The Hazard-Free Superscalar Pipeline Fast Fourier Transform Architecture and Algorithm

Bassam Mohd Earl E. Swartzlander, Jr. Adnan Aziz

Abstract. This chapter examines the superscalar pipeline Fast Fourier Transform algorithm and architecture. The algorithm presents a memory management scheme that avoids memory contention throughout the pipeline stages. The fundamental algorithm, a switch-based FFT pipeline architecture and an example 64-point FFT implementation are presented. The pipeline consists of $\log_2 N$ stages, where N is number of FFT points. Each stage can have M Processing Elements (PEs.) As a result, the architecture speed up is $M*\log_2 N$. The pipeline algorithm is configurable to any $M > 1$.

I. INTRODUCTION

THE FAST FOURIER TRANSFORM (FFT) ALGORITHM, presented in [1], is a standard method for computing the Discrete Fourier Transform (DFT). The FFT algorithm consists of $\log_2 N$ loops; where each loop executes N/2 complex operations. FFT processor design has been researched extensively in the last few decades for speed, area and power optimization. As a result, many implementations have been proposed and developed to address one or more of the following optimization areas: architecture, memory access and power consumption. A variety of *FFT architectures* have been proposed, which employ different techniques such as pipelining, multi-processing and cache-design, as shown in Figure 1 [2]. A *single memory* architecture consists of a scalar processor connected to a single N-word memory via a bidirectional bus. While this architecture is simple, its performance suffers from inefficient memory bandwidth. A *cache memory* architecture adds a cache memory between the processor and the memory to increase the effective memory bandwidth. A *dual memory* architecture uses two memories connected to a digital array signal processor. A memory controller generates addresses to memories in a ping-pong fashion. The *processor array* architecture consists of independent processing elements, with local buffers, which are connected using an interconnect network. Finally, the *pipeline FFT* architecture utilizes $\log_r N$ blocks; each block consists of delay lines and radix-r butterfly units.

Processor *memory access* is another area of optimization that has received considerable research. Several algorithms have been proposed to avoid memory contention. Specifically, the address generation algorithm and logic are optimized for speed and area. A memory address generation scheme was presented by Cohen in [3], that allows parallel organization of memory so that the pairs of data that are used at any instant reside in different memories. The address generation is based on a counter,

shifters and rotators. In [4], Pease proposed dividing the memory into sub-memories for overlapping the access. He observed that the operand addresses differ only in the (n-i)-th bit for the butterfly operand pair in stage i, where n is the number of address bits. A multi-bank memory address assignment for a radix-r FFT was developed in [5]. A fast address generation scheme is described in [6] with hardware cost comparable to the address generation scheme in [3]. Ma and Wanhammar presented an address generation scheme in [7] to reduce the hardware complexity and power consumption. Power is reduced by activating only half of the memory during memory access and by minimizing the number of memory accesses. The methods do not address conflicts for multi-processors accessing memory simultaneously.

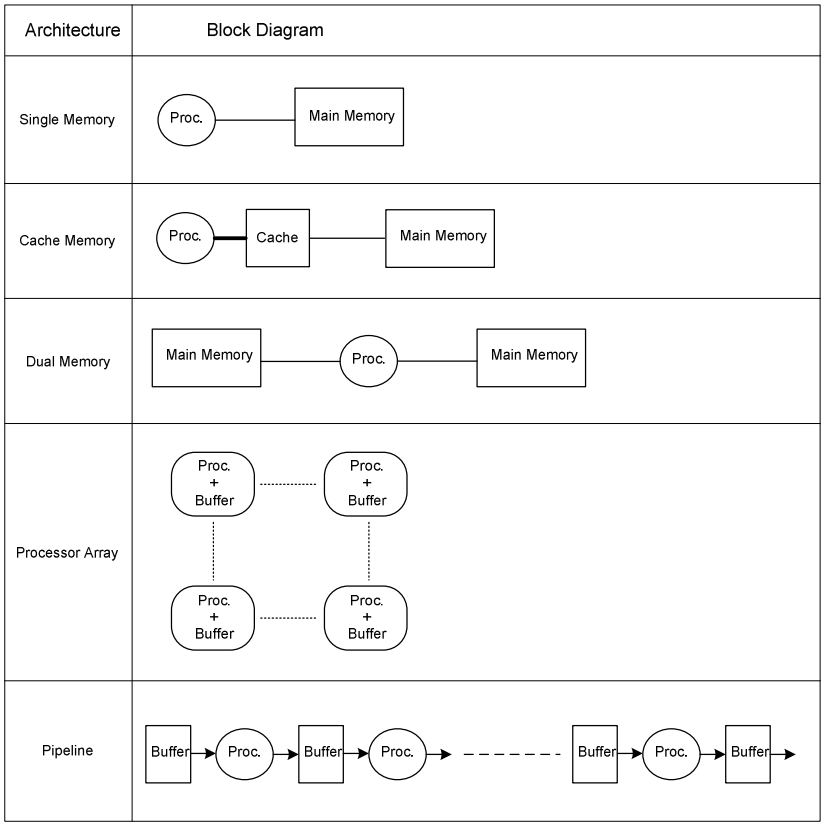

Fig. 1. FFT Processor Memory-System Architectures (after [2])

Lastly, several *power reduction* techniques were designed for energy-efficient processors; including techniques to reduce memory accesses. A cache-memory architecture was described in [8] to reduce communication energy between FFT processors and memories. In [9] and [10], Zhong, *et al.* described a power-scalable reconfigurable ring-architecture multiprocessor for a single chip FFT/IFFT processor.

The processor is capable of processing different FFT sizes with scalable power across FFT sizes. However, while the use of the processor ring architecture seems to be an interesting idea, the case for using the ring architecture to compute FFTs is weak. The architecture seems to be better suited for more serialized computations such as FIR filters. Also, large values of N require more complex processor programs. Further, power does scale well for $N \leq 128$.

This chapter presents a superscalar pipeline architecture to achieve maximum speed for FFT processing. A switch fabric controls and connects single-port memories and processing elements (PEs). A memory management algorithm avoids memory access contention. Rearranging data in the memories requires tracking them throughout the pipeline to process the right pair of data for FFT computations. The ordering of data elements is used to calculate the twiddle factors and other important indices. The algorithm provides an implicit method to track data. The superscalar pipeline achieves a speed up of $M*\log_2 N$.

The chapter is organized as follows. Section II discusses current pipeline designs. Next, Section III explains the pipeline architecture and analyzes pipeline speedup hazards and optimizations. Section IV discusses hazard conditions and resolutions. It provides a pseudo code for the pipeline memory management algorithm. Section V details the design of a 64-point FFT with emphasis on the data movement and storage in the pipeline and memories. Section VI compares the proposed design with other pipeline FFTs.

