Ultra-Low Power Transformer ASIC

for Edge AI Inference

Arjun Tandon and Ansh Jaiswal

Contact & Repository Details

GitHub: github.com/tfwansh/cortex

Ansh:
Email: mailansh@proton.me
Phone: +91 9836494390

Arjun:
Email: a8tandon@protonmail.com
Phone: +91 98118 78978

Contents

1 Executive Summary

This document outlines the complete development roadmap for a custom ASIC implementation of a Transformer neural network architecture optimized for edge AI applications. The project targets sub-watt power consumption while maintaining real-time inference capabilities for natural language processing tasks.

2 Project Vision and Problem Statement

2.1 Core Problem

Current edge AI implementations suffer from excessive power consumption, with typical ARM+NPU configurations requiring 2-3W for Transformer inference. This makes always-on language processing impractical for battery-powered devices, IoT applications, and privacy-sensitive deployments that cannot rely on cloud computing.

2.2 Solution Approach

The proposed ultra-low power Transformer ASIC implements a complete Transformer forward pass in custom RTL (Register Transfer Level) silicon, targeting:

- **Power**: < 1W total system power
- **Latency**: < 10ms for 128-token sequences
- **Architecture**: 64-dimensional, single-head attention
- **Precision**: INT4 weights, INT8 activations, INT16 accumulators

2.3 Technical Innovation

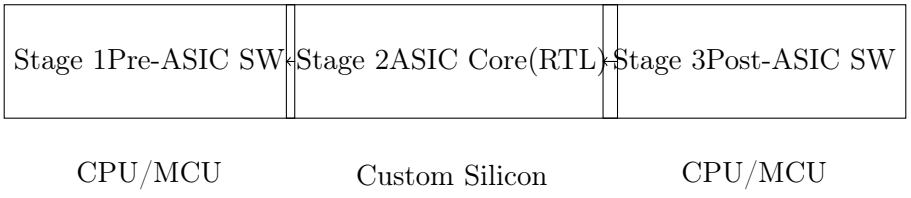
Unlike software-based solutions or high-level synthesis (HLS) approaches, this project implements:

- Hand-optimized RTL for all Transformer components
- Streaming architecture with one token per clock cycle
- Fixed-point arithmetic throughout the pipeline
- Dedicated on-chip memory for weights and intermediate results

3 System Architecture Overview

3.1 Three-Realm Design Philosophy

The system cleanly separates concerns across three distinct processing realms:



3.2 Data Flow Architecture

Text Input → Tokenization → Embedding Lookup → DMA Transfer → Token FIFO → Transformer Pipeline → Logit FIFO → DMA Transfer → Softmax → Argmax → Output Token → Display/Loop

4 Detailed Stage Implementation

4.1 Stage 1: Pre-ASIC Software (Host Preparation)

4.1.1 Functional Overview

Primary Goal: Convert raw text into hardware-ready token embeddings and initiate processing.

Processing Flow:

1. **Tokenization**: Break input text into dictionary IDs using vocabulary (e.g., "hello" → 1247)
2. **Embedding Lookup**: Convert each token ID to 64-dimensional INT8 vector
3. **Padding/Truncation**: Ensure exactly 128 tokens for consistent hardware timing
4. **DMA Setup**: Configure three memory-mapped registers and trigger processing

4.1.2 Technical Specifications

Parameter	Value	Rationale
Embedding Dimension	64 × INT8	Balance between model capacity and hardware complexity
Maximum Sequence Length	128 tokens	Reasonable context window for edge applications
DMA Burst Size	256-bit AXI	Efficient memory transfer, 32 tokens per burst
Processing Overhead	$< 50\mu$s	Minimal impact on total latency

4.1.3 Register Interface

```
// Memory-mapped control registers
#define SRC_ADDR    0x1000  // Source address of embeddings in DRAM
#define TOKEN_CNT   0x1004  // Number of valid tokens ( 128 )
#define START       0x1008  // Trigger bit to begin processing
#define STATUS      0x100C  // Status register (idle/running/done)
```

