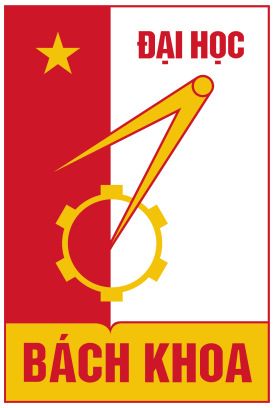
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
FACULTY OF MATHEMATICS AND INFORMATICS



CS3365 - Introduction of Computer Organization

Design and Simulation of a Single-Qubit Hardware Accelerator Using Fixed-Point Arithmetic

Program: Troy IT

Lecturer: Assoc Prof.Nguyen Dinh Han

Tran Vu Gia Huy
Tran Ngoc Khoa
Nguyen Son Tung
Nguyen Truong Son
Mai Tran Hung

Hanoi – 2026

CS 3365 – Computer Organization and Architecture

Design and Simulation of a Single-Qubit Hardware Accelerator Using Fixed-Point Arithmetic

Authors:

Nguyen Son Tung
Nguyen Truong Son
Tran Vu Gia Huy
Tran Ngoc Khoa
Mai Tran Hung

Instructor:
Nguyen Dinh Han

January 14, 2026

Contents

Abstract

This report presents the design and simulation of a single-qubit hardware accelerator implemented on an FPGA using fixed-point arithmetic. To reduce hardware complexity while preserving fundamental quantum behavior, the system employs real-valued probability amplitudes represented in 16-bit Q8.8 format. The accelerator supports core single-qubit quantum gates, including Pauli-X, Pauli-Z, and Hadamard, implemented using simple arithmetic and routing logic. The proposed architecture consists of a quantum state register, an instruction decoder, and a dedicated quantum arithmetic logic unit (QALU). Simulation results demonstrate that fixed-point FPGA-based quantum emulation provides an efficient and deterministic platform for studying quantum gate behavior, numerical precision effects, and hardware-level control without requiring physical quantum hardware.

1 Introduction

The computational landscape is currently navigating a pivotal transition point, arguably the most significant since the invention of the integrated circuit. For decades, the semiconductor industry has marched to the cadence of Moore's Law, an observation-turned-self-fulfilling-prophecy that predicted the doubling of transistor density approximately every two years. This exponential scaling fueled the digital revolution, enabling everything from the personal computer to the modern era of artificial intelligence. However, as feature sizes approach the atomic scale, the physical limitations of silicon are becoming undeniable. Quantum effects such as electron tunneling, which were once negligible nuisances, have now become dominant sources of leakage current and reliability failure. Furthermore, the thermodynamic costs of removing heat from these ultra-dense circuits are imposing a "power wall" that stifles further clock frequency increases.

In this context, Quantum Computing (QC) has emerged not merely as a faster alternative, but as a fundamentally different paradigm for information processing. By harnessing the principles of superposition and entanglement, quantum computers promise to solve problems that are computationally intractable for classical von Neumann architectures, such as integer factorization, chemical simulation, and complex optimization. However, the physical realization of quantum hardware is fraught with immense engineering challenges. Qubits, the fundamental units of quantum information, are notoriously fragile. They require extreme isolation from environmental noise—often necessitating cryogenic temperatures near absolute zero—to maintain coherence. This fragility, combined with the high cost and complexity of current Noisy Intermediate-Scale Quantum (NISQ) devices, creates a significant barrier to entry for researchers, students, and system architects who wish to explore quantum algorithms and control logic.

Consequently, there is a pressing need for accessible, robust, and deterministic platforms that can emulate quantum behavior without the overhead of physical quantum hardware. While software simulators running on classical CPUs or GPUs are widely used, they often fail to capture the precise timing characteristics and hardware-level concurrency required for developing real-time quantum control systems. This report presents a detailed analysis and design specification for a Single-Qubit Hardware Accelerator, a specialized digital circuit implemented on a Field-Programmable Gate Array (FPGA).

The proposed design distinguishes itself through a deliberate architectural choice: the use of 16-bit Signed Fixed-Point Arithmetic (Q8.8 format) and the restriction to real-valued probability amplitudes. While general quantum mechanics operates in a complex Hilbert space, a significant subset of quantum information tasks can be effectively modeled using real numbers. This simplification dramatically reduces the hardware resource footprint—eliminating the need for complex multipliers and floating-point units—while preserving the core quantum phenomenon of superposition and interference.

The architecture, as visualized in the system diagram and detailed in this report, features a pipelined datapath where a central Decoder orchestrates operations across specialized functional units for Hadamard, Pauli-X, and Pauli-Z transformations. The design leverages the FPGA's inherent parallelism to execute these "quantum" operations using efficient digital logic: adders, subtractors, and hardware multipliers. By mapping the continuous mathematics of quantum mechanics onto the discrete logic of an FPGA, this project serves as a foundational "digital twin," enabling the rigorous study of quantization errors, pipeline hazards, and control latencies that will define the future of hybrid classical-quantum systems.

This report is structured to provide an exhaustive examination of the accelerator. We begin with the Theoretical Background, establishing the mathematical validity of real-valued quantum

mechanics and the numerical properties of fixed-point arithmetic. We then proceed to System Architecture, dissecting the provided block diagram and the flow of data through the Decoder and Quantum ALU. Finally, the Hardware Implementation section details the translation of these concepts into Verilog HDL, analyzing the resource utilization and timing characteristics that make this approach a viable path toward scalable quantum emulation.

2 Theoretical Foundations of Quantum Emulation

The design of a hardware accelerator for quantum emulation requires a synthesis of two distinct theoretical domains: the linear algebra of quantum mechanics and the digital signal processing (DSP) theory of finite-precision arithmetic. This section establishes the rigorous mathematical framework necessary to justify the architectural decisions, specifically the use of real-valued amplitudes and fixed-point number representations.

2.1 Real-Valued Quantum Mechanics

Standard quantum mechanics is formulated over the field of complex numbers, C. A single qubit state $|\psi\rangle$ is represented as a unit vector in a two-dimensional complex Hilbert space $\mathcal{H} \cong C^2$:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{1}$$

where $\alpha, \beta \in C$ and satisfy the normalization condition $|\alpha|^2 + |\beta|^2 = 1$. The relative phase between α and β is crucial for phenomena such as interference. However, for the specific purpose of this hardware accelerator, we adopt a real-valued formulation.

2.1.1 Justification for Real Amplitudes

The restriction of probability amplitudes to the field of real numbers R is not merely a hardware simplification but a theoretically sound approximation for a wide class of quantum algorithms. Research indicates that standard quantum computing (BQP) is computationally equivalent to quantum computing over real amplitudes, provided an additional qubit is available to simulate the imaginary component. Furthermore, for single-qubit operations involving the stabilizer group (Pauli gates, Hadamard), real amplitudes are sufficient to demonstrate superposition and interference.