II. EXISTING PIPELINE FFT ARCHITECTURES

This section reviews the main pipeline FFT architectures. Groginsky and Works developed an early pipeline FFT design [11]. Several pipeline FFTs have been implemented [12]-[14]. Later, several pipeline architectures were proposed and designed [15]-[17]. Pipeline FFT processors consist of $\log_r N$ stages, each stage utilizes variable sizes of memories and complex multipliers/adders depending on the pipeline type. Because it performs $\log_r N$ butterflies in parallel, the radix-r pipeline FFT processor has as a speed-up of (at least) $\log_r N$ compared to an FFT performed on a single radix-r FFT processor. Based on the number of paths between stages, FFT pipelines are classified into Single-path Delay Feedback (SDF) and Multi-path Delay Commutator (MDC). The modular pipeline constructs the pipeline from two smaller pipelines to reduce power. The rest of this section will explain the SDF, MDC and modular pipelines.

SDF Pipeline FFT

The SDF pipeline FFT has one path between stages, as shown in Figure 2. The pipeline uses feedback registers in each stage. The feedback registers store previous stage outputs for use by the butterfly. Figure 2 illustrates the SDF pipeline FFT for a

radix-r N-point FFT and shows an example of an 8-point radix-2 pipeline [15], [16]. Each SDF stage is comprised of:

- A radix-r FFT butterfly. Each butterfly is followed by a complex multiplier (shown explicitly in Figure 2), with the exception of the last stage.
- Shift registers to hold intermediate values. For stage i, the number of shift registers is $(r-1)(N/r^{(stage+1)})$, e.g., stage 0 has (r-1)(N/r) registers.

The pipeline hardware complexity depends on the number of delay elements and multipliers. The total number of complex multipliers is $(\log_r N - 1)$ [15], [16]. Additionally, the total number of registers in the pipeline is N-1. A high radix SDF (i.e., r >2) can be also implemented by cascading several radix-2 processing elements referred to as 2^s [15]. Calculating pipeline throughput and complexity is straightforward. The SDF pipeline accepts a new point each clock cycle. Further, it outputs one point per cycle. Therefore, the pipeline throughput is one point per cycle.

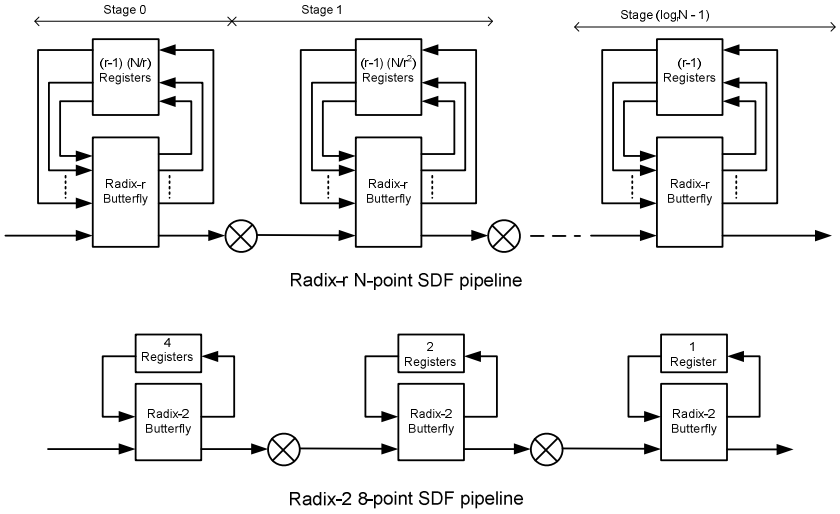

Fig. 2. SDF Pipeline FFT (after [15])

MDC Pipeline FFT

The radix-r MDC pipeline FFT utilizes *r* paths between stages, as shown in Figure 3 [15], [16]. With the exception of one path, all paths utilize delays with different numbers of registers. Each stage receives *r* intermediate results from the previous stage, and passes *r* outputs to the next stage. An example of an 8-point radix-2 MDC pipeline FFT is shown in Figure 3. An MDC stage is comprised of:

- An *r*-input commutator,
- A radix-r butterfly which includes *(r-1)* complex multipliers

- Two sets of shift registers. The first set is located before the commutator (shown as D). This set does not exist in stage 0. The second set is situated after the commutator. Moreover, the number of registers in the j-th element of each set in stage i can be expressed as: $Di_j = DDi_j = j \times (N/ r^{i+1})$. An example of the shift register sizes for a 1024-point radix-4 pipeline FFT is shown in Table 1.

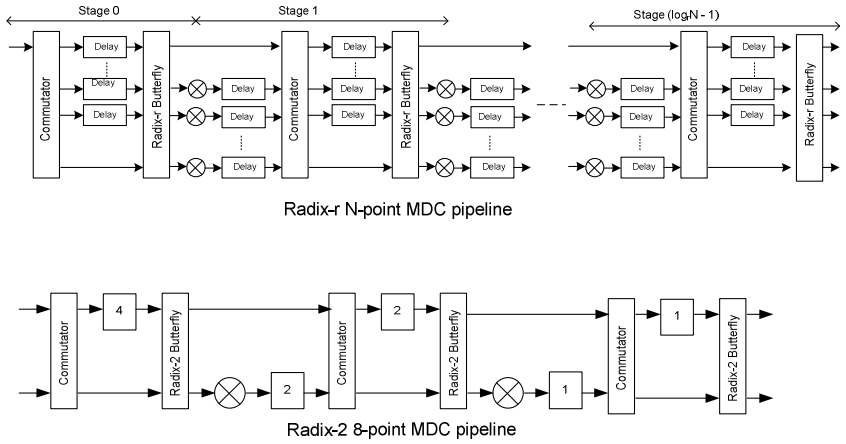

Fig. 3. Radix -r N-point MDC Pipeline (after [16])

Table 1. DMC Delay Element Sizes for a 1024 Point Radix-4 FFT Processor

Stage	D size	DD size
0	N/A	64, 128, 192
1	16, 32, 48	16, 32, 48
2	16, 32, 48	16, 32, 48
3	4, 8, 12	4, 8, 12
4	1, 2, 3	1, 2, 3

The pipeline complexity is a function of the number and size of delay shift registers, adders and multipliers. The total number of delay registers is (r+1)N/2 – r. In addition, there are *(r-1) (log$_r$N -1)* complex multipliers and *2(r-1) (log$_r$N -1)* complex adders in the pipeline [12], [16]. In contrast to the SDF pipeline, the MDC pipeline receives r points and outputs r points in each clock cycle. Thus, the pipeline throughput is r.

The Modular Pipeline

El-Khasahab, *et al*. developed the modular pipeline FFT detailed in [18]-[20]. The N-point modular pipeline FFT consists of two $\sqrt{N}$ -point FFT modules joined by a specialized center element. The center element contains coefficient and data memory as well as addressing, routing and control logic. The modular pipeline FFT significantly reduces the size of the shift registers. Moreover, the coefficient storage is concentrated within the center element, which can be implemented using energy-efficient RAM memories. Further, the throughput of the modular pipeline FFT is identical to that of the standard pipeline FFT, although the end-to-end latency is very slightly higher.

