Contention-Free Switch-Based Implementation of 1024-point Radix-2 Fourier Transform Engine

Hani Saleh Bassam Jamil Mohd Adnan Aziz Earl Swartzlander, Jr.
AMD *University of Texas, Electrical and Computer Engineering Department*

Abstract

*This paper examines the use of a switch based architecture to implement a Radix-2 decimation in frequency Fast Fourier Transform Engine. The architecture interconnects M processing elements with 2*M memories. An algorithm to detect and resolve memory access contention is presented. The implementation of 1024-point FFTs with 2 processing elements is discussed in detail, including timing and place-and-route results. The switch based architecture provides a factor of M speedup over a single processing element realization.*

1. Introduction

The Fast Fourier Transform, proposed by [1], is a standard method for computing the Discrete Fourier Transform (DFT). A variety of architectures have been proposed to increase the speed, reduce the power consumption.

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. Baas, in [2], presented a cache FFT algorithm which increases energy efficiency and effectively lowers the power consumption.

A *dual memory* architecture, implemented in [3], uses two memories connected to a digital array signal processor. The programmable array controller generates addresses to memories in a ping-pong fashion.

The *processor array* architecture [4], consists of independent processing elements, with local buffers, which are connected using an interconnect network.

Pipeline FFT architectures, introduced in [5], contain $\log_r N$ blocks; each block consists of delay lines, arithmetic units that implement a radix-r FFT butterfly operation and ROMs for twiddle factors. A variety of pipeline FFTs have been implemented [6-9]. Most pipeline FFT realizations use delay lines for data reordering between the processing elements. Although this gives simple data flow architecture, it causes high power consumption.

Several techniques have been proposed for memory address generation. Cohen described an address generation scheme based on a counter, shifters and rotators [14]. It allows parallel organization of memory so that the data used at any instant reside in different memories. Pease proposed dividing the memory into sub-memories for overlapping the access [15]. 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 [16]. The memory assignment minimizes the memory size and allows conflict-free simultaneous memory access. Ma developed a fast address generation scheme [17] with hardware cost comparable to the address generation scheme in [14]. Ma and Wanhammar proposed an address generation scheme in [18] to reduce the hardware complexity and power consumption.

This paper describes a scalable switch-based architecture to implement a radix-2 decimation in frequency N-point FFT engine. The switch fabric interconnects processing elements (PEs) with single-port memories and ROMs. The architecture concentrates the connectivity in the switch fabric, which enhances the power, area and timing. Moreover, unlike pipeline FFTs, the switch-based architecture does not use delay lines for data reordering, instead, RAMs are used for temporary data storage resulting in a significant reduction in power consumption. To detect and resolve memory contention (which causes performance degradation), an algorithm to eliminate memory hazards is presented. Finally, the paper presents the

1-4244-1258-7/07/$25.00 ©2007 IEEE

implementation of a 1024-point FFT using two PEs that perform radix-2 butterfly operations. The architecture and algorithm can be easily extended to other values of *M* and other radices; for example an architecture composed of (8, 16, 32, …) RAMs, (4, 8, 16, …) ROMs and (4, 8, 16, …) processing elements (PEs). Furthermore, the authors believe that the algorithm could be modified to substantially reduce the number of needed ROMs but this enhancement is outside the scope of this paper.

2. Switch Based Architecture

The switch based architecture is shown on Figure 1. It consists of a switch fabric, *M* processing elements (PEs), *2M* memories and *M* read only memories. It is assumed that $M=2^k$, where *k* is a positive integer. Each PE has three inputs (a, b, w) and two outputs (c, d) and performs a radix-2 decimation in frequency butterfly operation:

$$c = a + b$$

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

All of the data (*a, b, c, d* and *w*) are complex pairs. Data *(a, b)* are the inputs, *w* is the twiddle factor and *(c, d)* are the outputs.

