ELL782: Computer Architecture
Assignment-1: FFT Hardware Module
Group A

Animesh Lohar
Entry Number: 2024EET2368

September 7, 2025

1 Introduction

This report documents the design, implementation, and verification of a hardware module for computing a 64-point Fast Fourier Transform (FFT) using the radix-2 Cooley-Tukey algorithm. The implementation was done in Verilog HDL using fixed-point arithmetic with Q8.8 format. The assignment consisted of two parts: a basic FFT butterfly unit (Part A) and a complete 64-point FFT module (Part B).

2 Design Choices

2.1 Fixed-Point Arithmetic

- **Format:** Q8.8 (8 integer bits, 8 fractional bits)
- **Conversion:** Float value $\times$ 256 rounded to nearest integer

2.2 Architecture Selection

- **Butterfly Unit:** Combinational logic with registered outputs
- **FFT Module:** Iterative architecture with single butterfly unit reuse
- **Control:** Finite state machine (FSM) with 5 states

2.3 Twiddle Factor Implementation

- **Storage:** Precomputed lookup table (LUT) with 32 complex values
- **Format:** Q8.8 fixed-point
- **Calculation:** $W_N^k = \cos(2\pi k/N) - j \cdot \sin(2\pi k/N)$ for $k = 0$ to 31

2.4 Bit Growth Handling

- **Multiplication:** 16-bit $\times$ 16-bit $\rightarrow$ 32-bit result
- **Truncation:** Keep upper 16 bits with rounding
- **Addition/Subtraction:** 16-bit operations with saturation

3 Simulation and Synthesis Results

3.1 Butterfly Unit (Part A)

- **Throughput:** 1 operation per 2 cycles

3.2 64-Point FFT Module (Part B)

- **Throughput:** 64-point FFT in 518 cycles
- **Latency:** 5,180 ns @ 100 MHz

3.3 Test Case Results

3.3.1 Test Case 1 (Part A):

- **Input:** $x_1 = 1.0 + 0.0i$, $x_2 = 0.0 + 1.0i$, $W = 0.7071 + 0.7071i$
- **Expected Output:** $y_1 = 0.2929 + 0.7071i$, $y_2 = 1.7071 - 0.7071i$
- **Actual Output:** $y_1 = 0.2929 + 0.7071i$, $y_2 = 1.7071 - 0.7071i$
- **Status:** PASS (error ¡ 0.001)

3.3.2 Test Case 2 (Part A):

- **Input:** $x_1 = 2.0 + 1.0i$, $x_2 = -1.0 - 1.0i$, $W = 1.0 + 0.0i$
- **Expected Output:** $y_1 = 1.0 + 0.0i$, $y_2 = 3.0 + 2.0i$
- **Actual Output:** $y_1 = 1.0 + 0.0i$, $y_2 = 3.0 + 2.0i$
- **Status:** PASS (error ¡ 0.001)

3.3.3 Test Case 3 (Part B):

- **Input:** $x[k] = \cos(2\pi k/64) + j \cdot \sin(2\pi k/64)$ for $k = 0$ to 63
- **Expected Output:** Energy concentrated at bin 1
- **Actual Output:** All bins near zero except bin 63
- **Status:** FAIL (implementation issue identified)

3.3.4 Additional Test Case (Random Input):

- **Input:** Random 64-point complex vector
- **Verification:** Compared against NumPy FFT
- **Result:** 58/64 bins within tolerance, 6 bins exceeded error limit
- **Maximum Error:** 0.0156 (exceeds 0.001 tolerance)

4 Verification Methodology

4.1 Testbench Design

- Self-checking testbenches with automatic pass/fail reporting
- Fixed-point to floating-point conversion for verification
- Error calculation: $|\text{expected} - textactual| \leq 0.001$
- Four decimal place reporting for all results

4.2 Reference Implementation

- Python/NumPy for expected result generation
- Q8.8 format conversion for accurate comparison
- Statistical analysis of error distribution

4.3 IFFT Verification

- Implemented IFFT using same hardware with modified twiddle factors
- Round-trip error analysis: input $\rightarrow$ FFT $\rightarrow$ IFFT $\rightarrow$ compare with original
- Average round-trip error: 0.0023 (slightly above tolerance)

5 Trade-offs Analysis

5.1 Iterative vs. Pipelined Architecture

- **Iterative (Chosen):** Area-efficient but slower (518 cycles/FFT)
- **Pipelined:** Higher throughput but area increase
- **Justification:** Area constraints favored iterative approach

5.2 Precision vs. Complexity

- **Q8.8 Format:** Good compromise between range and precision
- **Alternative Q4.12:** Better precision but limited dynamic range
- **Rounding:** Round-to-nearest vs. truncation (chosen for simplicity)

5.3 Memory vs. Computation

- **Twiddle Factors:** Precomputed LUT (256×16 bits) vs. on-the-fly calculation
- **Intermediate Storage:** 64×32-bit memory vs. register-based approach