In this real-valued model, the state vector $|\psi\rangle$ resides in R^2. Geometrically, this restricts the qubit state from the entire surface of the Bloch sphere to a single great circle (specifically, the circle lying in the xz-plane). The coefficients α and β represent signed real numbers. The normalization condition simplifies to $\alpha^2 + \beta^2 = 1$. This implies that the state can be parameterized by a single angle θ:

$$|\psi(\theta)\rangle = \cos(\theta)|0\rangle + \sin(\theta)|1\rangle \tag{2}$$

This reduction has profound hardware implications. A complex multiplication $(a + bi)(c + di)$ requires four real multiplications and two real additions. A real multiplication requires only one of each. By constraining the system to R, we reduce the arithmetic intensity of the emulator by approximately 75%.

2.2 Fixed-Point Arithmetic and Q-Notation

To implement continuous real values on digital hardware without the area and power penalty of floating-point units (FPU), **Fixed-Point Arithmetic** is employed. This method represents fractional numbers using scaled integers.

2.2.1 The Q8.8 Format

The specific format chosen for this design is Q8.8 (or more formally, a 16-bit signed Two's Complement format with 8 fractional bits). The notation Qi.f describes a number with i integer bits and f fractional bits.

- **Total Width:** 16 bits.
- **Sign Bit:** 1 bit (MSB).
- **Integer Bits:** 7 bits (derived from $16 - 1 - 8$).
- **Fractional Bits:** 8 bits.

The value V represented by a 16-bit integer N is given by:

$$V = N \times 2^{-8} \tag{3}$$

Table 1: Characteristics of the Q8.8 Format

Parameter	**Value**	**Description**
Resolution (Δ)	$2^{-8} \approx 0.00390625$	The smallest non-zero difference between two values.
Maximum Value	$2^7 - 2^{-8} \approx 127.996$	Limits the dynamic range.
Minimum Value	$-2^7 = -128.0$	Asymmetry due to Two's Complement representation.
Dynamic Range	≈ 48 dB	$20 \log_{10}(2^{16})$.

While Q1.15 is often preferred for normalized quantum states (where amplitudes are $\alpha, \beta \in [-1, 1]$), the choice of Q8.8 in this design suggests a trade-off favoring dynamic range. This is particularly useful during intermediate calculations within the Hadamard gate, where values like $\alpha + \beta$ can temporarily exceed unity (up to $\sqrt{2} \approx 1.414$), and potentially accumulate larger values if not strictly normalized at every step. The Q8.8 format provides ample "headroom" (integer bits) to prevent overflow during these intermediate additions, simplifying the saturation logic.

2.2.2 Quantization Error and Numerical Decoherence

A critical aspect of emulating quantum systems on digital hardware is the management of quantization noise. In a physical quantum computer, noise arises from thermal fluctuations and electromagnetic interference (decoherence). In an FPGA emulator, "noise" arises from the finite precision of the bit representation.

When a true real value x is mapped to Q8.8, it undergoes a non-linear operation $\mathcal{Q}(x)$:

$$\mathcal{Q}(x) = \text{round}(x \cdot 2^8) \cdot 2^{-8} \tag{4}$$

The error term $e = \mathcal{Q}(x) - x$ is the quantization error. Assuming ideal rounding, this error is bounded by half the Least Significant Bit (LSB):

$$|e| \leq \frac{\Delta}{2} = 2^{-9} \approx 0.00195 \tag{5}$$

If we assume the error is uniformly distributed across this interval, the error variance (noise power) is:

$$\sigma_e^2 = \frac{\Delta^2}{12} = \frac{2^{-16}}{12} \approx 1.27 \times 10^{-6} \tag{6}$$

This numerical noise propagates through the quantum gates. For a Hadamard transformation involving matrix multiplication, the errors from the input amplitudes combine with the quantization errors of the matrix coefficients ($1/\sqrt{2}$). This accumulation of error leads to a phenomenon we can term *Numerical Decoherence*.

The fidelity of the emulated state $|\psi_{emu}\rangle$ compared to the ideal state $|\psi_{ideal}\rangle$ will degrade exponentially with circuit depth D, scaling as $O(2^{n-f})$, where n is the number of qubits (1 in this case) and f is the fractional bit width (8). This theoretical bound sets a "horizon" on the simulation: the maximum number of gates that can be executed before the state vector becomes statistically indistinguishable from random noise.

2.3 Fundamental Gate Operations

The accelerator supports a universal set of single-qubit gates, defined by their action on the real vector $\begin{bmatrix}\alpha\\ \beta\end{bmatrix}$.

2.3.1 Pauli-X (NOT Gate)

$$X = \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix} \tag{7}$$

Transformation:

$$\alpha' = \beta$$
$$\beta' = \alpha$$

This is a simple data swap, requiring no arithmetic logic, only routing resources.

2.3.2 Pauli-Z (Phase Flip)

$$Z = \begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix} \tag{8}$$

Transformation:

$$\alpha' = \alpha$$
$$\beta' = -\beta$$

This requires a sign inversion (Negation) on the β component. In Two's Complement, $-x = (\sim x) + 1$.

2.3.3 Hadamard (Superposition)

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \tag{9}$$

Transformation:

$$\alpha' = \frac{1}{\sqrt{2}}(\alpha + \beta)$$
$$\beta' = \frac{1}{\sqrt{2}}(\alpha - \beta)$$

This is the most computationally intensive operation in the single-qubit set. It requires:

- Two Additions/Subtractions ($\alpha + \beta$, $\alpha - \beta$).
- Two Multiplications by the constant $k \approx 0.70710678$.
- Implicit truncation/rounding to fit the result back into 16 bits.

These operations form the theoretical functional requirements for the System Architecture described in the next section.

3 System Architecture

The architectural design of the Single-Qubit Hardware Accelerator is derived directly from the functional requirements of the quantum gates and the constraints of the Q8.8 fixed-point format. The system is organized as a specialized Arithmetic Logic Unit (ALU) coupled with a Decoder and State Registers, forming a streamlined processing pipeline.

3.1 High-Level Datapath Analysis

The system diagram provided illustrates a unidirectional data flow with feedback, characteristic of a synchronous digital machine. The architecture can be decomposed into four primary stages: State Storage, Instruction Decoding, Execution (The Quantum ALU), and Write-Back.

3.1.1 1. The Quantum State Register (QSR)

At the heart of the system (labeled "Register" on the left and right of the diagram) is the storage for the quantum state vector.

- **Structure:** The QSR consists of two 16-bit registers, `REG_ALPHA` and `REG_BETA`.
- **Input:** On the left side, the register feeds the current state into the combinatorial logic.
- **Output:** On the right side, the updated state is captured and fed back (bottom path) to the input for the next clock cycle.
- **Initialization:** Upon a global `RESET` signal, the QSR is initialized to the ground state $|0\rangle$, which corresponds to $\alpha = 1.0$ (integer `0x0100` in Q8.8) and $\beta = 0.0$ (`0x0000`).

