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

Bassam Jamil Mohd Adnan Aziz Earl E. Swartzlander, Jr.

***Abstract*—This paper examines the superscalar pipeline Fast Fourier Transform algorithm and architecture. The algorithm presents a memory management scheme to prevent memory contention throughout the pipeline stages. The fundamental algorithm, a switch-based FFT pipeline architecture and an example 64-point FFT pipeline are presented. The proposed superscalar architecture substantially improves the FFT processing. 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.**

***Index Terms*— Discrete Fourier Transforms, Pipeline Processing, Memory Management**

I. Introduction

THE FAST FOURIER TRANSFORM ALGORIRTHM, developed by [1], is a standard method for computing the Discrete Fourier Transform (DFT). The FFT algorithm consists of $\log_2 N$ loops; each loop executes N/2 complex operations. The operations in loop i depend on the results from loop i-1 creating potential data dependencies between algorithm loops. This is referred to as inter-stage dependency. The dependencies can be mitigated by use of temporal parallelism in the pipeline architecture where each algorithm loop is mapped to a pipeline stage. The performance of the machine can be further enhanced by exploiting spatial parallelism: processing operations within each stage in parallel. The challenge is to arrange the data in the pipeline memories. A simple mapping to memories results in multiple data elements residing in the same memory and creating structural hazards and pipeline-stalls. This dependency is referred to as an intra-stage dependency. The solution is to rearrange the data in the memory.

A variety of architectures and hardware implementations have been proposed to enhance speed, reduce power and resolve memory contention. One of the earliest implementations of a pipeline FFT was described in [2]. A variety of pipeline FFTs have been surveyed in [11]. Most pipeline FFT realizations use delay lines for data reordering between the processing elements. Although this gives a simple data flow architecture, it causes high power consumption. A memory address generation scheme was proposed by Cohen in [3], which allows parallel organization of memory so that the date 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 number of address bits. A multi-bank memory address assignment for a radix-r FFT was developed in [5]. The memory assignment minimizes the memory size and allows conflict-free simultaneous memory access. Reference [6] developed a fast address generation scheme with hardware cost comparable to the address generation scheme in [3]. Ma and Wanhammar proposed 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. Reference [10] proposed using cache-memory architecture to reduce communication energy between FFT processor and memory.

This paper proposes 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 resolves any 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 paper is organized as follows. Section-II explains the pipeline architecture and analyzes pipeline speedup hazards and optimizations. Section-III discusses hazard conditions and resolutions. It provides a pseudo code of the pipeline memory management algorithm. Section-IV details the design of a 64-point FFT with emphasis on the data movement and storage in the pipeline and memories. Section-V compares the proposed design with other pipeline FFTs.

II. Architecture

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

A. Superscalar Pipeline Architecture

The pipeline architecture of an N-point FFT consists of $\log_2(N)$ stages. Figure 1 shows the block diagram of 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 which have three inputs (a, b, w) and two outputs (c, d) and perform the radix-2 FFT 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 to store twiddle factors. Since the twiddle factors do not change, twiddle factor memories can be implemented as ROMs. There are M ROMs per stage, each with size equal to N/(2*M) words.

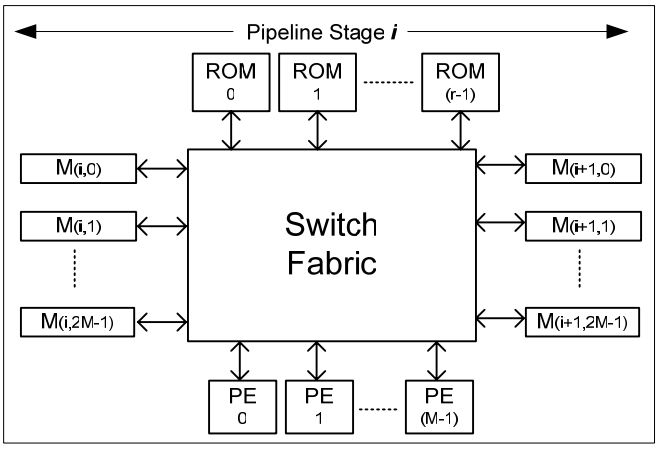

Fig. 1. Block Diagram of the Switch-Based Pipeline Stage

Figure 2 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 [8], 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 can be expressed as: $M*\log_2(N)$. 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 practical value of M is decided by design parameters: speed, area and power.