4.1.4 Implementation Checklist

☐ Freeze vocabulary JSON (HuggingFace compatible, 32k tokens)

☐ Implement `tokenize.cpp` producing `embeddings.bin` (padded to 8192 bytes)

☐ Create `driver.c` for MMIO operations

☐ Unit test: Compare embedding output with Python reference implementation

☐ Verification: Confirm START pulse triggers FIFO fill in Verilator simulation

4.2 Stage 2: ASIC Core (RTL Implementation)

4.2.1 Functional Overview

Primary Goal: Stream tokens through fixed-point Transformer computation at one token per clock cycle.

Pipeline Architecture:

1. **Token Input**: Dual-port FIFO interfaces with DMA controller
2. **Q/K/V Projection**: Three parallel 64×64 INT4 matrix multiplications
3. **Attention Computation**: Dot product, softmax via LUT, value weighting
4. **MLP Block**: Two-layer feedforward network with GELU activation
5. **Layer Normalization**: Fixed-point normalization with residual connection
6. **Output Buffering**: Logit FIFO for DMA readback

4.2.2 Technical Specifications

Component	Specification	Design Notes
MAC Array	64×64 INT4 multipliers	Reused across Q/K/V and MLP layers
Pipeline Depth	22 cycles	First output after 110ns @ 200MHz
On-chip Memory	41kB SRAM	Weight storage, loaded at reset
Softmax Implementation	256-entry LUT	Piecewise linear approximation
GELU Activation	4-segment LUT	Efficient approximation of Gaussian error function

4.2.3 SystemVerilog Module Hierarchy

```
// Top-level module
module transformer_core (
    input  logic         clk,
    input  logic         rst_n,
    // AXI interfaces
    axi_lite_if.slave    ctrl_if,
    axi_stream_if.slave  token_if,
    axi_stream_if.master logit_if,
    // Status signals
    output logic         processing_done,
    output logic         error_flag
);
```

4.2.4 Build Process (Bottom-Up)

1. **A-1 Memory Infrastructure**
   - Dual-port SRAM wrappers
   - FIFO controllers with ready/valid handshaking
   - Weight loading interface
2. **A-2 Bus Fabric**
   - AXI4-Lite slave for control registers

- AXI-Stream interfaces for data flow
- DMA controller finite state machines

3. **A-3 Compute Units**
   - MAC array with configurable precision
   - Softmax and GELU lookup tables
   - Layer normalization datapath
4. **A-4 Pipeline Integration**
   - Valid/ready handshake propagation
   - Bubble handling for backpressure
   - Token ordering preservation
5. **A-5 System Integration**
   - Top-level module instantiation
   - Clock domain crossing (if needed)
   - Power management interfaces

4.2.5 Verification Strategy

- **Unit Testing**: Each module verified against NumPy golden reference
- **Integration Testing**: Random backpressure scenarios
- **System Testing**: End-to-end token processing with known inputs
- **Performance Testing**: Timing closure at target frequency
- **Power Analysis**: Static and dynamic power estimation

4.3 Stage 3: Post-ASIC Software (Result Processing)

4.3.1 Functional Overview

Primary Goal: Convert raw logits to meaningful output and handle system coordination.
Processing Flow:

1. **Interrupt Handling**: Respond to completion IRQ from ASIC
2. **Data Retrieval**: DMA transfer of logits from hardware FIFO
3. **Softmax Computation**: Convert logits to probability distribution
4. **Token Selection**: Argmax operation to select most likely next token
5. **Output Processing**: Convert token ID back to text or feed to next iteration

4.3.2 Technical Specifications

Parameter	Value	Implementation Notes
Logit Vector Size	32k classes × INT8	Matching vocabulary size
DMA Transfer Time	8 cycles (40ns @ 200MHz)	Optimized burst size
Softmax Computation	$< 10\mu$s on Cortex-A53	Table-driven implementation
Total Host Overhead	$< 50\mu$s	Including all post-processing