The modular pipeline FFT algorithm is expressed mathematically by the following equation, which demonstrates the two-stage N-point FFT:

$$X(\sqrt{N}k_1 + k_0) = \sum_{m_0=0}^{\sqrt{N}-1} W_N^{m_0 k_0} \left(\left(\sum_{m_1=0}^{\sqrt{N}-1} x(\sqrt{N}m_1 + m_0) W_{\sqrt{N}}^{m_1 k_0} \right) \times W_{\sqrt{N}}^{m_0 k_1} \right) \quad (1)$$

$$X(\sqrt{N}k_1 + k_0) = \sum_{m_0=0}^{\sqrt{N}-1} \sum_{m_1=0}^{\sqrt{N}-1} x(\sqrt{N}m_1 + m_0) W_N^{m_1 k_0 \sqrt{N} + m_0 k_1 \sqrt{N} + m_0 k_0}$$

where: $0 \le k_0, k_1 \le \sqrt{N} - 1$

To obtain the correct results, the transforms of the first stage are combined (in a fixed way) and fed to the second stage. Further, adjustment is made for intermediate results prior to second stage. Figure 4 shows how to construct a 16-point FFT with the second stage having same four FFTs as first stage. This demonstrates that the N-point FFT is now divided into two $\sqrt{N}$ point FFTs.

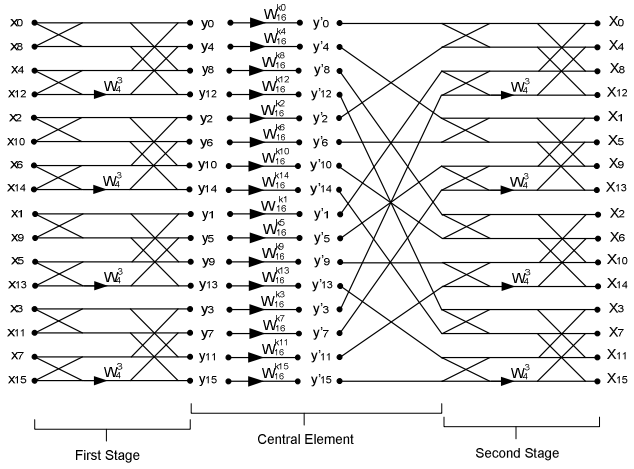

Fig. 4. 16-Point FFT Butterfly with Identical First and Second Stages [18]

Figure 5 shows the overall architecture of an N-point radix-2 modular pipeline FFT. It consists of the two $\sqrt{N}$ -point FFT blocks and a center element. The center element includes an address generator, RAMs for storing intermediate values and ROMs for the coefficients. The design allows data to be both read and written simultaneously to maximize performance. The pipeline operation can be explained as follows. Two discrete inputs are received from the left side of the pipeline. The address generation guarantees the two points have different parities, and hence they reside in different memories. Once $\sqrt{N}$ points have been output from the first stage, the control dispatches intermediate data to second stage. At the same time, the next $\sqrt{N}$ points begin entering the first stage. Hence the pipeline is able to input and output data every clock.

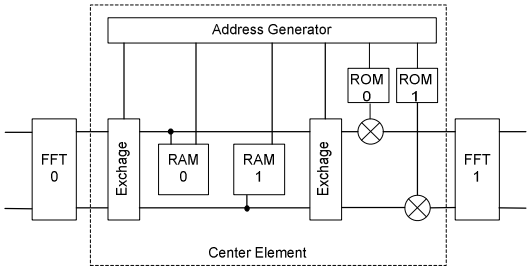

Fig. 5. Radix-2 Modular Pipeline Architecture [19]

Table 2 compares the modular pipeline with a conventional N-point pipeline FFT. Despite the fact that it requires a larger memory; the modular pipeline has fewer shift registers. The modular pipeline FFT requires an additional pre-rotation multiplication and has very slightly higher latency than the standard pipeline FFT.

Table 2. Complexity of Radix-r Conventional and Modular Pipeine FFTs Using Optimum Sized Stages

Parameter	STANDARD	Modular
ROM (Coefficient)	N-r	$2(\sqrt{N}$ -r)
Shift Registers	N-r	$2(\sqrt{N}$ -r)
Complex Multipliers	$\log_r(N)$-1	$\log_r(N)$-1
Central Element RAM	0	N
Throughput	r points / cycle	r points / cycle
Delay	$2\left(\frac{N}{r}\right)$	$\frac{2}{r}\left(N+\sqrt{N}\right)$

III. THE SWITCH-BASED ARCHITECTURE

This section describes the superscalar pipeline architecture for a radix-2 FFT.

Superscalar Pipeline Architecture

The pipeline architecture of an N-point radix-2 FFT consists of $\log_2(N)$ stages. Figure 6 shows a block diagram of the pipeline stage. Stage i of the pipeline executes the i-th loop of the Radix-2 decimation-in-frequency FFT algorithm.

Each stage consists of:

1. A switch fabric that connects PEs and memories.
2. PEs that have three inputs (a, b, w) and two outputs (c, d) and perform the radix-2 butterfly operation:

$$c = a + b$$
$$d = (a - b) * w \quad (1)$$

   (a, b) are inputs, *w* is the twiddle factor and *(c, d)* are outputs. There are *M* PEs per stage, where
   - $N/2 \geq M \geq 2$
   - $M = 2^p$, where *p* is an integer $p > 1$.
3. Memories that store intermediate results. There are 4*M single-port memories per stage, the size of each memory is equal to N/(2*M). Memories can be implemented as RAM, caches, register files or flip-flops, based on the size of the memory and cost constraints. One half of the input memories will be active per cycle, while the other half will be active in the following cycle

4. Memories that store twiddle factors. Since the twiddle factors do not change, the twiddle factor memories can be implemented as ROMs. There are M ROMs per stage, each with size equal to N/(2*M) words.