The memory elements store the inputs, intermediate results and the final results. The memories shown as MEMs on Figure 1 are read/write random access memories (e.g., RAM, cache or register files), with size equal to at least *N/(2*M)*. The other type of memory elements stores the pre-computed twiddle factors shown as ROMs in Figure 1. In spite of the name, these memories may be implemented with either read only or read/write memories. The size of each ROM is *N/(2*M)*. The PEs perform single radix-2 butterfly operations. The FFT algorithm consists of $\log_2 N$ stages; each stage consists of N/2 radix-2 butterfly operations. Figure 2 shows an example for N=16 and M=2. The architecture is designed to exploit operation-level parallelism in each stage.

3. Memory Contention Algorithm

Memory contention occurs when a PE requests two accesses to a given memory at the same time. 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 $\log_2(N)-\log_2(M)-1$. The 16-point decimation in frequency FFT shown on Figure 2 demonstrates memory contention. Stages 0 and 1 have no contention, but contention occurs in stages 2 and 3. In stage 2 the inputs for the top PE are $x_2(0)$ and $x_2(2)$, both of which reside in MEM 0. In stage 3 the inputs for the top PE are $x_3(0)$ and $x_3(1)$, both of which reside in MEM0.

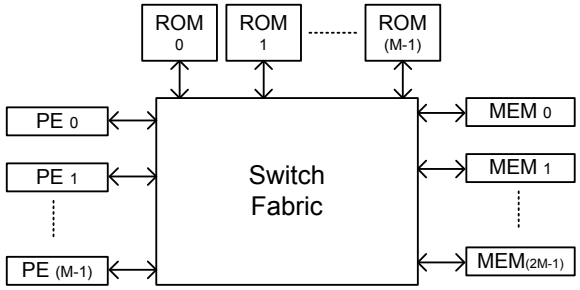

Figure 1. Switch-based architecture

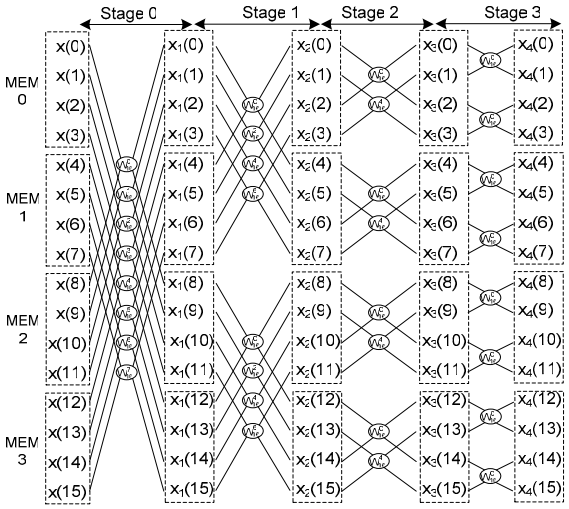

Figure 2. 16-point DiF FFT

3.1. Predicting Memory Contention

Define the stage distance as the index delta of data feeding PEs in each stage. The stage distance for a 16-point decimation in frequency FFT is 8 in stage 0, 4 in stage 1, 2 in stage 2 and 1 in stage 3. In general, for an N-point decimation in frequency 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. Since memory size is equal to

*N/(2*M)*, memory contention occurs in stage i if the following condition is satisfied:

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

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 2, 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 $(0, 4)_1$.

A pair $(i, j)_t$ could be a hazard pair if:

1) *t* is a hazard stage
2) The bit wise Exclusive-OR of addresses i and j is less than *N/(2*M)*.

For example, the address pair $(5, 7)_2$ is a hazard pair since: $5_{10} \oplus 7_{10} = 101_2 \oplus 111_2 = 010_2 < 4$

On the other hand, address pair $(0, 4)_1$ is a safe pair because: $0_{10} \oplus 4_{10} = 000_2 \oplus 100_2 = 100_2$

Furthermore, a stronger definition is proposed to determine hazard pairs. A pair $(i, j)_t$ is 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:

$Stage\text{-}2\ distance = 2_{10}$

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

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

3.2. 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 3):

- **Normal Operation**: Input x_i and x_j are provided to the first and second inputs (a and b) of the PE. The results (c and d) are saved in x_i and x_j.
- **Shuffle Operation** affects how PE results are saved back in memory. In shuffle operation, the results (c and d) are saved in x_j and x_i.
- **Swap Operation**: The swap operation affects the order of PE inputs. In swap operation, x_i is provided to b and x_j is provided to a.