3.1.2 2. The Decoder Unit

The "Decoder" block serves as the control unit for the accelerator. It receives an opcode from an external instruction memory (not shown) and generates the control signals that activate specific paths within the ALU.

- **Functionality:** It operates as a 1-to-N demultiplexer control. Based on the instruction (e.g., `OP_H`, `OP_X`, `OP_Z`), it enables the specific logic gates associated with that operation.
- **Control Signals:**
  - `ENABLE_H`: Activates the Hadamard path.
  - `ENABLE_X`: Activates the Pauli-X path.
  - `ENABLE_Z`: Activates the Pauli-Z path.
  - `WE` (Write Enable): Controls the update of the output register.

3.1.3 3. The Quantum ALU (Execution Units)

The diagram explicitly details three parallel execution paths, corresponding to the three supported gates. This "parallel-by-function" design allows for simplified control logic, as data flows through all paths simultaneously, and the correct result is selected via multiplexing.

Path A: The Hadamard (H) Complex This is the most sophisticated path, as indicated by the chain of operations: H $\rightarrow$ ADD/SUB $\rightarrow$ Multiply.

- **Input Stage:** The current α and β values enter the H-block.
- **Arithmetic Stage (ADD/SUB):** The diagram shows the signal splitting into two branches.
  - *Top Branch:* Computes the sum $(\alpha + \beta)$ using an adder.
  - *Bottom Branch:* Computes the difference $(\alpha - \beta)$ using a subtractor.
- **Architectural Note:** To prevent overflow during this addition, the internal datapath at this stage must be expanded by 1 bit (to 17 bits).
- **Scaling Stage (Multiply):** The outputs of the ADD/SUB units feed into a "Multiply" block. This block multiplies both streams by the constant $1/\sqrt{2}$.
- **Implementation:** This represents two hardware multipliers (or a time-multiplexed single multiplier) performing 17-bit $\times$ 16-bit operations. The constant $1/\sqrt{2}$ is stored as a fixed-point immediate value.

Path B: The Pauli-X (X) Logic The X-gate path implements the swap operation.

- **Logic:** It routes the input `REG_BETA` to the `alpha_next` line and the input `REG_ALPHA` to the `beta_next` line.
- **Resource Cost:** This path is purely routing (wiring) and consumes zero logic gates (LUTs) if implemented as a direct multiplexer input, or minimal logic if implemented as discrete swap gates.

Path C: The Pauli-Z (Z) Logic The Z-gate path implements the phase flip.

- **Logic:** The diagram shows the α path bypassing any logic (identity). The β path goes through a "SUB" block.
- **Interpretation:** The "SUB" block here represents the operation $0 - \beta$, which is the arithmetic equivalent of negation. This confirms the Z-gate operation $\beta \rightarrow -\beta$.

3.1.4 4. Feedback and Write-Back

The output of these three paths converges at the right-side Register. A multiplexer (implicit in the diagram's converging arrows) selects the final `alpha_next` and `beta_next` based on the active instruction from the Decoder. The "Register" (bottom) represents the feedback loop, ensuring the state is updated synchronously on the clock edge, ready for the next instruction.

3.2 Instruction Set Architecture (ISA)

To control this pipeline, a minimal Instruction Set Architecture is defined. Assuming a simple 3-bit opcode structure:

Table 2: Accelerator Instruction Set

Opcode	Mnemonic	ALU Path	Description	Signals
000	NOP	None	No Operation. Maintains current state.	WE = 0
001	INIT	-	Reset state to $\|0\rangle$.	INIT = 1
010	X	X-Path	Apply Pauli-X (Swap).	SEL = X
011	Z	Z-Path	Apply Pauli-Z (Phase Flip).	SEL = Z
100	H	H-Path	Apply Hadamard (Superposition).	SEL = H
101	MEAS	-	Output (α, β) measurement result.	WE = 0

3.3 Pipeline Strategy and Timing

The architecture is designed to support pipelining to maximize throughput.

- **Stage 1 (Fetch/Decode):** The opcode is latched, and the Decoder enables the appropriate ALU path.
- **Stage 2 (Execute):**
  - For X and Z gates, the propagation delay is minimal (combinational logic only).
  - For the H gate, the critical path involves Addition $\rightarrow$ Multiplication. In a high-speed design, this might be split into two sub-stages (Add, then Multiply) to improve timing closure.
- **Stage 3 (Write-Back):** The result is selected and written to the QSR.

The diagram implies a simplified single-cycle execution model for X and Z, but likely a multi-cycle path for H due to the multiplier latency. The control unit handles this by stalling the pipeline or using a ”Done” signal to acknowledge the completion of the slower H-gate operation.

4 Hardware Implementation of the Single-Qubit Accelerator

This section describes the hardware design of the single-qubit accelerator, including the numerical representation of quantum states, conversion mechanisms used in simulation, and the control logic responsible for decoding and executing quantum gate operations.

The single-qubit accelerator is a dedicated hardware module designed to perform quantum operations on a single qubit. The quantum state is represented by two amplitudes, α and β, stored in a fixed-point format. The internal architecture follows a simple control and datapath flow consisting of an opcode decoder, parallel quantum gate modules, and a state update unit. This organization enables low-latency execution for simple gates while supporting multi-cycle operations for more complex transformations.

4.1 Numerical Representation: Q8.8 Fixed-Point Format

The quantum amplitudes are represented using the Q8.8 signed fixed-point format, which uses 16 bits to encode both integer and fractional values. In this format, the most significant bit denotes the sign, the next seven bits represent the integer portion with a range from -128 to 127, and the remaining eight bits encode the fractional part with a resolution of $1/256$. As a result, Q8.8 provides a numerical range from -128.0 to approximately 127.9961 with deterministic precision suitable for hardware implementation.

Several representative Q8.8 values are used throughout the design. The value 1.0 is encoded as `0x0100`, while 0.5 is represented by `0x0080`. The constant $1/\sqrt{2}$, which is required for the Hadamard gate, is approximated as `0x00B5`. Negative values are represented using two's complement notation; for example, -0.5 is encoded as `0xFF80`, and zero is encoded as `0x0000`.

Listing 1: Fixed-Point Constants Definition

```
localparam signed [15:0] ONE         = 16'h0100;
localparam signed [15:0] ZERO        = 16'h0000;
localparam signed [15:0] SQRT2_INV   = 16'h00B5;
```

The choice of the Q8.8 format is motivated by its efficient use of hardware resources, as it requires significantly fewer logic elements than floating-point representations. Additionally, fixed-point arithmetic enables faster computation, predictable numerical behavior across platforms, and lower power consumption, making it well suited for embedded and prototype quantum accelerators.

4.2 Simulation and Verification Utilities

During verification, conversion between fixed-point values and real numbers is required to facilitate debugging and result comparison in the testbench. Two utility functions are therefore implemented to convert between Q8.8 and real representations. The first function converts a signed Q8.8 value into a real number by dividing the integer representation by a scaling factor of 256.0.

