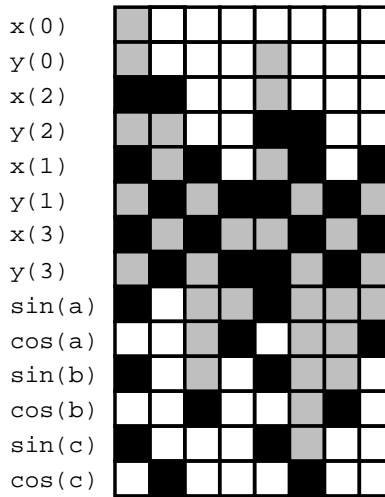


Figure 6: Processing Footprint for Radix-4 FFT Butterfly

processing element is storing data for later use, and a black square indicates that the processing element is performing an arithmetic operation. The descriptive column on the left indicates which data element is input to the processor at each step.

It is clear from this illustration that the processor resources are not used very efficiently by this particular implementation of the radix-4 algorithm. However, this program was created only as an intermediate step, while the ultimate goal was to create a program that would cascade two levels of radix-4 processing. The cascaded radix-4 program would effectively process 16 input samples as a single unit but has a more regular data flow than the radix-16 algorithm. Since the output of this program is 16 complex values it becomes necessary to accumulate 32 real values during the processing of the first stage radix-4 algorithm. These 32 values will consist of sums and differences of the output values from the first radix-4 stage. The MAC registers in the first eight processing elements are used for computations in the first radix-4 stage so they cannot be used as temporary storage for the second stage.

The DREG registers in each DPE are not used in the first radix-4 stage so they could be used to accumulate the 32 intermediate values. This also seems to be a natural choice because the 16 data path elements can contribute exactly 32 data registers. Unfortunately, the original architecture for the reconfigurable processor data path elements made it difficult to use these registers as accumulators. In particular, DREG1 could only be used as a multiplier term. Although it is possible to use DREG1 as an accumulator by setting the other multiplier equal to 1.0 this technique would require that another processing element be dedicated to providing this constant on the internal bus, and an additional shifting operation might be necessary to reformat the product. Once the 32 intermediate values have been accumulated it is necessary to multiply them by the $\sin(x)$ and $\cos(x)$ terms in the complex twiddle factors. In this part of the algorithm it becomes difficult to use DREG2 because it has no direct input path to the multiplier. It is possible to pass DREG2 directly to the DPE output in one step and then use it as an input to the multiplier in the next step, but this results in poor utilization of the data path resources and an inefficient program.

As a result of these observations a modified data path processor element has been devel-

oped that allows both DREG1 or DREG2 to be used directly as a multiplier or an addend. This was accomplished by simply rearranging the internal data paths in the DPE, as shown in Figure 7. In this configuration both of the data registers connect to both of the data selectors. Some ability to multiplex inputs has been sacrificed, but this does not appear to reduce the usefulness of this architecture.

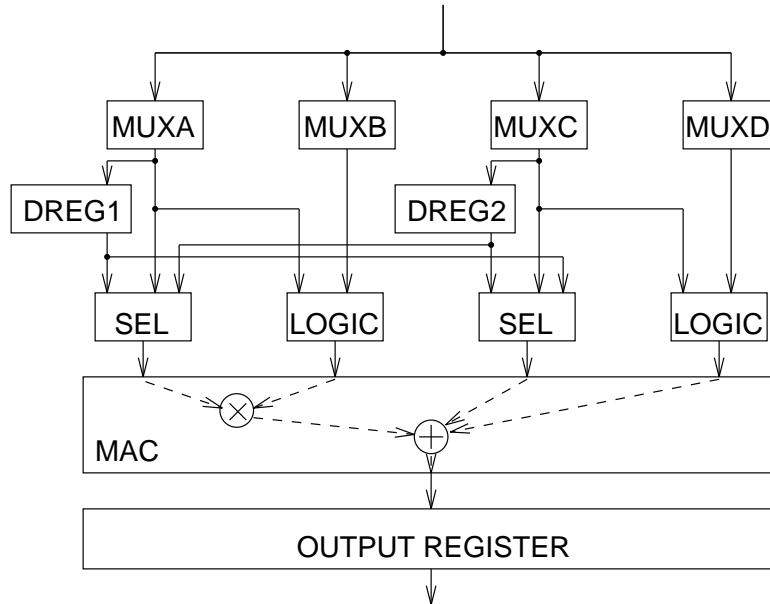


Figure 7: Block Diagram of Modified Reconfigurable Data Path Processor

Other modifications are required at the global processor level, in addition to those described above for the individual processing elements. In the original processor design all data inputs passed through a single processing element. This is a reasonable design choice for FIR and IIR filters where the same simple operation will be performed on every incoming data value. However, this is not the case for an FFT algorithm so the processing element that receives input data can do nothing but pass this data value on to other processing elements, and the resources in this processing element are unavailable to the programmer. There are several ways that the data input problem can be solved. The most straightforward approach would be to add a dedicated input bus which could be accessed through the input multiplexers, but this would require additional bits in the instruction word.