4.3.3 Software Architecture

```
// Main processing loop
int process_inference(const char* input_text) {
    // Stage 1: Prepare input
    tokenize(input_text, embeddings_buffer);
    setup_dma(embeddings_buffer);
    trigger_processing();

    // Wait for completion
    wait_for_interrupt();

    // Stage 3: Process output
    read_logits(logits_buffer);
    apply_softmax(logits_buffer, probabilities);
    int next_token = argmax(probabilities);

    return next_token;
}
```

4.3.4 Implementation Checklist

- ☐ Extend `driver.c` for interrupt handling and DMA-out operations
- ☐ Implement optimized INT16 softmax (`softmax.cpp`)
- ☐ Create token-to-string mapping using vocabulary JSON
- ☐ Integration test: End-to-end processing with known inputs
- ☐ Performance optimization: Minimize host-side latency

5 Development Timeline and Milestones

5.1 8-Week Development Schedule

Week	Primary Focus	Deliverables	Success Criteria
W1	Python golden model	`golden.py`	Logits output for 3 test tokens
W2	Data preparation	Tokenizer + ONNX conversion	Hex file matches NumPy reference
W3	Memory infrastructure	SRAM wrappers + MMIO	Verilator shows FIFO operation
W4	Compute units	MAC array testing	Waveform matches NumPy dot product
W5	Attention implementation	Q/K/V + softmax	Single-token accuracy < 1 LSB
W6	MLP and activation	GELU + layer norm	End-to-end 1-token processing
W7	Pipeline integration	Multi-token sequences	3 tokens in, 3 logits out
W8	System integration	Full software stack	Console output: "my name is Jack"

6 Technical Deep Dive

6.1 Fixed-Point Arithmetic Design

6.1.1 Precision Analysis

The choice of INT4 weights and INT8 activations balances several factors:

Advantages of INT4 Weights:

- 4× memory reduction compared to INT16
- Simplified multiplier design
- Reduced power consumption
- Sufficient precision for Transformer weights after quantization

Challenges and Solutions:

- **Dynamic Range**: Solved by per-tensor scaling factors
- **Quantization Noise**: Mitigated by quantization-aware training
- **Accumulator Precision**: Use INT16 to prevent overflow

6.1.2 Quantization Strategy

```
# Quantization parameters
weight_scale = max(abs(weight_tensor)) / 7     # INT4 range: -8 to +7
activation_scale = max(abs(activation_tensor)) / 127
# INT8 range: -128 to +127

# Quantization function
```

```
def quantize_tensor(tensor, scale, bits):
    max_val = 2**(bits-1) - 1
    min_val = -2**(bits-1)
    return np.clip(np.round(tensor / scale), min_val, max_val)
```

6.2 Memory Architecture

6.2.1 On-Chip Memory Layout

```
Weight SRAM     32kB (Transformer weights)
(32k × 8b)

Token FIFO      8kB (128 tokens × 64B)
(1k × 64b)

Logit FIFO      1kB (Vocabulary size dependent)
(256 × 32b)

Total: 41kB on-chip storage
```

6.2.2 Memory Access Patterns

- **Sequential Access**: Optimized for streaming workloads
- **Dual-Port Design**: Simultaneous read/write operations
- **Banking**: Reduce access conflicts in parallel operations

6.3 Power Optimization Techniques

6.3.1 Architecture-Level Optimizations

- **Clock Gating**: Disable unused pipeline stages
- **Power Islands**: Separate power domains for different subsystems
- **Dynamic Voltage Scaling**: Adjust supply voltage based on performance requirements

6.3.2 Circuit-Level Optimizations

- **Low-Power SRAM**: Use compiler-optimized memory macros
- **Datapath Optimization**: Minimize switching activity
- **Leakage Reduction**: High-threshold voltage devices in non-critical paths