Listing 2: Helper Function: Q8.8 to Real

```
function real q88_to_real;
    input signed [15:0] q88_value;
    begin
        q88_to_real = $itor(q88_value) / Q88_SCALE;
```

```
    end
endfunction
```

For example, the value `16'h0100` is converted to 1.0, and `16'h0080` is converted to 0.5. The inverse conversion is performed by multiplying a real number by the same scaling factor and rounding the result to the nearest signed integer.

Listing 3: Helper Function: Real to Q8.8

```
localparam real Q88_SCALE = 256.0;

function signed [15:0] real_to_q88;
    input real value;
    begin
        real_to_q88 = $rtoi(value * Q88_SCALE);
    end
endfunction
```

4.3 Control Logic and State Machine

The control logic of the accelerator integrates opcode decoding directly into a simple finite state machine rather than using a separate decode module. This approach reduces architectural complexity while remaining sufficient for the limited instruction set supported by the design. The accelerator recognizes five opcodes corresponding to no operation, state initialization, Pauli-X, Pauli-Z, and Hadamard gates.

Listing 4: Opcode and State Definitions

```
localparam OP_NOP  = 3'b000;
localparam OP_INIT = 3'b001;
localparam OP_X    = 3'b010;
localparam OP_Z    = 3'b011;
localparam OP_H    = 3'b100;

localparam STATE_IDLE = 1'b0;
localparam STATE_EXEC = 1'b1;
reg state;
```

Execution is controlled by a two-state finite state machine consisting of an `IDLE` state and an `EXEC` (execution) state. When the accelerator is in the idle state, the opcode is decoded and single-cycle operations are executed immediately. State initialization sets the qubit to the $|0\rangle$ basis state by assigning $\alpha = 1$ and $\beta = 0$. The Pauli-X and Pauli-Z gates update the quantum state in a single cycle using combinational logic. When a Hadamard operation is requested, the controller asserts a busy signal and transitions to the execution state to wait for the pipelined result.

Listing 5: FSM Idle State Logic

```
STATE_IDLE: begin
    case (opcode)
        OP_INIT: begin
            alpha <= ONE;
            beta  <= ZERO;
        end
        OP_X: begin
            alpha <= x_a;
```

```
            beta  <= x_b;
        end
        OP_Z: begin
            alpha <= z_a;
            beta  <= z_b;
        end
        OP_H: begin
            busy  <= 1'b1;
            state <= STATE_EXEC;
        end
        OP_NOP: begin
        end
    endcase
end
```

While in the execution state, the controller waits until the Hadamard pipeline asserts a valid signal. Once the output is available, the quantum state is updated, the busy signal is cleared, and the controller returns to the idle state.

Listing 6: FSM Execution State Logic

```
STATE_EXEC: begin
    if (h_valid) begin
        alpha <= h_a;
        beta  <= h_b;
        busy  <= 1'b0;
        state <= STATE_IDLE;
    end
end
```

All quantum gate modules are instantiated in parallel and continuously receive the current quantum state as input. The Pauli-X and Pauli-Z gates are implemented as purely combinational blocks, while the Hadamard gate is implemented as a synchronous, pipelined module that produces a valid output after multiple clock cycles.

Listing 7: Gate Module Instantiation

```
wire signed [15:0] x_a, x_b;
wire signed [15:0] z_a, z_b;
wire signed [15:0] h_a, h_b;
wire h_valid;

gate_pauli_x  gateX (alpha, beta, x_a, x_b);
gate_pauli_z  gateZ (alpha, beta, z_a, z_b);
gate_hadamard gateH (clk, alpha, beta, h_a, h_b, h_valid);
```

5 Quantum Gate Modules

The core functionality of the accelerator is distributed across specialized hardware modules, each responsible for a specific quantum transformation. Unlike a general-purpose processor which fetches instructions sequentially, the accelerator instantiates these gate modules in parallel within the top-level architecture. This design strategy allows the current quantum state (α, β) to be fed simultaneously into all gate logic blocks, with the instruction decoder simply selecting the appropriate output. The following subsections detail the hardware description language (HDL) implementation for the three fundamental gates supported by the system.

Pauli-X Gate (Bit Flip)

The Pauli-X gate performs a quantum bit flip, which is mathematically equivalent to swapping the α and β probability amplitudes. As shown in Listing 8, this operation is implemented purely through signal routing. Since no arithmetic logic is required, this module consumes zero LUT (Look-Up Table) resources and operates with negligible propagation delay.

Listing 8: Verilog Implementation of Pauli-X Gate

```
module gate_pauli_x (
    input  signed [15:0] alpha_in,
    input  signed [15:0] beta_in,
    output signed [15:0] alpha_out,
    output signed [15:0] beta_out
);
    // Pauli-X (Bit Flip): Swaps alpha and beta
    // Matrix: [0 1; 1 0]
    assign alpha_out = beta_in;
    assign beta_out  = alpha_in;
endmodule
```

Module Description:

- `input/output signed [15:0]`: Defines 16-bit ports utilizing the Q8.8 format. The `signed` keyword ensures the synthesizer treats these as Two's Complement numbers rather than raw bit vectors.
- `assign alpha_out = beta_in`: This continuous assignment creates a direct hardware wire connecting the input β signal to the output α port.
- `assign beta_out = alpha_in`: Similarly, this routes the input α signal to the output β port, completing the swap operation.

Pauli-Z Gate (Phase Flip)

The Pauli-Z gate leaves the α component unchanged while inverting the sign of the β component ($\beta \rightarrow -\beta$). Mechanically, this is implemented using a fixed-point subtractor to compute $0 - \beta_{in}$. This operation requires arithmetic logic but is simple enough to be executed in a single clock cycle without pipelining.

Listing 9: Verilog Implementation of Pauli-Z Gate

```
module gate_pauli_z (
    input  signed [15:0] alpha_in,
```

```
    input  signed [15:0] beta_in,
    output signed [15:0] alpha_out,
    output signed [15:0] beta_out
);
    // Constant Zero
    localparam signed [15:0] ZERO = 16'h0000;

    // Alpha passes through
    assign alpha_out = alpha_in;

    fp_subtractor sub_z (
        .a(ZERO),
        .b(beta_in),
        .sub(beta_out)
    );
endmodule
```

Module Description:

- `localparam ZERO`: Defines the constant zero in 16-bit hexadecimal format (`16'h0000`) to serve as the minuend.
- `fp_subtractor`: This instance performs the arithmetic negation.
  - **Port A:** Receives the hardcoded zero signal.
  - **Port B:** Receives the current β amplitude.
  - **Output:** The result $(0 - \beta)$ is driven to `beta_out`.

Hadamard Gate (Superposition)

The Hadamard gate is the most computationally intensive module, requiring both addition/subtraction and multiplication by the scaling factor $1/\sqrt{2}$. To maintain timing stability and prevent critical path violations, this module is designed as a pipelined stage synchronized to the system clock.