If the algorithm detects a case when inputs are incorrect, the swap operation is performed. As shown on Figure 3, a PE operation can have both swap and shuffle memory operations at the same time.

3.3. 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). 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 the time for 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 methodic way to simplify data tracking in the pipeline. This approach resolves hazards for next stage only.

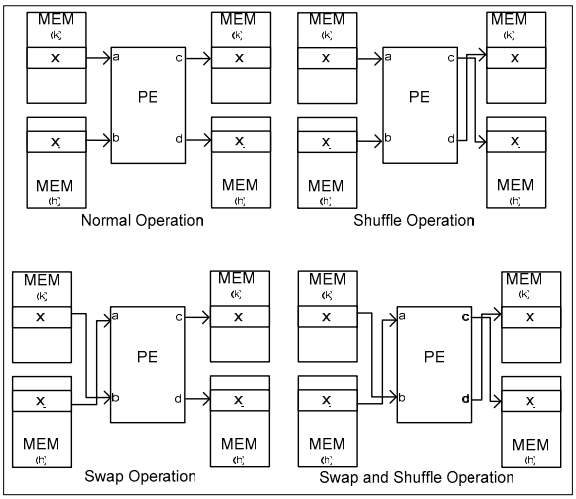

Figure 3. Memory Management Operations

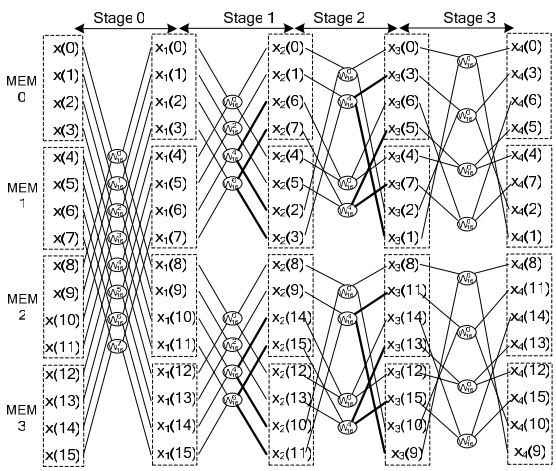

Figure 4. Contention-free 16-point FFT

The algorithm can be summarized as follows. For each PE operation:

- If data has been reversed in memory, the PE input is swapped.
- If the present data pair will create a 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 should be reordered. Figure 4 shows the intermediate and final memory locations for contention free 16-point FFT. Compare the following observations to those made in Figure 2:

- In Stage-2 the inputs for the top butterfly are $x_2(0)$ and $x_2(2)$. There is no contention since $x_2(0)$ and $x_2(2)$ reside in MEM 0 and MEM 1 respectively.
- Similarly, in Stage-3 the inputs for the top butterfly are $x_3(0)$ and $x_3(1)$ which reside in MEM 0 and MEM 1 respectively.

Table 1 summarizes the definition of the variables used in the algorithm pseudo code.

Table 1. Variables Definition

Name	Definition
N	Number of FFT points
NoPE	Number of PEs

Below is a detailed pseudo code of the algorithm for swap/shuffle operations.

```
// Preparation Step
Number_O_Stages   = log2(N)
Cycles_Per_Stage  = N/(2*NoPE)
Memory_Size       = N/2^(NoPE+1)
Safe_Stage        = log2(NoPE)
// Start main nester loops
for Current_Stage=0 to (Number_O_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_O_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
```

4. Implementation of a 1024-Point FFT

Table 2 summarizes the design specification of the FFT implementation. The block diagram of the FFT engine is shown in Figure 5. Multiplexers are used to

route the input and output data to and from the butterflies. The two butterflies are used to perform the radix-2 decimation in frequency FFT butterfly operation.