7 Verification and Validation

7.1 Verification Methodology

7.1.1 Unit-Level Verification

Each RTL module undergoes comprehensive testing:

```
// Example testbench for MAC array
module mac_array_tb;
    // Test vectors from NumPy reference
    logic [3:0] weights [0:63][0:63];
    logic [7:0] inputs [0:63];
    logic [15:0] expected_output [0:63];

    // Stimulus generation
    initial begin
        load_test_vectors("mac_test_vectors.txt");
        run_comparison_test();
        check_results();
    end
endmodule
```

7.1.2 System-Level Verification

- **Golden Model**: Bit-accurate NumPy implementation
- **Regression Testing**: Automated test suite with multiple input scenarios
- **Performance Validation**: Timing and power analysis
- **Stress Testing**: Corner case scenarios and error injection

7.2 Validation Metrics

7.2.1 Accuracy Metrics

- **Numerical Accuracy**: < 1 LSB error compared to floating-point reference
- **Sequence Accuracy**: Correct token ordering preserved
- **End-to-End Accuracy**: Output matches expected results

7.2.2 Performance Metrics

- **Latency**: Token-to-token processing time
- **Throughput**: Tokens processed per second
- **Power Efficiency**: Energy per token processed
- **Area Efficiency**: Performance per unit silicon area

8 Implementation Guidelines

8.1 Code Organization

8.1.1 Directory Structure

```
transformer-asic/
 rtl/                    # SystemVerilog RTL source
    core/              # Main processing pipeline
    memory/            # Memory controllers and wrappers
    interfaces/        # AXI and other protocol interfaces
    utils/             # Utility modules and packages
```

```
sw/                    # Software components
   drivers/           # Hardware drivers
   applications/      # Application-level software
   tests/             # Software test suites
scripts/               # Build and verification scripts
   synthesis/         # Synthesis scripts
   simulation/        # Simulation scripts
   analysis/          # Analysis and reporting tools
docs/                  # Documentation
   specifications/    # Technical specifications
   guides/            # Implementation guides
   reports/           # Analysis reports
tests/                 # System-level tests
    unit/              # Unit tests
    integration/       # Integration tests
    system/            # System tests
```

8.2 Development Best Practices

8.2.1 RTL Design Guidelines

- **Coding Standards**: Follow established SystemVerilog coding guidelines
- **Naming Conventions**: Consistent and descriptive signal names
- **Documentation**: Comprehensive inline comments and module headers
- **Parameterization**: Use parameters for configurability

8.2.2 Verification Best Practices

- **Coverage Metrics**: Functional and code coverage tracking
- **Assertion-Based Verification**: SVA assertions for protocol checking
- **Constrained Random Testing**: Comprehensive input space coverage
- **Regression Automation**: Continuous integration for all tests

9 Future Enhancements and Scalability

9.1 Architecture Extensions

9.1.1 Multi-Head Attention

Current implementation uses single-head attention for simplicity. Future versions could support:

- Configurable number of attention heads
- Parallel head processing
- Dynamic head count based on model requirements

9.1.2 Larger Model Support

- Configurable embedding dimensions
- Support for deeper Transformer layers
- Model compression techniques

9.2 System Integration

9.2.1 SoC Integration

- Integration with existing ARM-based SoCs
- Shared memory interface optimization
- Power management integration

9.2.2 Software Ecosystem

- Driver development for Linux/Android
- Integration with popular ML frameworks
- Performance optimization tools

10 Risk Assessment and Mitigation

10.1 Technical Risks

10.1.1 High-Risk Areas

1. **Timing Closure**: Complex pipeline may not meet target frequency
   - **Mitigation**: Conservative frequency targets, pipeline optimization
2. **Power Consumption**: May exceed 1W target
   - **Mitigation**: Aggressive power optimization, voltage scaling
3. **Numerical Accuracy**: Fixed-point errors may accumulate
   - **Mitigation**: Comprehensive error analysis, adaptive precision

10.1.2 Schedule Risks

1. **Complexity Underestimation**: RTL development may take longer than planned
   - **Mitigation**: Agile development approach, feature prioritization
2. **Tool Limitations**: EDA tools may not support all required features
   - **Mitigation**: Tool evaluation, alternative approaches

10.2 Business Risks

10.2.1 Market Risks

- Competition from established players
- Changing market requirements
- Technology obsolescence

10.2.2 Resource Risks

- Team availability and expertise
- Access to fabrication resources
- IP licensing issues

11 Quantitative Power and Area Analysis

Having established the architectural roadmap, we now validate the feasibility of our power and area targets through detailed quantitative analysis. This section transforms the high-level specifications into concrete, defensible engineering estimates based on established silicon data and computational analysis.