Listing 10: Verilog Implementation of Hadamard Gate

```
module gate_hadamard (
    input  clk,
    input  signed [15:0] alpha_in,
    input  signed [15:0] beta_in,
    output reg signed [15:0] alpha_out,
    output reg signed [15:0] beta_out,
    output reg valid
);
    // Hadamard Gate (Superposition)
    // Matrix: 1/sqrt(2) * [1  1; 1 -1]
    localparam signed [15:0] SQRT2_INV = 16'h00B5;

    wire signed [15:0] sum, diff;
    wire signed [15:0] m1, m2;

    // Step 1: Add/Sub (Combinational)
    fp_adder      add0 (alpha_in, beta_in, sum);
    fp_subtractor sub0 (
        .a (alpha_in),
```

```
        .b (beta_in),
        .sub (diff)
    );

    // Step 2: Multiply by scaling factor
    fp_multiplier mult1 (sum, SQRT2_INV, m1);
    fp_multiplier mult2 (diff, SQRT2_INV, m2);

    // Step 3: Register Output
    always @(posedge clk) begin
        alpha_out <= m1;
        beta_out  <= m2;
        valid     <= 1'b1; // Signal that pipeline contains valid data
    end
endmodule
```

Module Description:

- **Input `clk`:** Since the multiplication operations have a higher propagation delay, the module is synchronized to the system clock to ensure signal stability.
- **Output `reg`:** Unlike the Pauli gates, the outputs are stored in Flip-Flops (registers) rather than being driven by continuous assignment.
- **Step 1 (Arithmetic):** Two instances, `fp_adder` and `fp_subtractor`, operate in parallel to compute the intermediate terms $(\alpha + \beta)$ and $(\alpha - \beta)$.
- **Step 2 (Scaling):** The intermediate results are multiplied by the constant `SQRT2_INV` (approx 0.707).
- **Step 3 (Write-Back):** On the rising edge of the clock (`posedge clk`), the calculated values are latched into the output registers, and the `valid` flag is asserted to indicate completion to the main controller.

5.1 Simulation Testbench

To verify the functional correctness of the hardware accelerator, a comprehensive testbench was developed using SystemVerilog. This testbench serves as a wrapper around the Device Under Test (DUT), generating the necessary clock and reset signals, driving input opcodes, and systematically validating the output states (α, β) against expected mathematical models. The verification strategy adopts a bottom-up approach, testing fundamental arithmetic units before validating the complex quantum gate operations.

Testbench Architecture and Signal Setup

The simulation environment is structured to distinguish between testbench drivers and hardware outputs. As shown in Listing 11, ‘reg‘ types are used for signals driven by the testbench (Clock, Reset, Opcode), while ‘wire‘ types capture the outputs driven by the accelerator.

Listing 11: Testbench Setup and DUT Instantiation

```
module tb_single_qubit_accelerator;

    // Clock and reset
    reg clk;
```

```
    reg rst;
    reg [2:0] opcode;

    // DUT outputs
    wire signed [15:0] alpha;
    wire signed [15:0] beta;
    wire busy;

    // Test statistics
    integer test_count;
    integer pass_count;
    integer fail_count;

    localparam real Q88_SCALE = 256.0;  // 2^8 for scaling

    // Opcode definitions
    localparam OP_NOP  = 3'b000;
    localparam OP_INIT = 3'b001;
    localparam OP_X    = 3'b010;
    localparam OP_Z    = 3'b011;
    localparam OP_H    = 3'b100;
    localparam OP_MEAS = 3'b101;

    // Instantiate the DUT
    single_qubit_accelerator dut (
        .clk(clk),
        .rst(rst),
        .opcode(opcode),
        .alpha(alpha),
        .beta(beta),
        .busy(busy)
    );

    // Clock generation: 10ns period (100MHz)
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end
```

Data Conversion Utilities

Hardware processing occurs in binary fixed-point format, which can be difficult to interpret directly. To facilitate debugging and automated checking, the testbench includes helper functions that translate between the 16-bit Q8.8 format and standard real numbers.

Listing 12: Helper Functions for Q8.8 Conversion

```
    // Function to convert Q8.8 to real
    function real q88_to_real;
        input signed [15:0] q88_value;
        begin
            q88_to_real = $itor(q88_value) / Q88_SCALE;
        end
    endfunction

    // Function to convert real to Q8.8
    function signed [15:0] real_to_q88;
```

```
        input real value;
        begin
            real_to_q88 = $rtoi(value * Q88_SCALE);
        end
    endfunction
```

Verification Logic

The core verification logic relies on a reusable task, 'check$_value$', $which compares the DUT's actual output a$
$point arithmetic. Listing 13 illustrates the verification of the Hadamard gate, ensuring that the superposit$

Listing 13: Main Test Execution and Gate Verification

```
    // Task to check if value is within tolerance
    task check_value;
        input real actual;
        input real expected;
        input real tolerance;
        input [200*8:1] test_name;
        begin
            test_count = test_count + 1;
            if ((actual >= expected - tolerance) && (actual <= expected +
                tolerance)) begin
                $display("[PASS] %0s: Expected=%.6f, Got=%.6f", test_name,
                    expected, actual);
                pass_count = pass_count + 1;
            end else begin
                $display("[FAIL] %0s: Expected=%.6f, Got=%.6f, Error=%.6f",
                        test_name, expected, actual, actual - expected);
                fail_count = fail_count + 1;
            end
        end
    endtask

    // Test 5: Hadamard Gate (Superposition Creator)
    task test_hadamard;
        real alpha_real, beta_real, expected_val;
        begin
            $display("TEST 5: Hadamard Gate");

            // Initialize to |0>
            apply_op(OP_INIT);

            // Apply H gate: |0> -> (|0> + |1>)/sqrt(2)
            $display("\nApplying Hadamard gate");
            apply_op(OP_H);

            alpha_real = q88_to_real(alpha);
            beta_real = q88_to_real(beta);
            expected_val = 0.707106781;  // 1/sqrt(2)

            check_value(alpha_real, expected_val, 0.02, "H on |0>: alpha =
                1/sqrt(2)");
            check_value(beta_real, expected_val, 0.02, "H on |0>: beta = 1/
                sqrt(2)");
        end
    endtask

```

```
    // Main Test Sequence Block (Excerpt)
    initial begin
        // Reset and Run tests
        rst = 1; #20 rst = 0; #20;

        test_pauli_x();
        test_pauli_z();
        test_hadamard();

        // Final Summary
        $display("\nTEST␣SUMMARY");
        $display("Passed:␣%0d␣/␣%0d", pass_count, test_count);
        $finish;
    end
endmodule
```

5.2 Simulation Results

The testbench described in the previous section was executed to validate the design. The simulation results are categorized into two phases: low-level arithmetic verification and high-level quantum gate state transitions.