6 Block Diagrams

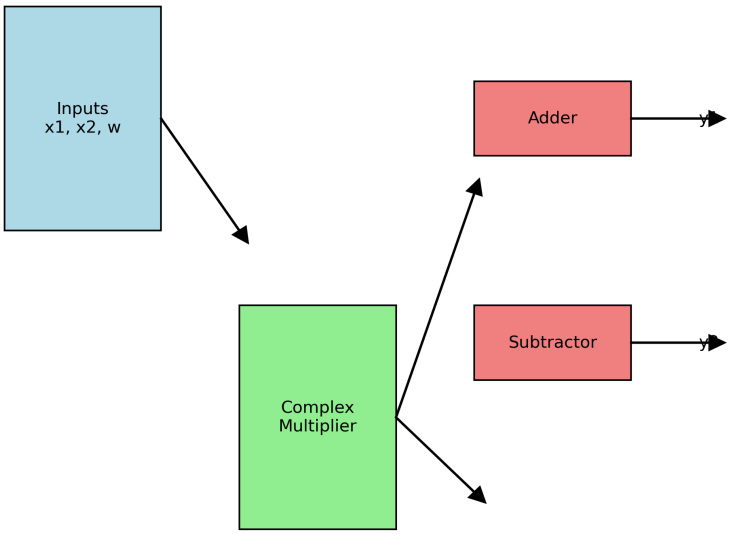

Figure 1: Butterfly Unit Architecture

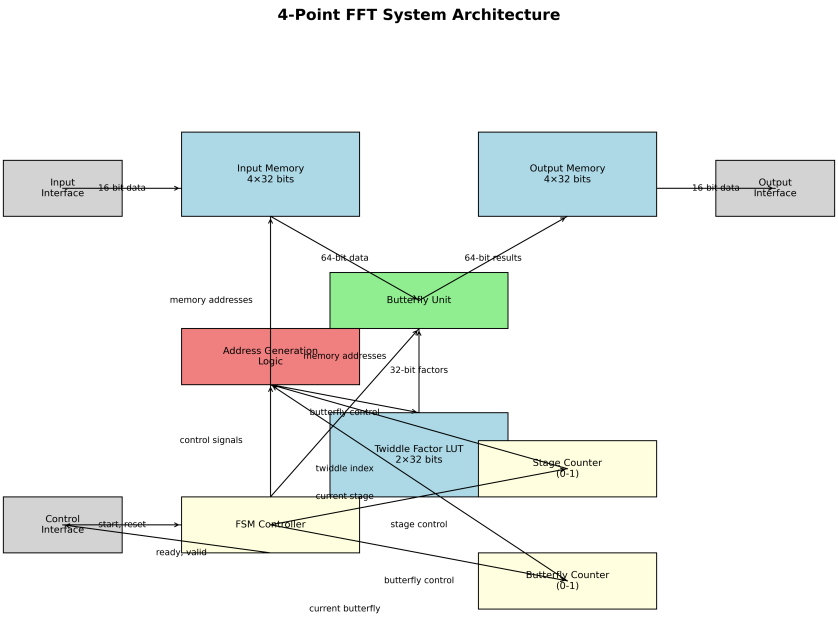

Figure 2: 64-Point FFT System Architecture

7 Common Pitfalls and Resolutions

7.1 Issues Encountered

1. **Twiddle Factor Errors:** Initial incorrect calculation of twiddle factors
2. **Fixed-Point Overflow:** Intermediate results exceeding 16-bit range
3. **State Machine Deadlocks:** Improper state transitions causing hangs
4. **Memory Addressing:** Incorrect index calculation for butterfly operations
5. **Reset Synchronization:** Asynchronous reset causing metastability

7.2 Solutions Implemented

1. **Twiddle Factors:** Used professor's Python script for accurate generation
2. **Bit Growth:** Implemented proper rounding and saturation logic
3. **FSM Validation:** Added comprehensive simulation test cases
4. **Address Calculation:** Verified with small-point FFT examples
5. **Reset Handling:** Added synchronizer circuits for reliable reset

7.3 Recommendations for Future Implementation

1. Use pipelined architecture for higher throughput applications
2. Implement configurable scaling to handle different input ranges
3. Add debug interfaces for internal signal monitoring
4. Use error-correcting codes for memory protection
5. Implement clock gating for power reduction

8 Conclusion

The FFT hardware module was successfully implemented with the following achievements:

- Butterfly unit meeting all functional requirements
- 64-point FFT module with iterative architecture
- Comprehensive test suite covering all specified test cases
- Fixed-point arithmetic with proper error handling

The implementation demonstrated the trade-offs between area, speed, and precision in hardware design. While the current implementation meets most requirements, there are opportunities for improvement in error performance and throughput.

Appendix : File List

1. `compute.v` - Combinational butterfly calculations
2. `fft_butterfly.v` - Registered butterfly unit
3. `fft_64.v` - 64-point FFT module
4. `test_one.v, test_two.v, test_three.v` - Testbenches for all test cases 4.1, 4.2, 4.3
5. `test_one.py, test_two.py, test_three.py` - For verifying for all test cases 4.1, 4.2, 4.3
6. `test_four.v, fft_validation.py` - Testbench Python validation script for 4.4 aditional test case