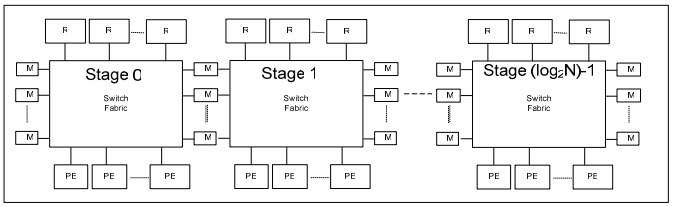

Fig. 2. Overview of Pipeline Architecture

B. *Pipeline Design Optimizations*

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 1.

TABLE 1: NUMBER AND SIZE OF ROMS 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 static, the pipeline connectivity and twiddle factors are static. As a result, the design implementation can be optimized since the connectivity of each stage is predetermined. Figure 3 illustrates the connectivity of 16-point 2-PE pipeline. Furthermore, in many computations the value of twiddle factor is one. A twiddle factor of one reduces the PE computation to add/subtract operation. Also, several PEs executes specific sets of twiddle factors, which can lead to design simplification.

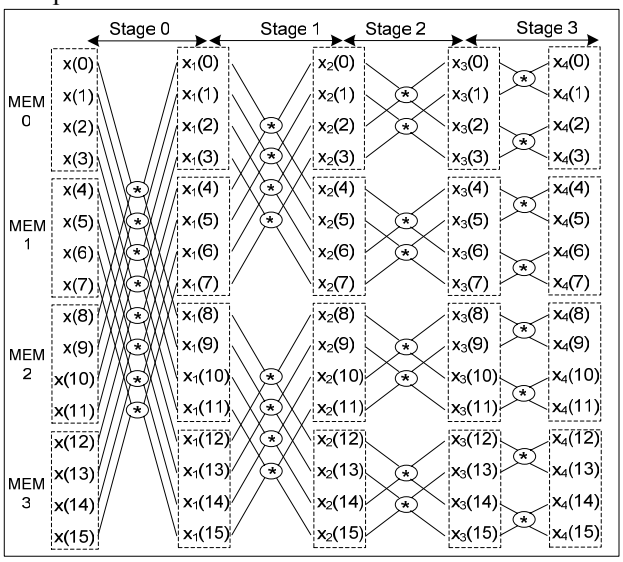

Fig. 3. Pipeline Hazard Example

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 this question: *"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 2. 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 quadratically proportional to S.

TABLE 2: ANALYZING SPEEDUP 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.

C. 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, the contention affects stage 0 to stage $\log_2(N)-\log_2(M)-1$.

Figure 3 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.

III. HAZARD FREE PIPELINE ALGORITHM

The main idea of the algorithm is resolve memory contention in the early stages of the pipeline. First, the condition that causes contention is described and then the hazard free algorithm is described.

A. Detecting Pipeline Hazard

From Figure 3, in stage 0, x(0) and x(8) are 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 3.

TABLE 3: 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 II, 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 3, 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*

B. 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 4):

- **Normal Operation**: Input $x_i(t)$ 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 $x_j(t+1)$.

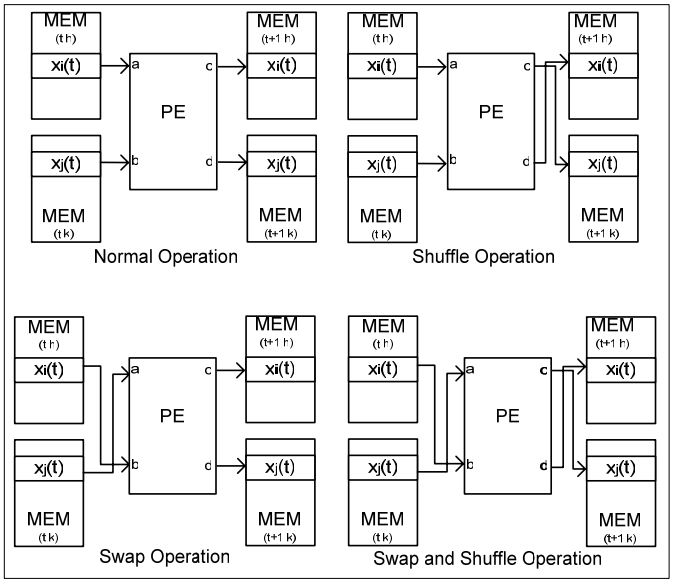

Fig. 4. Pipeline Memory Management Operations

- **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 when inputs are incorrect, the swap operation is performed. A PE operation can have both swap and shuffle memory operations at the same time.