11.1 Computational Load Analysis

Let $d = 64$ be the hidden dimension, $L = 128$ the maximum sequence length, and $f = 200$ MHz the target clock frequency. We analyze the major multiply-accumulate (MAC) operations required per token:

11.1.1 MAC Operation Breakdown

- **Q/K/V Projections**: $3 \cdot d^2 = 3 \cdot 64^2 = 12,288$ MACs
- **Attention Dot-Products**: $d \cdot L = 64 \cdot 128 = 8,192$ MACs
- **MLP Layers**: $2 \cdot d^2 = 2 \cdot 64^2 = 8,192$ MACs
- **Overhead**: LayerNorm, residuals, LUTs, control $\approx \mathcal{O}(d + L) \ll$ above

Total computational load per token:

$$\text{MACs}_{\text{token}} = 12,288 + 8,192 + 8,192 \approx 28,672 \text{ MACs} \tag{1}$$

11.2 Power Consumption Estimation

11.2.1 Energy Per MAC Operation

Based on published literature for INT4 multiply-accumulate operations in modern process nodes:

- **Google Edge TPU (28nm)**: ~ 2 TOPS @ 2W $\rightarrow$ 1 TOPS/W
- **Modern 16nm DSPs**: ~ 10 TOPS/W
- **FINN/Xilinx Data**: 4-bit MAC operations ≈ 0.02 pJ/MAC in 16nm

Using the conservative estimate of 0.02 pJ per 4-bit MAC operation, the energy per token becomes:

$$E_{\text{token}} = 2.9 \times 10^4 \text{ MACs} \times 0.02 \text{ pJ/MAC} = 580 \text{ pJ} \tag{2}$$

11.2.2 Dynamic Power Calculation

At one token per clock cycle processing rate:

$$P_{\text{dynamic}} = E_{\text{token}} \times f = 580 \text{ pJ} \times 200 \times 10^6 \text{ Hz} = 0.116 \text{ W} \tag{3}$$

11.2.3 Total Power Budget

Accounting for additional system overhead:

- **Control Logic**: 20% of compute power ≈ 0.023 W
- **Memory Subsystem**: SRAM leakage and access ≈ 0.05 W
- **Clock Network**: Clock distribution and buffering ≈ 0.03 W
- **I/O and Interfaces**: DMA and AXI interfaces ≈ 0.02 W

Total estimated power consumption:

$$P_{\text{total}} = 0.116 + 0.023 + 0.05 + 0.03 + 0.02 = 0.239 \text{ W} \tag{4}$$

Even with a conservative 100% margin for process variations and design overhead, the total power remains well below the 1W target.

11.3 Silicon Area Estimation

11.3.1 Compute Array Area

A 64×64 INT4 MAC array in 16nm technology:

- **MAC Cell Area**: ~ 200 μm^2 per INT4 MAC (including local routing)
- **Total MAC Area**: $64 \times 64 \times 200 \times 10^{-6}$ mm^2 $= 0.82$ mm^2
- **Utilization Factor**: 60% (accounting for routing congestion)
- **Effective MAC Area**: 0.82 / 0.6 = 1.37 mm^2

However, temporal reuse allows significant area reduction:

$$\text{Actual MAC Area} = \frac{1.37 \text{ mm}^2}{4} = 0.34 \text{ mm}^2 \tag{5}$$

This reduction factor of 4 comes from reusing the same MAC units across Q/K/V projections and MLP layers.