Table 2. Design Specifications

Item	Details
FFT Algorithm	Radix-2, Decimation-in-Frequency
N	1024 points
Format	Fixed-point (int.frac): 16.16
Number of PEs	2
Number of RAMs	4
RAM size	256
RAM word width	32-bit
Number of ROMs	2
ROM size	512
ROM word width	32-bit
Frequency	1.4GHz

4.1. Placement and Route

The FFT core was designed using Verilog-HDL and implemented using an automatic synthesize, place and route approach. The RAM/ROM memories were modeled as hard macros (which is the industry standard for implementing data arrays), the area occupied was estimated based on guidelines presented in [10], the timing models for the data-arrays was generated using QTM methodology presented in [11], for write the data setup time for a typical D-flop in this library was used, while for read the RAM/ROM memories were given a full cycle to generate the data after latching the address in. The memories were assumed to be high performance memories and will be able to meet the intended timing if designed in similar fashion to [12] which presented a 65nm SRAM that runs at 3 GHz and [13] which presented a 65nm SRAM that ran above 4.0 GHz. A very high performance 65nm process was used for the implementation with standard cell library carefully designed for high speed applications. The routing was limited to metal layer-7. Table 3 shows post-synthesis cell count. Figure 6 shows the floorplan of the memory macros, and the standard cells used to implement the control, multiplexers and the processing elements. The bar graph on the left and bottom edges show the placement congestion distribution. Figure 7 shows the finished FFT core. The FFT core occupied an area of 451µm by 226µm, of which the memory macros occupy 43,359 μm^2 (42.5%) while the standard cells occupy 15,570 um^2 (15.3%) with a total utilization of ~58%.

Table 3. Post Synthesis Cell Count

Cell (x1-equiv)	Number of Instances
Inv	1737
Xor	157
Bufs	1629
nand2	780
nor2	430
DFF	131
Oai	425
Aoi	229
mux2	145
256x32 RAM	4
512x32 ROM	2
Total	**5663**

4.2. Timing

The placed, routed and tapeout ready FFT core meets timing for setup and hold at 1.43 GHZ (~700ps period) using industry standard STA tools, an extracted and back-annotated netlist was analyzed. At this cycle speed, a 1024-point FFT will complete in (2 cycles for RAM read/write * 256 cycles to loop through all of the memories contents * 10 stages to generate the final FFT results) = 5120 cycles. At a 700ps cycle time, this translates to 5120* 0.7 ns = 3.584µs. If a dual-ported RAMs are used, or if a register file approach was used to realize the RAMs to achieve reading at the positive edge and writing at the negative edges then the 1024-point FFT will take (1 cycle RAM read/write*256 cycles to loop through all of the memories contents * 10 stages) = 2560 cycles. With a 700ps cycle time this translates to (2560* 0.7 ns) = 1.792µs.