There are other potential modifications to the processing elements that are not required but might enhance the processor's applicability to signal processing algorithms. For example, the original design for the DPE specified that the inputs to the multiplier were half as wide as the other data inputs, so that the multiplier output would then be the same width as the adder outputs. Some precision is lost when internal values are truncated, so a better approach might be to retain more data bits inside the multiplier and then round the result to the desired width. It might also be helpful to support shifts of more than one bit in the output register, perhaps with a partial barrel shifter. When complex data is processed, providing two separate input ports for simultaneous loading of the real and imaginary portions of the data word could greatly increase throughput. Another option might be to input both the

real and imaginary data in the upper and lower halves of a single data word, and modify the LOGIC block so that it can easily select which half of the input word to process.

4 Programming the Parallel Processor

After a few attempts to define programs for the processor, it became apparent that all of the necessary operations performed by an individual DPE while processing the FFT could be reduced to the form

$$A \times B + C \pm D \text{ shift} \quad R1 = A \quad R2 = C$$

and a simple assembler was created to convert instructions in this text form to the 528-bit binary programming words used by the processor. The first expression in each instruction defines the operation performed by the ALU in the DPE and the next two expressions load the data registers, DREG1 and DREG2. Each of these three expressions is optional. If the first expression is omitted the ALU output remains unchanged. If either of the last two expressions is omitted then the corresponding data register remains unchanged. For readability, the assembler allows the instruction field to contain the word HOLD to signify that the ALU output is needed by another processor at some point in the future or NOP to signify that the processor is not being used. Both HOLD and NOP are equivalent to the default instruction, which results in no change to the ALU or data registers.

In general, the operands in the first instruction field, A , B , C , and D , can refer to the ALU outputs of any DPE in the processor. The assembler expects to see each of these operands replaced with P x , where the x is the hexadecimal digit for the processor output to be used. In addition, one of the multipliers and one of the addends can be a data register, which the assembler recognizes as R1 or R2. Note that only the final operand can be subtracted. This is accomplished by forming the 1's complement of this operand using the LOGIC block and setting the carry input to the ALU to a 1. The *shift* value, if specified, causes the ALU result to be shifted one bit to the left if *shift* is <1 or one bit to the right arithmetically if *shift* is >1.

If the first operand is omitted (i.e. the first character in the ALU field is * or +) then the first operand is assumed to be the output of the processor executing the instruction. This notation mimics the *= and += operators found in most programming languages. If both of the multiplier operands (and the * operator) are omitted the assembler substitutes a constant zero value for them, effectively bypassing the multiplier. Similarly, a constant zero is substituted for either or both of the addend operands if they are omitted. If one or both of the data registers are to be loaded then the MUXA or MUXC must be used for this purpose. The assembler recognizes conflicts for assignment of the MUXA and MUXC inputs and will attempt to rearrange the operands in the ALU instruction to avoid such conflicts. Some typical DPE instructions are shown in the table below.

DPE Instruction	Description
P0*P1<1 R1=P1 R2=P7	Multiply the ALU outputs from DPEs 0 and 1 and shift the result one bit to the left. Load DREG1 with the ALU output from DPE 1 and load DREG2 with the ALU output from DPE 7.
*P4-P3	Multiply the ALU output for the DPE executing the instruction by the ALU output from DPE 4 and then subtract the ALU output from DPE 3.
P6*P8 R1=P9	This is an illegal instruction since it causes a conflict for the use of MUXA.
P3*R1-P5	Multiply the ALU output from DPE 3 by the contents of DREG1 in this DPE, then subtract the ALU output from DPE 5.
R2	Load the ALU output with the contents of DREG2.

The reconfigurable data path processor operates most efficiently in a data flow mode. In other words, the data to be processed should flow through the processor uninterrupted once the necessary program has been loaded. Also, conditional branches in the program would greatly increase the complexity of programming the processor so this capability is not supported. One implication of these facts is that arithmetic overflow conditions must be avoided. This can be accomplished by careful specification of the data format and an understanding of how the primitive arithmetic operations affect the data format of the result. The ‘binary point’ notation and methods described in [1] and [2] provide a means for specifying the format of the data inputs so that no overflow can occur. Each data value has an associated format specifier of the form $S/I/F$, where S indicates the number of sign bits in the format, I indicates the number of bits used to express the integer part of the real data value, and F indicates the number of bits used to represent the fractional part of the real value.