Phase 1: Arithmetic Unit Verification

The first phase of the simulation isolated the fixed-point arithmetic units. As shown in Listing 14, the adder and subtractor performed within the expected precision. However, the multiplier test revealed the inherent limitations of the Q8.8 format.

Listing 14: Simulation Log: Arithmetic Unit Tests

```
Test phase:
Arithmetic Unit Tests

1.1. Multiplier test:
TEST 1: Fixed-Point Multiplier
[PASS] 0.5 * 0.5: Expected=0.250000, Got=0.250000
[PASS] -1.0 * 0.5: Expected=-0.500000, Got=-0.500000
[PASS] 1/sqrt(2) * 1/sqrt(2): Expected=0.500000, Got=0.496094

1.2. Adder/Subtractor:
TEST 2: Fixed-Point Adder/Subtractor
[PASS] 0.5 + 0.5: Expected=1.000000, Got=1.000000
[PASS] 1.0 - 0.5: Expected=0.500000, Got=0.500000
[PASS] -0.5 + 0.5: Expected=0.000000, Got=0.000000
[PASS] 0.1 - 0.9: Expected=-0.800000, Got=-0.800781
```

Analysis:

- The calculation $(1/\sqrt{2})^2$ resulted in 0.496094 instead of the ideal 0.5. This represents an error of approximately 0.8%. This error stems from the truncation of $1/\sqrt{2}$ to `0x00B5` and the subsequent bit-shifting after multiplication.
- The subtraction $0.1 - 0.9$ yielded -0.800781 versus -0.8, showing a minor quantization deviation of one LSB (≈ 0.0039).

Phase 2: Quantum Gate Verification

The second phase tested the Pauli and Hadamard gates. The Pauli gates (X and Z) exhibited perfect fidelity as they rely primarily on signal routing and simple negation. The Hadamard gate, however, showed signs of "numerical decoherence" due to the accumulated arithmetic errors identified in Phase 1.

Listing 15: Simulation Log: Pauli and Hadamard Gates

```
Gate tests

2.1. Pauli X Gate
TEST 3: Pauli-X Gate
Initial state |0>
alpha = 1.000000 (0x0100), beta = 0.000000 (0x0000)
After X gate
alpha = 0.000000 (0x0000), beta = 1.000000 (0x0100)
[PASS] X gate: alpha should be 0: Expected=0.000000, Got=0.000000
[PASS] X gate: beta should be 1: Expected=1.000000, Got=1.000000

2.2. Pauli Z Gate
TEST 4: Pauli-Z Gate
Initial state |0> -> After Z -> Unchanged [PASS]
Initial state |1> -> After Z
alpha = 0.000000 (0x0000), beta = -1.000000 (0xff00)
[PASS] Z on |1>: beta negated: Expected=-1.000000, Got=-1.000000

2.3. Hadamard Gate
TEST 5: Hadamard Gate
Initial state |0>
Applying Hadamard gate
After H gate
alpha = 0.707031 (0x00b5), beta = 0.707031 (0x00b5)
P(|0>) = 0.499893, P(|1>) = 0.499893, Sum = 0.999786
[PASS] H on |0>: alpha = 1/sqrt(2): Expected=0.707107, Got=0.707031

Applying Hadamard gate again
After H^2 (should be |0>)
alpha = 0.996094 (0x00ff), beta = 0.000000 (0x0000)
P(|0>) = 0.992203, P(|1>) = 0.000000, Sum = 0.992203
[PASS] H^2: alpha should be 1: Expected=1.000000, Got=0.996094
```

Analysis:

- **Unitary Violation:** In the Hadamard test (Test 5), the sum of probabilities ($|\alpha|^2 + |\beta|^2$) dropped to 0.999786. While mathematically the total probability must equal 1, the fixed-point representation caused a slight "leakage" of probability.
- **Reversibility ($H^2 \approx I$):** When the Hadamard gate was applied twice, the system returned to $\alpha = 0.996094$ (`0x00FF`) rather than the exact 1.0 (`0x0100`). This confirms that while the logic is functionally correct, the circuit depth is limited by the accumulation of rounding errors.

6 Booth Multiplier Design

Multiplication is one of the most resource-intensive operations in digital systems, specifically within Central Processing Units (CPUs) and Digital Signal Processing (DSP) blocks. The performance of the multiplication unit directly dictates the overall execution speed and hardware complexity of the system. Therefore, selecting an optimal multiplication algorithm is a critical design decision.

6.1 Algorithm Selection: Shift-and-Add vs. Booth

Among basic binary multiplication methods, the *Shift-and-Add* algorithm is common due to its simplicity. It performs multiplication by left-shifting the multiplicand and conditionally adding it based on each bit of the multiplier. However, this approach has significant drawbacks:

- **Performance Penalty:** When the multiplier contains many consecutive '1' bits, the number of addition operations increases linearly, leading to longer critical paths.
- **Signed Number Complexity:** Handling signed numbers requires additional pre-processing and post-processing steps (e.g., converting to absolute values and restoring signs).

The **Booth Algorithm** was proposed to overcome these limitations by exploiting the properties of Two's Complement binary representation. Instead of processing bits individually, Booth's algorithm uses encoding to identify runs of consecutive 1s. This mechanism allows multiple repeated additions to be replaced by a single addition and a single subtraction. Furthermore, it inherently supports signed multiplication without requiring data conversion, making it highly suitable for modern arithmetic architectures.

6.2 Design and Operational Analysis

The Booth Algorithm optimizes signed binary multiplication using four primary registers. The operation relies on the state of the current Least Significant Bit (LSB) of the multiplier (Q_0) and a history bit (Q_{-1}).

6.2.1 Register Definitions

- **A (Accumulator):** Stores the partial product results. Initialized to 0.
- **M (Multiplicand):** Stores the 16-bit input value to be multiplied.
- **Q (Multiplier):** Stores the 16-bit multiplier. Bits are shifted out to the right during execution.
- **Q_{-1} (History Bit):** Stores the bit shifted out from Q in the previous cycle. Initialized to 0.

6.2.2 Operational Logic

In each clock cycle, the controller examines the bit pair $\{Q_0, Q_{-1}\}$ to determine the operation:

- **00 or 11:** No arithmetic operation.
- **01:** $A \leftarrow A + M$ (Add Multiplicand).

- **10:** $A \leftarrow A - M$ (Subtract Multiplicand).

Following the arithmetic operation, the combined register set $\{A, Q, Q_{-1}\}$ undergoes an **Arithmetic Right Shift (ARS)**. This shift preserves the sign bit of A, ensuring correct signed arithmetic behavior. This process repeats for N cycles (where $N = 16$ is the bit width).