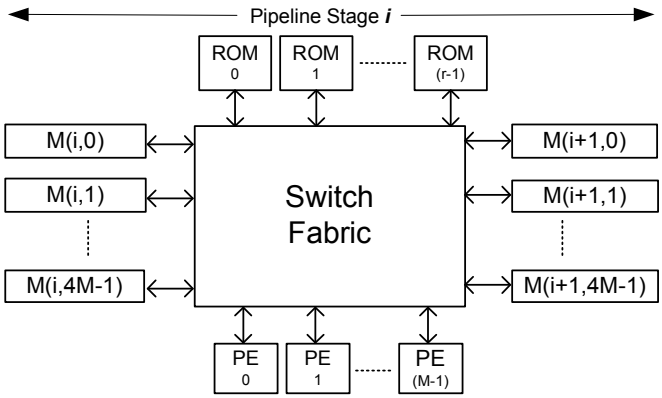

Fig. 6. Block Diagram of the Switch-Based Pipeline Stage [21]

Figure 7 shows an overview of pipeline architecture. Each stage is capable of calculating M radix-2 butterfly results. Using the Instruction Level Parallelism (ILP) classification from [22], the architecture is a superscalar machine with Instruction Parallelism (IP) equal to M. It is also a super-pipeline where each cycle has N/(2*M) minor-cycles. The architecture applies to the decimation-in-time FFT as well, where the specifications of stage i in the decimation-in-time algorithm is the same as that of stage $\log_2(N)$–i in the decimation-in-frequency algorithm. A scalar machine takes $(N/2)*\log_2(N)$ steps to execute an N-point radix-2 FFT algorithm. The architecture consists of $\log_2(N)$ stages, where each stage executes M operations. Therefore, the pipeline speedup is: $M*\log_2(N)$. The maximum pipeline speedup is $(N/2)*\log_2(N)$, when M = N/2. In this case memories are reduced to registers, and the switch fabric connects each any register to any PE. Clearly, while this case provides the most speed up, its hardware is expensive. The optimum value of M is decided by design parameters: speed, area and power.

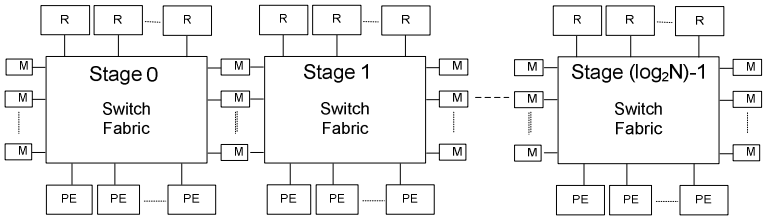

Fig. 7. Overview of the Pipeline Architecture [21]

Pipeline Design Optimization

Upon close examination of the FFT algorithm, it is clear that not all twiddle factors are used in all stages. Also, the algorithm allows PEs to have identical twiddle factors in some stages, and therefore, not all the ROMs are required. In fact, the number and size of ROMs per stage can be reduced as outlined in Table 3.

Table 3. Number and Size of ROM Size Per Stage

Stage "i"	Number of ROMs	Size of ROM
0	M	N/(2*M)
$\log_2 M \geq i \geq 0$	M	$N/(M* 2^i)$
$i > \log_2 M$	$M/2^{(i- \log 2M)}$	1

If the pipeline is designed for a specific value of N, where N is fixed, the pipeline connectivity and twiddle factors are fixed. As a result, the design implementation can be optimized since the connectivity of each stage is predetermined. Figure 8 illustrates the connectivity of 16-point 2-PE pipeline. Furthermore, in many computations the value of the twiddle factor is one. A twiddle factor of one reduces the PE computation to add/subtract operations. Also, several PEs execute specific sets of twiddle factors, which can lead to design simplification.

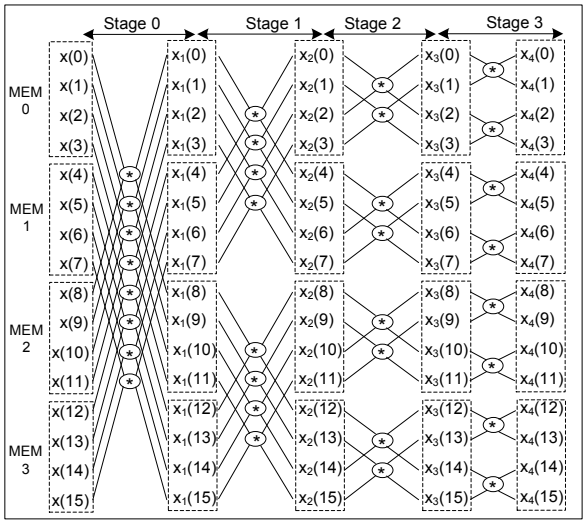

Fig. 8. Example FFT Data Flow [21]

As indicated earlier, the speed up of the pipeline depends on two factors: the number of PEs/stage (i.e., M) and the number of stages ($\log_2(N)$) since Speedup = $M*\log_2(N)$. One might ask, *"Given fixed target speedup (e.g., S), which factor should be increased to achieve more efficient design: the number-of-stages or the number-of-PEs/stage?"* Consider a pipeline with a speedup of S with two designs: Design A and design B, as shown in Table 4. Design A has one PE per stage, while design B has one stage. Clearly,

- Design B requires less memory than design A since the design A total memory is proportional to S.
- Design A switch fabric is simpler than that of design B. The complexity of the design B switch fabric is proportional to S^2.

Table 4. Analyzing Speed Up Factors

Parameter	Design A	Design B
Number of Stages	S	1
Number of PEs per Stage	1	S
Memory Size	N/2	N/(2*S)
Number of Memories	4*(S+1)	2*S
Total Memory	2*N*(S+1)	N
Switch Complexity	2*2	S*S

The main disadvantage of the increasing the number of stages is the increase in total memory. On the other hand, increasing the number of PEs per stage increases the complexity of the switch fabric. Hence, the tradeoffs between the two factors depend

on the constraints on the total memory and the maximum complexity of the switch. Only specific design goals and technology processes can determine the optimum solution.

Pipeline Hazards

The main source of hazards in the pipeline is memory contention. Memory contention occurs when one or more PEs requests two or more accesses to a given memory at the same time. Memory contention results in stalling the pipeline and reduces the system speed. In the decimation-in-frequency FFT, memory contention does not occur in the early stages, it occurs from stage $\log_2(M)+1$ to the last stage. In the decimation-in-time FFT, contention affects stage 0 to stage $\log_2(N) - \log_2(M) - 1$.

Figure 8 shows an example of memory contention for N=16 and M=2. It is clear that stage 0 and stage 1 have no contention. However, contention occurs in stage 2 and stage 3. Observe the following:

- In stage 2 the inputs for the top PE are $x_2(0)$ and $x_2(2)$, both of which reside in MEM0.
- In stage 3 the inputs for the top PE are $x_3(0)$ and $x_3(1)$, both of which reside in MEM0.

One solution for memory contention is to use a multi-port memory. However, multi-port memories are expensive and can slow down the system performance. In addition, the later stages of the pipeline have higher degree of contention which requires more ports in the memory. Eventually, it becomes impractical to implement the required multi-port memory. Moreover, the number of memory ports varies in the memory hierarchy. Register files usually have more ports than caches and SRAMs. Requiring a certain number of memory ports restricts where the intermediate results can be saved in the memory system. Another solution to resolve memory contention is to employ a memory management mechanism to mitigate the hazard, as discussed in the next section.

IV. HAZARD FREE PIPELINE ALGORITHM

The main idea of the algorithm is resolve memory contention in the early stages of the pipeline. The rest of the section describes the hazard conditions, memory management operations and the algorithm.

Detecting Pipeline Hazards

From Figure 8, in stage 0, x(0) and x(8) go to PE_0. Similarly, x(1) and x(9) go to PE_1,..., etc. Define stage distance as the index delta in each stage. The stage distance for a 16-point pipeline FFT is shown in Table 5.