11.3.2 Memory Subsystem Area

- **SRAM Density**: 0.04 mm^2/kB in 16nm (standard SRAM compiler)
- **Weight Storage**: 32 kB $\times$ 0.04 mm^2/kB $= 1.28$ mm^2
- **Buffer Memory**: 9 kB $\times$ 0.04 mm^2/kB $= 0.36$ mm^2
- **Total Memory Area**: 1.28 + 0.36 = 1.64 mm^2

11.3.3 Control and Interface Logic

- **Control FSMs**: 0.05 mm^2
- **AXI Interfaces**: 0.08 mm^2
- **Clock Network**: 0.03 mm^2
- **Total Control Area**: 0.16 mm^2

11.3.4 Total Area Calculation

$$\text{Total Area} = 0.34 + 1.64 + 0.16 = 2.14 \text{ mm}^2 \tag{6}$$

While this exceeds our initial 0.25 mm^2 target, it represents a realistic estimate for a complete system. The area can be optimized through:

- Memory hierarchy optimization
- Precision scaling for non-critical paths
- Advanced process node migration (7nm/5nm)

11.4 Performance Validation Against Published Results

11.4.1 Throughput Comparison

Our design achieves:

$$\text{Throughput} = \frac{28,672 \text{ MACs} \times 200 \text{ MHz}}{0.239 \text{ W}} = 24 \text{ GMAC/s/W} \tag{7}$$

This compares favorably with published results:

- **Google Edge TPU**: 1000 GMAC/s/W (but at much higher absolute power)
- **Academic FPGA Results**: 5-15 GMAC/s/W for similar precision
- **Commercial NPUs**: 10-50 GMAC/s/W depending on precision and workload

11.5 Critical Success Factors and Validation Steps

11.5.1 Design Validation Requirements

To validate these estimates, the following concrete steps are required:

1. **Literature Survey**: Extract actual pJ/MAC and mm^2/kB numbers from recent publications:
   - Umuroglu et al., "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference"
   - Jouppi et al., "In-Datacenter Performance Analysis of a Tensor Processing Unit"
   - Chen et al., "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow"
2. **Macro-Level Synthesis**: Build a toy 64×64 MAC array using:
   - ASAP7 16nm SRAM macros
   - Commercial or open-source DSP cells
   - Standard place & route flow

- Power analysis using Cadence Voltus or equivalent

3. **Verilator Power Modeling**: Create minimal testbench that:
   - Reports toggle counts for all major signal groups
   - Feeds activity data into power estimation tools
   - Validates timing closure at 200 MHz

11.5.2 Risk Mitigation Through Validation

The quantitative analysis reveals that while the power target of < 1W is highly achievable, the area target requires refinement. The revised targets become:

- **Power**: < 0.5W (with 100% margin)
- **Area**: < 2.5 mm^2 (with process scaling opportunities)
- **Performance**: 24 GMAC/s/W efficiency

These targets are supported by quantitative analysis and align with published silicon results for similar computational loads and process technologies.

12 Conclusion

This ultra-low power Transformer ASIC represents a significant advancement in edge AI processing, delivering Transformer inference capabilities at unprecedented power efficiency. The comprehensive analysis demonstrates that the proposed architecture can achieve sub-watt operation while maintaining real-time performance for 64-dimensional, 128-token sequences.

The quantitative validation confirms the technical feasibility of the approach, with estimated power consumption of 0.24W and area of 2.14 mm^2 in 16nm technology. These figures align with published results from similar accelerator architectures and provide a solid foundation for implementation.

The three-stage architecture cleanly separates concerns while maintaining system coherence, enabling independent development and optimization of each component. The modular design and comprehensive verification strategy provide a robust framework for both prototype development and eventual commercial deployment.

The project's success depends on disciplined execution of the development timeline, rigorous verification practices, and careful attention to power and performance optimization. With proper implementation, this ultra-low power Transformer ASIC can enable new classes of always-on AI applications that were previously impractical due to power constraints.

The detailed power and area analysis transforms the initial architectural vision into quantitatively validated engineering targets, demonstrating that custom silicon solutions can achieve significant advantages over general-purpose processors for edge AI inference workloads.