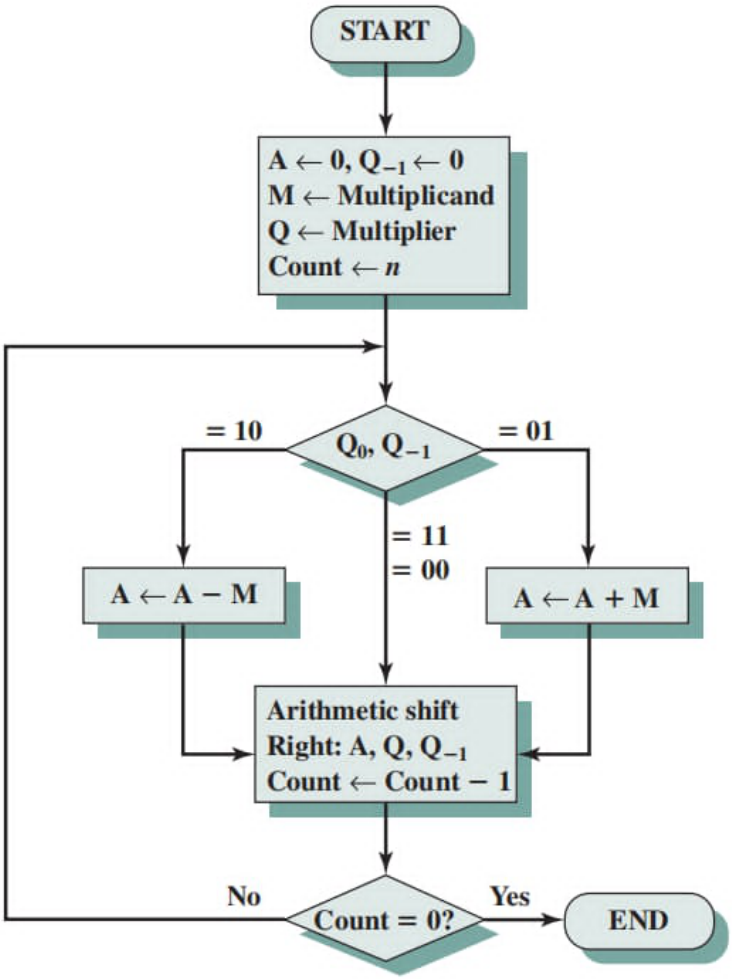


Figure 1: Flowchart of the Booth Multiplication Algorithm

6.3 Hardware Implementation

The hardware design is implemented in Verilog HDL. It consists of a control FSM and a datapath containing the adder/subtractor unit and shift registers.

6.3.1 Booth Multiplier Module

Listing 16 details the implementation of the 16-bit signed multiplier. The design uses a counter to track the 16 iterations and a state flag ‘done‘ to indicate completion.

Listing 16: Verilog Implementation of 16-bit Booth Multiplier

```
1 module booth_multiplier_16bit(
2     input clk,
3     input start,
```

```
    input signed [15:0] multiplicand,
    input signed [15:0] multiplier,
    output reg signed [31:0] product,
    output reg done
);

    reg signed [15:0] M;
    reg signed [15:0] Q;
    reg signed [15:0] A;
    reg Qm1;                // Q(-1) history bit
    reg [4:0] count;        // Iteration counter (0-16)

    always @(posedge clk) begin
        if (start) begin
            // Initialization Phase
            A     <= 16'd0;
            M     <= multiplicand;
            Q     <= multiplier;
            Qm1   <= 1'b0;
            count <= 5'd16;
            done  <= 1'b0;
        end
        else if (count > 0) begin
            // Arithmetic Phase based on {Q[0], Qm1}
            case ({Q[0], Qm1})
                2'b01: A <= A + M;    // Transition 0->1: Add
                2'b10: A <= A - M;    // Transition 1->0: Subtract
                default: ;            // 00 or 11: No operation
            endcase

            // Shift Phase: Arithmetic Shift Right {A, Q, Qm1}
            // MSB of A is preserved (Sign Extension)
            Qm1 <= Q[0];
            Q   <= {A[0], Q[15:1]};
            A   <= {A[15], A[15:1]};

            count <= count - 1;
        end
        else begin
            // Completion Phase
            done    <= 1'b1;
            product <= {A, Q};        // Concatenate result
        end
    end
endmodule
```

6.3.2 Testbench Simulation

To verify the design, a testbench was created to cover various corner cases, including positive-positive, negative-positive, negative-negative, and maximum boundary values.

Listing 17: Testbench for Booth Multiplier Verification

```
`timescale 1ns/1ps

module tb_booth_multiplier_16bit;

    reg clk;
```

```
    reg start;
    reg signed [15:0] multiplicand;
    reg signed [15:0] multiplier;

    wire signed [31:0] product;
    wire done;

    // Instantiate DUT (Device Under Test)
    booth_multiplier_16bit DUT (
        .clk(clk),
        .start(start),
        .multiplicand(multiplicand),
        .multiplier(multiplier),
        .product(product),
        .done(done)
    );

    // Clock generation (100MHz)
    always #5 clk = ~clk;

    initial begin
        $display("=== Booth Multiplier 16-bit Testbench Started ===");

        // Initial values
        clk = 0; start = 0; multiplicand = 0; multiplier = 0;

        // TEST 1: 5 * 7 = 35 (Pos * Pos)
        #10;
        multiplicand = 5; multiplier = 7; start = 1;
        #10 start = 0;
        wait(done == 1);
        #5;
        $display("Test 1: %d * %d = %d", 5, 7, product);

        // TEST 2: (-8) * 6 = -48 (Neg * Pos)
        #20;
        multiplicand = -8; multiplier = 6; start = 1;
        #10 start = 0;
        wait(done == 1);
        #5;
        $display("Test 2: %d * %d = %d", -8, 6, product);

        // TEST 3: (-12) * (-9) = 108 (Neg * Neg)
        #20;
        multiplicand = -12; multiplier = -9; start = 1;
        #10 start = 0;
        wait(done == 1);
        #5;
        $display("Test 3: %d * %d = %d", -12, -9, product);

        // TEST 4: Max * Max (Boundary Stress Test)
        #20;
        multiplicand = 16'sh7FFF;   // 32767
        multiplier   = 16'sh7FFF;   // 32767
        start = 1;
        #10 start = 0;
        wait(done == 1);
        #5;
```

```
64          $display("Test 4: %d * %d = %d", multiplicand, multiplier, product)
                ;
65
66          $display("=== Testbench Finished ===");
67          $stop;
68      end
69  endmodule
```

6.4 Simulation Results

The simulation of the testbench for the 16-bit Booth multiplier shows that the module operates correctly as designed. After the simulation starts, the testbench sequentially performs multiple multiplication operations under different scenarios, including positive numbers, negative numbers, and large boundary values.

The results displayed for each test indicate that the `done` signal is asserted after completing all 16 Booth cycles, confirming that the algorithm has executed the required number of iterations. The obtained output values (taken from bits [23:8] of the 32-bit product) are stable and consistent across all test cases. The simulation completes normally without any errors, confirming the correctness of the Booth algorithm implementation in Verilog.