Table 5. Stage Distance For 16-point Pipeline FFT

Stage	Stage Distance	
	Decimation-In-Frequency	Decimation-in-Time
0	8	1
1	4	2
2	2	4
3	1	8

In general, for an N-point pipeline FFT, the stage distance for stage *i* is equal to $N/2^{(i+1)}$. Memory contention occurs when the stage distance falls in a single memory space. From Section III, the memory size is equal to N/(2*M). Hence, memory contention occurs in stage *i* if the following condition is satisfied:

$$N / 2^{(i+1)} \le N /(2^{M}) \qquad (2)$$
$$i \ge \log_2(M)$$

A stage that satisfies condition (2) will be referred to as a hazard stage; the rest of the stages are safe stages. For instance, in Figure 8, stage 2 and stage 3 are hazard stages. Define memory pair $(i, j)_t$ as memory location x(i) and x(j) for stage *t*. In stage 2, the following memory pairs are hazard pairs: $(0, 2)_2$, $(1, 3)_2$, $(4, 6)_2$, $(5, 7)_2$. Other pairs will be referred to as safe pairs, for instance $(3, 5)_2$. The stage distance can be represented in binary form:

Stage-3 distance = 001

Define pair $(i, j)_t$ as a hazard pair if and only if:

1. *t* is a hazard stage
2. The bit wise Exclusive-OR of addresses i and j is equal to the stage t distance.

For example, the address pair $(5, 7)_2$ is a hazard pair since:

Stage-2 distance = 2_{10}

$5_{10} \oplus 7_{10} = 101_2 \oplus 111_2 = 010_2$ = *Stage-2 distance*

On the other hand, address pair $(3, 5)_2$ is a safe pair because:

$3_{10} \oplus 5_{10} = 011_2 \oplus 101_2 = 110_2$!= *Stage-2 distance*

Memory Management Operations

Let $x_i(t)$ and $x_j(t)$ be the i-th and j-th elements in stage t and i < j. Define the memory management operations as follows (see Figure 9):

- **Normal Operation**: Inputs $x_i(t)$ and $x_j(t)$ are provided to the first and second inputs of the PE: a, b. The results c and d are saved in $x_i(t+1)$ and $x_j(t+1)$.
- **Shuffle Operation** affects how PE results are saved back in memory. In shuffle operation, the results c and d are saved in $x_j(t+1)$ and $x_i(t+1)$
- **Swap Operation**: The swap operation affects the order of PE inputs. In swap operation, $x_i(t)$ is provided to b (instead of a) and $x_j(t)$ is provided to a (instead of b). The reason for the swap operation is because the PE is an asymmetric unit and the memory management algorithm changes the normal order of data in the

memory. If the algorithm detects a case with incorrect inputs, the swap operation is performed.

- **Swap and shuffle operation**: A PE operation can have both swap and shuffle memory operations at the same time.

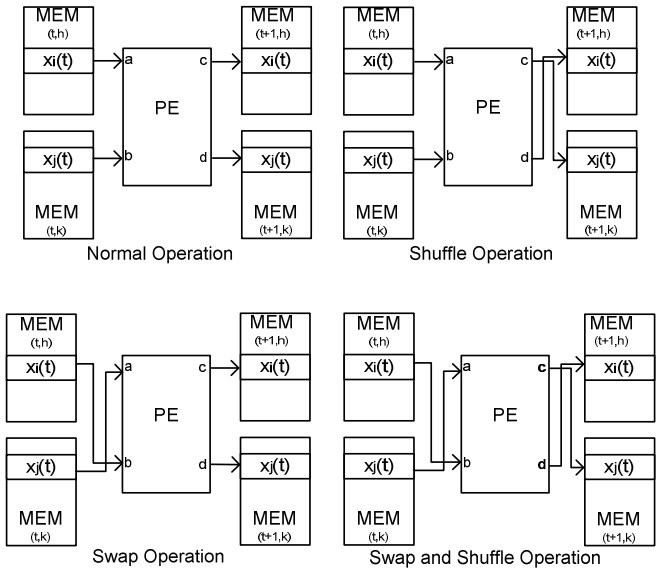

Fig. 9. Memory Management Operations [21]

The Algorithm

The main idea of the pipeline algorithm is to identify hazard pairs in early stages and perform memory management operations to resolve the hazard. Because data is rearranged in memory, the algorithm has to track where data is. One idea to track the movement of data is to use a separate memory to store the data indexes (i.e., pointers), as shown in Figure 10. This approach provides a great flexibility in moving data in the memory. It also simplifies the reordering logic of the final stage hardware. The downside of this approach is it increases memory size. Also, it increases loading the operands in the PE by one cycle to retrieve pointers from memory. Another (less flexible) solution is to move data in memory in a fixed way to simplify data tracking in the pipeline. This approach resolves hazards for next stage only. As a result of reordering data in the pipeline, results from the last stage in the pipeline should be reordered.

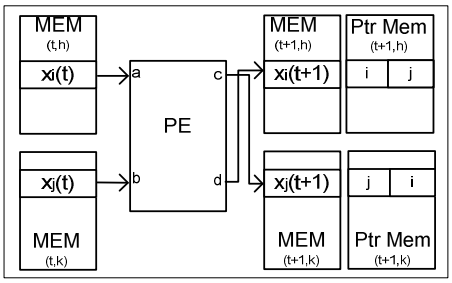

Fig. 10. Tracking Shuffled Data [21]

The algorithm utilizes several counters to calculate memory addresses and determine memory management operations. There are three main counters which are described in the upper three rows of Table 6. Other counters are derived from the main counters and described in the rest of the table. The flow of the algorithm of stage *i* is shown in Figure 11. The pseudocode of the algorithm is listed at the end of the section. Figure 12 illustrates the shuffle and swap operations performed by the algorithm to resolve the memory contentions in Figure 8 example.