For example, the cascaded radix-4 FFT program assumed that all data inputs and twiddle factors had a maximum value of $\pm 0.999\dots$, which is a convenient assumption for real data obtained from an analog-to-digital converter. In this case there are no bits needed in the data format to specify an integer value, which could allow an $S/I/F$ format of $1/0/31$ for a 32-bit operand. When multiplying any operand by a constant value less than one (such as a twiddle factor) we know that the result can not have a larger magnitude than that of the first operand, so the number of integer bits needed to express this value does not increase. On the other hand, forming sums and differences does increase the number of integer bits required. The sum of four values, each not quite equal to 1.0, can have a maximum value not quite equal to 4.0, which requires two bits to express the integer value. In order to prevent an overflow in this summation each of the operands must have at least two extra sign bits. We would say that four operands with a $3/0/29$ format can be summed and the result will have a $1/2/29$ format.

In the first radix-4 stage several sums and differences of four input data values are formed. These are then multiplied by a twiddle factor, and two of the products are summed to produce an output value. If all of the data values and twiddle factors are not quite equal to 1.0 then the outputs from the first radix-4 stage can have a maximum value of not quite 8.0, which

requires three integer bits. The operations in the second radix-4 stage will increase the maximum size of the result by another factor of 8.0, so the outputs of the cascaded radix-4 FFT will need a total of six integer bits in their binary point format, so we choose a format of $1/6/25$ for these output values. To prevent overflows we can provide six additional sign bits in the input data that are converted to integer bits during the FFT processing. However, the product of a signed multiplication has as many sign bits as the sum of the number of sign bits in the multiplier and multiplicand, so in each radix-4 stage the twiddle factor multiplication introduces an additional sign bit into the format. The net result is that we must provide four additional sign bits in the input data values, so we might select a format of $5/0/27$ for the data inputs when using a format of $1/0/31$ for the twiddle factor inputs.

5 Processing Throughput

The cascaded radix-4 algorithm can process a block of 16 complex input data points in 63 clock cycles, where 62 clock cycles are needed just to input the data values and twiddle factors. For a data set of N samples, where $N = 16^x$ for some positive integer x , the data is divided into $\frac{N}{16}$ such blocks for each pass through the processor. A total number of $\log_{16} N$ passes are needed to complete the FFT, so the total processing is

$$T = \frac{N}{16} \times 63 \times \log_{16} N \text{ where } T \text{ is the number of clock cycles}$$

This processing time does not include the overhead needed to initially load the program, which is approximately 1000 clock cycles for the cascaded radix-4 algorithm. The table below summarizes the processing time for several values of N . If N is not 16^x the processor can also use the single radix-4 or radix-2 butterflies as necessary. For example, if $N = 1024$ we can use two passes with the cascaded radix-4 algorithm followed by a single radix-4 pass.

N	$\log_{16} N$	T, clock periods	t, 75MHz clock rate
16	1	63	$0.84\mu\text{s}$
256	2	2,016	$26.9\mu\text{s}$
4,096	3	48,384	$645.1\mu\text{s}$
65,536	4	1,032,192	13.76ms

As a point of reference, the processing time for a 4096-point complex FFT on a 133-MHz Pentium processor is $30,130\mu\text{s}$, almost 50 times slower than the reconfigurable data path processor [3]. The data path processor can outperform commercial DSP processors such as the 40-MHz SHARC ($1,814\mu\text{s}$) and competes favorably with dedicated FFT processors such as the 60-MHz BDS9124 ($208\mu\text{s}$).

6 Conclusion

A complex Fast Fourier Transform has been implemented on a reconfigurable data path processor. Improved techniques for programming and simulating the processor have been

developed. A number of architectural issues were explored that could enhance the ability of this processor to perform the FFT and other signal processing algorithms. Performance estimates based on this work suggest that this processor has a much higher performance than general purpose processors and can approach the capabilities of dedicated FFT processors.

References

- [1] J. E. Simpson, "A block floating-point notation for signal processes," Tech. Rep. SAND79-1823, Sandia National Laboratories, Mar. 1981.
- [2] K. J. Hass, D. H. Lenhart, and N. Ahmed, "On a microcomputer implementation of an intrusion-detection algorithm," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. ASSP-27, pp. 782–789, Dec. 1979.
- [3] H. R. B. Holdijk, "Commercially available DSP chip benchmarks." Available at <http://www.butterflydsp.com/dspbnchm.htm>, Sept. 1997.