C. Pipeline 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 5. 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.

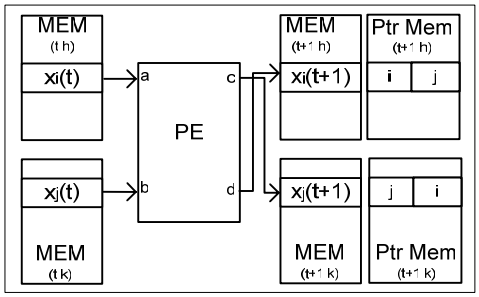

Fig. 5. Tracking Shuffled Data

Another (less flexible) solution is to move data in memory in a methodic way to simplify data tracking in the pipeline. This approach resolves hazards for next stage only. The algorithm can be summarized as follows. For each PE operation:

- If data has been reversed in memory, the PE input is swapped.
- If present data pair will create hazard in the next pipeline stage, the PE results are shuffled.

As a result of reordering data in the pipeline, results from the last stage in the pipeline should be reordered. Below is a detailed pseudo code of the algorithm for swap/shuffle operations.

```
// 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
```

IV. 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 size 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 4 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 5-9 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 10 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 4: Pipeline Stage-0 PE Operand Pairs

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 5: Pipeline Stage-1 PE Operand Pairs

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 6: Pipeline Stage-2 PE Operand Pairs

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 7: Pipeline Stage-3 PE Operand Pairs

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 8: Pipeline Stage-4 PE Operand Pairs

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 9: Pipeline Stage-5 PE Operand Pairs

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 10: PIPELINE MEMORY CONTENTS

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

V. COMPARISON WITH OTHER FFT PIPELINES

Table 11 summarizes features of FFT pipeline architectures discussed in reference [11] and 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 registers (which dissipate high dynamic power) or by RAMs with additional address generation hardware (which increases design complexity). The switch-based pipeline uses SRAM caches, which consume less power than registers and is easier to implement. Moreover, the throughputs of the other pipelines are limited to one (single-path) or a few (multi-path), while the switch based implementation has a throughput of M. Unfortunately, the switch based pipeline requires larger memorys and more hardware in the data path.

TABLE 11: FFT PIPELINE ARCHITECTURES

FFT Pipeline	Multiplier #	Adder #	Memory Size	Speed up
Radix-2 Multi-path Delay Commutator	$2(\log_4 N\text{-}1)$	$4\log_4 N$	$3N/2 - 2$	$\log_2 N$
Radix-2 Single-path Delay Feedback	$2(\log_4 N\text{-}1)$	$4\log_4 N$	$N - 1$	$\log_2 N$
Radix-4 Single-path Delay Feedback	$\log_4 N\text{-}1$	$8\log_4 N$	$N - 1$	$\log_2 N$
Radix-4 Multi-path Delay Commutator	$3(\log_4 N\text{-}1)$	$8\log_4 N$	$5N/2 - 4$	$\log_2 N$
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	$\log_4 N\text{-}1$	$4\log_4 N$	$N - 1$	$\log_2 N$
Switch-Based Pipeline	$M*2(\log_4 N\text{-}1)$	$M*4\log_4 N$	$2*N*(1+\log_2 N)$	$M*\log_2 N$

VI. CONCLUSION AND FUTURE WORK

In this paper we have proposed switch-based architecture for FFT engine implementation. We have also presented an algorithm to predict and resolve memory contentions. As a result the pipeline speedup is $M*\log_2 N$, where N is number of points and M is number of processing elements. An implementation of a 64-point FFT machine using the proposed architecture was presented. The proposed architecture was compared to other FFT pipelines. Future research should focus on reducing power consumption of the FFT pipeline and extending the work done in [7], [9] and [10].

REFERENCES

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

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

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

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

[10] 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.

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