Table 6. The Main Counters

Counter	Description/Usage
Current_Stage	Stage counter
Current_Stage_Cycle	Cycle counter within a stage
Current_Cycle_Operation	Operation counter within a cycle
Horizontal_op_index	Determines shuffle operations
Vertical_op_index	Used in generating RAM addresses
Group_Count	Determines swap operation
Current_Operation	Used in generating RAM addresses

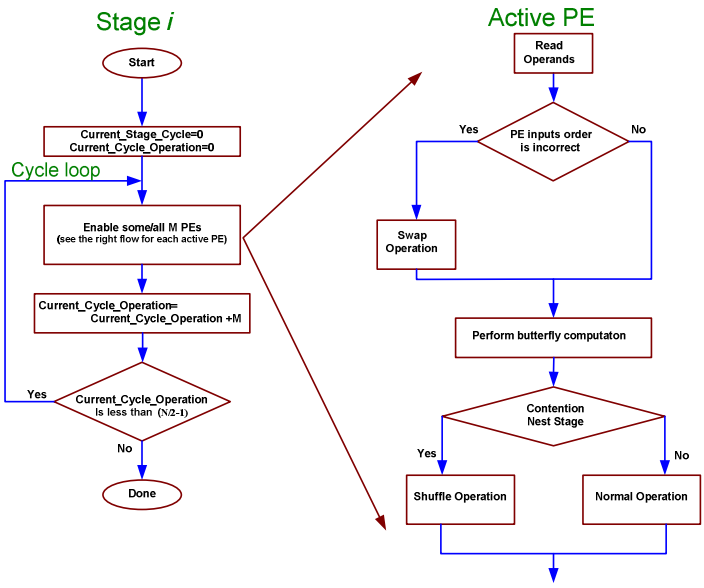

Fig. 11. Algorithm Flow in Stage *i*

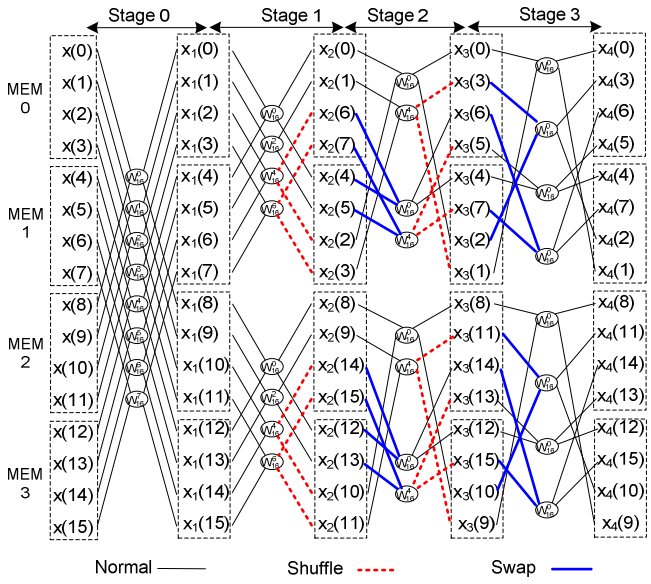

Fig. 12. Resolving Contentions in Pipeline Hazard Example [21]

Algorithm Pseudocode

```
// Preparation Step
Number_Of_Stages   = log2NUMBER_OF_FFT_POINTS
Cycles_Per_Stage   = N/(2*NUMBER_OF_PE)
Memory_Size        = N/2^(NUMBER_OF_PE+1)
Safe_Stage         = log2NUMBER_OF_PE
// Start main nester loops
for Current_Stage=0 to (Number_Of_Stages -1)
 Group_Size = N/2^(Current_Stage+1)
 for Current_Stage_Cycle=0 to (Cycles_Per_Stage -1)
  for Current_Cycle_Operation=0 to (NUMBER_OF_PE -1)
   // Calculate Operation Indices
   Horizontal_op_index = Cycles_Per_Stage *
                         Current_Cycle_Operation
                         + Current_Stage_Cycle
   Vertical_op_index   = NUMBER_OF_PE * Current_Stage_Cycle
                         + Current_Cycle_Operation
   Current_Stage_Rev = Number_Of_Stages - Current_Stage - 1
   Current_Group     = floor(Horizontal_op_index/
                             2^Current_Stage_Rev)
   Current_Operation = Horizontal_op_index mod 2^Current_Stage_Rev
   // Calculate Memory Address
   M0_addr = Current_Stage_Cycle
   If Current_Stage <= Safe_Stage
     M1_addr = M0_addr
   Else
     K = Safe_Stage +1
     L = Current_Stage
     M1_Addr = Reverse M0_Addr0 bits between K to L bits
   End
   // Calculate Memory Select
   If Current_Stage <= Safe_Stage
     Group_Offset = Current_Group * N /2^Current_Stage
     Group_Count  = Horizontal_op_index mod Group_Size
     Memory_Count = floor (Group_Count / Memory_Size)
     Offset       = Memory_Count * Memory_Size
     M0_Select    = Offset + Group_Offset
     M1_Select    = Offset + Group_Offset + Group_Size
   Else
     Memory_Count = Vertical_op_index mod NUMBER_OF_PE
     Offset    = 2 * Memory_Count * Memory_Size
     M0_Select = Offset;
     M1_Select = Offset + 2 * Memory_SiZe
   End
   M0_data = Memory(Current_Stage, M0_Select0) [ M0_addr ]
   M1_data = Memory(Current_Stage, M1_Select1) [ M0_addr ]
   // Determine if swap operation is required
   If  Current_Group is even
       AND Current_Sage <= Safe_Stage
     // Read data with no swap
     M0_data = Memory(Current_Stage, M0_Select) [ M0_addr ]
     M1_data = Memory(Current_Stage, M1_Select) [ M1_addr ]
   Else
     // Read Data and perform Swap
     M1_data = Memory(Current_Stage, M0_Select) [ M0_addr ]
     M0_data = Memory(Current_Stage, M1_Select) [ M1_addr ]
End
// Read Twiddle
```

```
ROM_SELECT  = Current_Cycle_Operation
ROM_Address = Current_Operation * 2^Current_Stage
   W    = ROM(Current_Stage, ROM_SELECT) [ROM_Address ]
// Enable PE to perform FFT butterfly operation
[Result1, Result0] =
      PE_Current_Cycle_Operation(M0_data, M1_data, W);
   // Perform shuffle operation
   Shuffle_Bit = log2NUMBER_OF_FFT_POINTS
                 - Current_Stage - 2
   Shuffle_Flag = Horizontal_op_index [Shuffle_Bit]
   If  Current_Stage >= Sage_Stage  AND
     Shuffle_Flag == 1
     // Shuffle ResultsShuffle = 1
     Memory(Current_Stage+1, M0_Select) [ M0_addr ] = Result1
     Memory(Current_Stage+1, M1_Select) [ M1_addr ] = Result0
   Else
     // No Shuffling
     Memory(Current_Stage+1, M0_Select) [ M0_addr ] = Result0
     Memory(Current_Stage+1, M1_Select) [ M1_addr ] = Result1
   End
  end // Current_Cycle_Operation
 end // Current_Stage_Cycle loop
end // Current_Stage loop
```

V. 64-POINT PIPELINE FFT DESIGN

This section explains a 64-point pipeline FFT design using four PEs per stage. Therefore, although there are 16 memories per stage, only eight memories will be active memory at any time. The memory size is eight words. There are four ROMs per stage, each with a capacity of eight words. The pipeline speed up equals 6*4=24. The following tables detail the operation of the pipeline PEs and illustrate the memory contents.