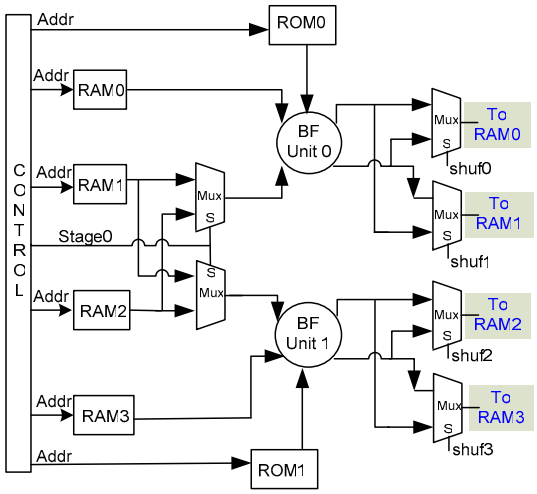

Figure 5. Block diagram the FFT engine

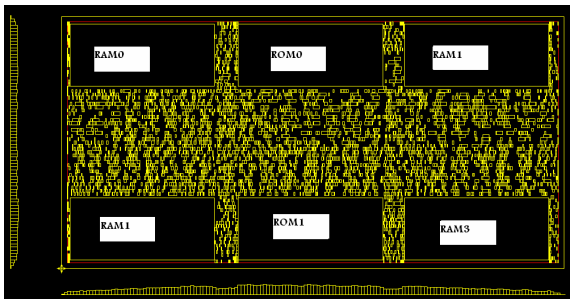

Figure 6 FFT core placement

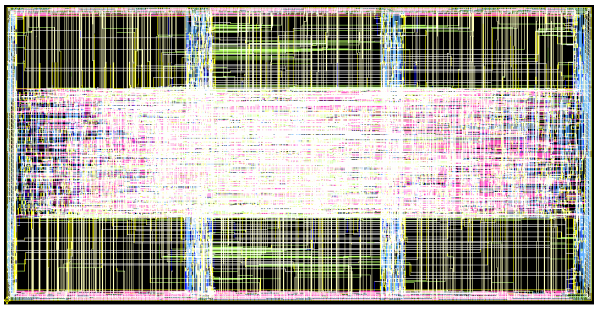

Figure 7. FFT core routing

5. Conclusions

We have presented a switch-based architecture to implemented a radix-2 decimation in frequency N-point FFT engine. An algorithm to detect and resolve memory contentions has been described. We have demonstrated the architectural and algorithmic ideas in the 1024-point FFT implementation. Future research can focus on reducing power consumption of the FFT engine and extending the work done in [2] and [18]. Moving data between PEs and memories consumes considerable switching energy due to the charging and discharging of long-buses and memory banks. Minimizing data movement using caches or registers should be examined. PE execution is also major power contributor. Techniques to reduce the size and number of PEs should be also examined. One idea is to study the effect of internally pipelining the PE to reduce PE power.

6. 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] B. M. Baas, “A low-power, high-performance, 1024-point FFT processor,” IEEE Journal of Solid-State Circuits, vol.34, no. 3, pp. 380–387, March 1999.

[3] Magar, S., S. Shen, G. Luikuo, M. Fleming, and R. Aguilar, “An Application Specific DSP Chip Set for 100 MHz Data Rates,” International Conference on Acoustics, Speech, and Signal Processing, vol. 4, pp. 1989–1992, April 1988.

[4] O’Brien, J., J. Mather, and B. Holland, “A 200 MIPS Single-Chip 1K FFT Processor,” IEEE International Solid-State Circuits Conference, pp. 166–167, 327, 1989.

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

[6] E.H. Wold and A.M. Despain, “Pipeline and parallel-pipeline FFI7 processors for VLSI implementation,” IEEE Transactions on Computers, vol. C-33, pp. 414-426, May 1984.

[7] G. Bi and E. V. Jones, “A pipelined FFT processor for word sequential data,” IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. 37, pp. 1982-1985, December 1989.

[8] E. E. Swartzlander, V. K. Jain, and H. Hikawa, “A radix 8 wafer scale FFT processor,” Journal of VLSI Signal Processing, vol. 4, pp. 165-176, May 1992.

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

[10] A. Steegen, et al., “65nm CMOS technology for low power applications,” IEEE International Electron Devices Meeting Technical Digest, pp. 64- 67, 2005.

[11] Synopsys PrimeTime User manuals in addition to Synopsys solvent article number “010857”.

[12] K. Zhang, et al., "A 3-GHz 70Mb SRAM in 65nm CMOS Technology with Integrated Column-Based Dynamic Power Supply," ISSCC 2005, pp. 474-475, 2005.

[13] Antonio R. Pelella, et al., "A 8Kb Domino Read SRAM with Hit Logic and Parity Checker," ESSCIRC, Grenoble, France, pp. 359-362, 2005.

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

[15] M. C. Pease, “Organization of large scale Fourier processors,” Journal of the ACM, vol. 16, pp. 474-482, 1969.

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

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

[18] 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.