Table 7 gives the PE operand pairs for Stage 0. The rows give the operand pairs for PE_0, PE_1, PE_2 and PE_3. The columns give the pairs for each micro-cycle in Stage 0 cycles. There are eight micro-cycles per stage. For example, at micro-cycle 0:

- PE_0 input operands will be MEM[0] and MEM[32]
- PE_1 input operands will be MEM[8] and MEM[40]
- PE_2 input operands will be MEM[16] and MEM[48]
- PE_3 input operands will be MEM[24] and MEM[56]

Tables 8-12 give the PE operand pairs for Stages 1-5. Underlined pairs indicate shuffle operation. Since Stages 0-2 are safe stages, the first shuffle operation starts in Stage 2 to prevent hazards in stage 3. Table 13 lists the memory contents for pipeline stages. For example, the output of stage 2 has the memory contents for Memory 0 as follows: 0, 1, 2, 3, 12, 13, 14, and 15.

Table 7. Pipeline Stage-0 Operand Paris

PE	Stage-0 Cycles							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0,32	1,33	2,34	3,35	4,36	5,37	6,38	7,38
1	8,40	9,41	10,42	11,43	12,44	13,45	14,46	15,47
2	16,48	17,49	18,50	19,51	20,52	21,53	22,54	23,55
3	24,56	25,57	26,58	27,59	28,60	29,61	30,61	31,63

Table 8. Pipeline Stage-1 Operand Paris

PE	Stage-1 Cycles							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0,16	1,17	2,18	3,19	4,20	5,21	6,22	7,23
1	8,24	9,25	10,26	11,27	12,28	13,29	14,30	15,31
2	32,48	33,49	34,50	35,51	36,52	37,53	38,54	39,55
3	40,56	41,57	42,58	43,59	44,60	45,61	46,62	47,63

Table 9. Pipeline Stage-2 Operand Paris

PE	Stage-2 Cycles							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0,8	1,9	2,10	3,11	4,12	5,13	6,14	7,15
1	16,24	17,25	18,26	19,27	20,28	21,29	22,30	23,31
2	32,40	33,41	34,42	35,42	36,44	37,45	38,46	39,47
3	48,56	49,57	50,58	51,59	52,60	53,61	54,62	55,63

Table 10. Pipeline Stage-3 Operand Paris

PE	Stage-3 Cycles							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0,4	1,5	2.6	3,7	12,8	13,9	14,10	15,11
1	16,20	17,21	18,22	19,23	28,24	29,25	30,26	31,27
2	32,36	33,37	34,38	35,39	44,40	45,41	46,42	47,43
3	48,52	49,53	50,54	51,55	60,56	61,57	62,58	63,59

Table 11. Pipeline Stage-4 Operand Paris

PE	Stage-4 Cycles							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0,2	1,3	6,4	7,5	12,14	13,15	10,8	11,9
1	16,18	17,19	22,20	23,21	28,30	29,31	26,2	27,25
2	32,34	33,35	38,36	39,37	44,46	45,47	42,40	43,41
3	48,50	49,51	54,52	55,53	60,62	61,63	58,56	59,57

Table 12. Pipeline Stage-5 Operand Paris

PE	Stage-5 Cycles							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0,1	3,2	6,7	5,4	12,13	15,14	10,11	9,8
1	16,17	19,18	22,23	21,20	28,29	31,30	26,27	25,25
2	32,33	35,34	38,39	37,36	44,45	47,46	42,43	41,40
3	48,49	51,50	54,55	53,52	60,61	63,62	58,59	57,56

Table 13. Pipeline Memory Content

MEM	Stages						
	Input	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0	0
	1	1	1	1	1	3	3
	2	2	2	2	6	6	6
	3	3	3	3	7	5	5
	4	4	4	12	12	12	12
	5	5	5	13	13	15	15
	6	6	6	14	10	10	10
	7	7	7	15	11	9	9
1	8	8	8	8	8	8	8
	9	9	9	9	9	11	11
	10	10	10	10	14	14	14
	11	11	11	11	15	13	13
	12	12	12	4	4	4	4
	13	13	13	5	5	7	7
	14	14	14	6	2	2	2
	15	15	15	7	3	1	1
2	16	16	16	16	16	16	16
	17	17	17	17	17	19	19
	18	18	18	18	22	22	22
	19	19	19	19	23	21	21
	20	20	20	28	28	28	28
	21	21	21	29	29	31	31
	22	22	22	30	26	26	26
	23	23	23	31	27	25	25
3	24	24	24	24	24	24	24
	25	25	25	25	25	27	27
	26	26	26	26	30	30	30

	27	27	27	27	31	29	29
	28	28	28	20	20	20	20
	29	29	29	21	21	23	23
	30	30	30	22	18	18	18
	31	31	31	23	19	17	17
4	32	32	32	32	32	32	32
	33	33	33	33	33	35	35
	34	34	34	34	38	38	38
	35	35	35	35	35	37	37
	36	36	36	44	44	44	44
	37	37	37	45	45	47	47
	38	38	38	46	42	42	42
	39	39	39	47	43	41	41
5	40	40	40	40	40	40	40
	41	41	41	41	41	43	43
	42	42	42	42	46	46	46
	43	43	43	43	47	45	45
	44	44	44	36	36	36	36
	45	45	45	37	37	39	39
	46	46	46	38	34	34	34
	47	47	47	39	35	33	33
6	48	48	48	48	48	48	48
	49	49	49	49	49	51	51
	50	50	50	50	54	54	54
	51	51	51	51	55	53	53
	52	52	52	60	60	60	60
	53	53	53	61	61	63	63
	54	54	54	62	58	58	58
	55	55	55	63	59	57	57
7	56	56	56	56	56	56	56
	57	57	57	57	57	59	59
	58	58	58	58	62	62	62
	59	59	59	59	63	61	61
	60	60	60	52	52	52	52
	61	61	61	53	53	55	55
	62	62	62	54	50	50	50
	63	63	63	55	51	49	49

VI. Comparison with Other FFT Pipelines

The hardware complexity of a pipeline FFT is measured by the number of complex adders, complex multipliers and the memory size. A radix-2 butterfly consists of one complex multiplier and two complex adders which can be implemented using four real multipliers and six real adders. A radix-4 butterfly consists of three complex multipliers and eight complex adders and can be implemented using 12 real multipliers and 22 real adders. Less expensive (but slower) butterfly implementations exist especially for slow pipelines, e.g., SDF pipelines. The rest of this section uses counts of complex operations to compare different pipelines.

The SDF pipeline FFT has a total of $(\log_r N - 1)$ multipliers and $N-1$ delay elements. Further, the MDC pipeline FFT utilizes $(r+1)N/2 - r$ delay elements, and $(r-1)(\log_r N - 1)$ real multipliers and roughly $2(r-1)(\log_r N - 1)$ adders. Table 14 summarizes the hardware and timing complexities for FFT pipeline architectures discussed in references [18], [20]. The table also illustrates the complexities for the switch based architecture (shown in the last row of the table.) The other pipeline architectures require delay elements in the pipeline implementation. Delays are implemented by shift registers (which dissipate high dynamic power) or by RAMs with additional address generation hardware (which increases design complexity). The modular pipeline reduces number of delay elements to $2(\sqrt{N}$ -r). The switch-based pipeline uses SRAM memory arrays, which consume less power than registers and are easier

to implement. Moreover, the throughputs of the other pipelines are limited to one (single-path) or a few (multi-path) data per clock, while the switch based implementation has a throughput of M. Unfortunately, the switch based pipeline requires larger memory size and more hardware in the data path.

Table 14. FFT Pipeline Architectures

FFT Pipeline	Multipliers	Adders	Memory Size	Speed up
Radix-2 SDF	$2(\log_4 N\text{-}1)$	$4\log_4 N$	N - 1	$\log_2 N$
Radix-4 SDF	$\log_4 N\text{-}1$	$8\log_4 N$	N - 1	$\log_2 N$
Radix-2 MDC	$2(\log_4 N\text{-}1)$	$4\log_4 N$	$3N/2$ - 2	$\log_2 N$
Radix-4 MDC	$3(\log_4 N\text{-}1)$	$8\log_4 N$	$5N/2$ - 4	$\log_2 N$
Radix-4 Single-path Delay Commutator	$\log_4 N\text{-}1$	$3\log_4 N$	$2N/2$ - 2	$\log_2 N$
Radix-2^2 Single-path Delay feedback	$\log_4 N\text{-}1$	$4\log_4 N$	N - 1	$\log_2 N$
Radix-2 Modular Pipeline	$2(\log_4 N\text{-}1)$	$4\log_4 N$	N - 6 + 2*sqrt(N)	$\log_2 N$
Switch-Based Pipeline	$M*2(\log_4 N\text{-}1)$	$M*4\log_4 N$	$2*N*(1+\log_2 N)$	$M*\log_2 N$

VII. CONCLUSION AND FUTURE WORK

This chapter extends results from [21]. It presents a switch-based architecture for FFT engine implementation. It also presents an algorithm to predict and resolve memory contentions. As a result the pipeline speedup is $M*\log_2 N$, where N is the number of points and M is the number of processing elements. An implementation of a 64-point FFT machine using the proposed architecture is presented. The architecture compares favorably to other FFT pipelines. Future research should focus on reducing power consumption of the FFT pipeline.

References

[1] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation,* vol. 19, pp. 297-301, 1965.

[2] B. M. Baas, "A low-power high-performance 1024-point FFT processor," *IEEE Journal of Solid-State Circuits*, vol. 34, pp. 380–387, March 1999.

[3] D. Cohen, "Simplified control of FFT hardware," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-24, pp. 577-579, 1976.

[4] M. C. Pease, "Organization of large scale Fourier processors," *JACM*, vol. 16, pp. 474-482, 1969.

[5] L. G. Johnson, "Conflict free memory addressing for dedicated FFT hardware," *IEEE Transactions on Circuits and Systems, II*, vol. 39, pp. 312-316, 1992.

[6] Y. Ma, "An effective memory addressing scheme for FFT processors," *IEEE Transactions on Signal Processing*, vol. 47, pp. 907-911, 1999

[7] Y. Ma and L. Wanhammar, "A hardware efficient control of memory addressing for high-performance FFT processors," *IEEE Transactions on Signal Processing*, vol. 48, pp. 917-921, 2000.

[8] B. M. Baas, "A generalized cached-FFT algorithm," *IEEE International Conference on Acoustic, Speech and Signal Processing*, 18-23 March 2005 pp. v/89 - v/92.

[9] G. Zhong, F. Xu and A. N. Willson, Jr., "A power-scalable reconfigurable FFT/IFFT IC based on a multi-processor ring," *IEEE Journal of Solid-State Circuits*, Volume 41, Issue 2, Feb. 2006 pp. 483 - 495

[10] G. Zhong, F. Xu and A. N. Willson, Jr., "An energy-efficient reconfigurable FFT/IFFT processor based on a multi-processor ring," *XII European Signal Processing Conference (EUSIPCO), 2004, Vienna, Austria*. pp. 2023-2026.

[11] H. L. Groginsky and G. A. Works, "A pipelined fast Fourier transform," *IEEE Transactions on Computers,* vol. C-19. pp. 1015-1019, 1970

[12] J. H. McClellan and R. J. Purdy, "Applications of Digital Signal Processing to Radar," in A. V. Oppenheim, ed., *Applications of Digital Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, pp. 239-329, 1978

[13] E. E. Swartzlander, Jr., "Systolic FFT Processors," in W. Moore, A. McCabe and R. Urquhart, eds., *Systolic Arrays*, Boston: Adam Hilger, 1987, pp. 133-140.

[14] S. M. Currie, P. R. Schumacher, B. K. Gilbert, E. E. Swartzlander, Jr. and B. A. Randall, "Implementation of a Single Chip, Pipelined, Complex, One-Dimensional Fast Fourier Transform in 0.25 µm Bulk CMOS*," IEEE International Conference on Application-Specific Systems, Architectures and Processors*, 2002, pp. 335-343.

[15] S. He and M. Torkelson, "Designing pipeline FFT processor for OFDM (de)modulation," *Proc. of URSI International Symposium on Signals, Systems, and Electronics*, 1998, pp. 257-262

[16] S. He and M. Torkelson. "Design and Implementation of a 1024-point Pipeline FFT Processor," *IEEE Custom Integrated Circuits Conference*, pp. 131–134, May 1998

[17] P.-Y. Tsai, T.-H. Lee and T.-D. Chiueh, "Power-Efficient Continuous-Flow Memory-Based FFT Processor for WiMax OFDM Mode," *International Symposium on Intelligent Signal Processing and Communication Systems (IPACS 2006),* December 12-15, 2006.

[18] A. M. El-Khashab and E. E. Swartzlander, Jr., "The Modular Pipeline Fast Fourier Transform Algorithm and Architecture," *Proceedings of the Thirty-Seventh Asilomar Conference on Signals, Systems, and Computers*, November 9-12, 2003, Pacific Grove, CA, pp. 1463-1467.

[19] A. M. El-Khashab and E. E. Swartzlander, Jr., "A modular pipelined implementation of large fast Fourier transforms," *Proceedings of the Thirty-Sixth Asilomar Conference on Signals, Systems and Computers,* November 3-6, 2002, Pacific Grove, CA, pp. 995 – 999.

[20] A. M. El-Khashab and E. E. Swartzlander, Jr., "An architecture for a radix-4 modular pipeline fast Fourier transform," *IEEE International Conference on Application-Specific Systems, Architectures and Processors*, June 24-26, 2003, pp. 378 – 388

[21] B. J. Mohd, A. Aziz and E. E. Swartzlander, Jr. "The Hazard-Free Superscalar Pipeline Fast Fourier Transform Algorithm and Architecture," *15th Annual IFIP VLSI SoC 2007*, Atlanta, Oct, 2007

[22] J Shen and M Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*, New York: McGraw-Hill, 2005, pp. 27-32.