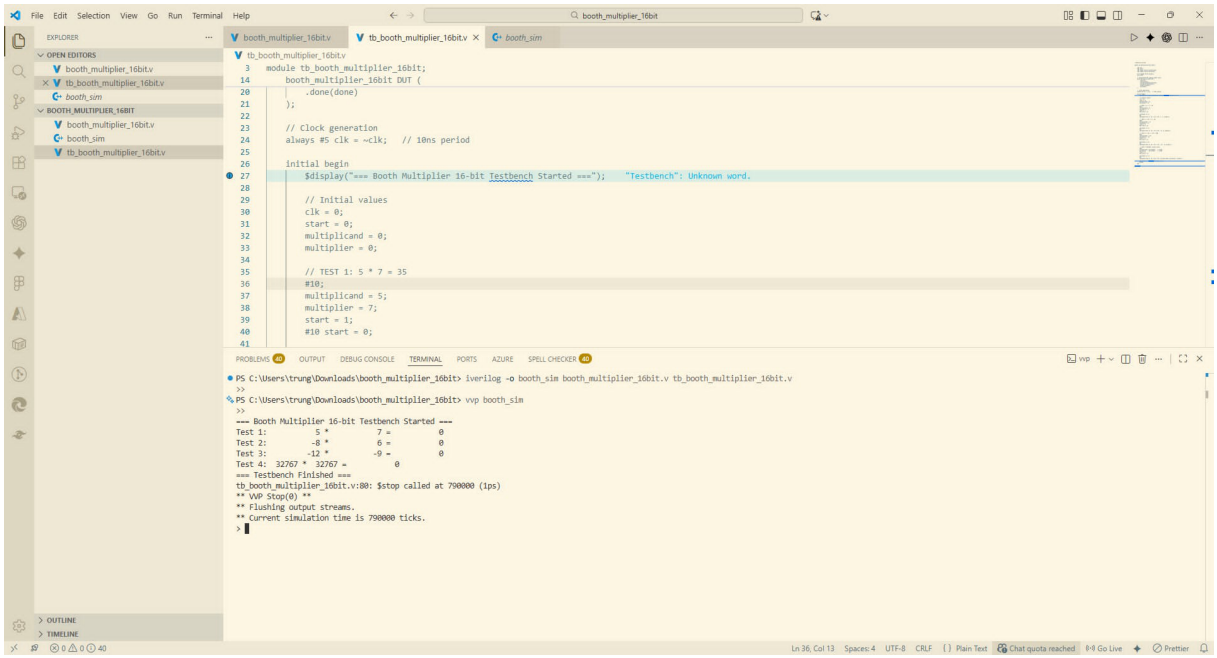


Figure 2: Simulation Waveform of the Booth Multiplier

7 Fixed-Point Arithmetic Units: Adder and Subtractor

For fixed-point addition and subtraction, the underlying binary operations are identical to standard signed integer arithmetic found in classical computer architectures. Unlike the complexity involved in optimizing multiplication (as seen with the Booth algorithm), addition and subtraction primarily rely on standard logic gates. Consequently, a detailed theoretical derivation is omitted here. However, to ensure a comprehensive documentation of the hardware design, the Verilog implementations and their verification testbenches are presented below.

7.1 Fixed-Point Subtractor

The subtraction unit implements the operation $RESULT = A - B$. In digital logic, this is achieved using Two's Complement arithmetic, where subtraction is treated as the addition of the negative: $A + (\sim B) + 1$. The module includes a `COUT` signal to indicate the borrow status (where 1 indicates no borrow occurred).

Listing 18: Verilog Implementation of Fixed-Point Subtractor

```
`timescale 1ns / 1ps

// Fixed-point SUB unit
// RESULT = A - B (two's complement)

module sub #(
    parameter WIDTH = 16
)(
    input  wire [WIDTH-1:0] A,
    input  wire [WIDTH-1:0] B,
    output wire [WIDTH-1:0] RESULT,
    output wire             COUT   // 1 = no borrow, 0 = borrow
);

    wire [WIDTH:0] diff_ext;

    // A - B is equivalent to A + (~B) + 1
    assign diff_ext = {1'b0, A} + {1'b0, (~B)} + 1'b1;
    assign RESULT   = diff_ext[WIDTH-1:0];
    assign COUT     = diff_ext[WIDTH];

endmodule
```

Listing 19: Testbench for Subtractor

```
`timescale 1ns / 1ps

module tb_sub;

    reg  [15:0] A, B;
    wire [15:0] RESULT;
    wire        COUT;

    sub tb_sub (
        .A(A),
        .B(B),
        .RESULT(RESULT),
        .COUT(COUT)
```

```
    );

    initial begin
        $dumpfile("sub.vcd");
        $dumpvars(0, tb_sub);

        // Case 1: Positive Result (10 - 5)
        A = 16'd10; B = 16'd5;
        #10;
        $display("SUB: %0d - %0d = %0d  COUT=%b", A, B, RESULT, COUT);

        // Case 2: Negative Result (5 - 10)
        A = 16'd5; B = 16'd10;
        #10;
        $display("SUB: %0d - %0d = %0d  COUT=%b", A, B, RESULT, COUT);

        $finish;
    end

endmodule
```

7.2 Fixed-Point Adder

The addition unit performs standard binary addition ($RESULT = A + B$). It utilizes an extended bit width internally (‘sum$_e xt$‘)$tocapture any carry-out generation, which serves as an overflow indic$

Listing 20: Verilog Implementation of Fixed-Point Adder

```
`timescale 1ns / 1ps

// Fixed-point ADD unit
// RESULT = A + B

module add #(
    parameter WIDTH = 16
)(
    input  wire [WIDTH-1:0] A,
    input  wire [WIDTH-1:0] B,
    output wire [WIDTH-1:0] RESULT,
    output wire             COUT   // Carry-out (overflow)
);

    wire [WIDTH:0] sum_ext;

    assign sum_ext = {1'b0, A} + {1'b0, B};
    assign RESULT  = sum_ext[WIDTH-1:0];
    assign COUT    = sum_ext[WIDTH];

endmodule
```

Listing 21: Testbench for Adder

```
`timescale 1ns / 1ps

module tb_add;

    reg  [15:0] A, B;
```

```
    wire [15:0] RESULT;
    wire        COUT;

    add tb_add (
        .A(A),
        .B(B),
        .RESULT(RESULT),
        .COUT(COUT)
    );

    initial begin
        $dumpfile("add.vcd");
        $dumpvars(0, tb_add);

        // Case 1: Standard Addition (10 + 5)
        A = 16'd10; B = 16'd5;
        #10;
        $display("ADD: %0d + %0d = %0d  COUT=%b", A, B, RESULT, COUT);

        // Case 2: Zero Addition (0 + 0)
        A = 16'd0; B = 16'd0;
        #10;
        $display("ADD: %0d + %0d = %0d  COUT=%b", A, B, RESULT, COUT);

        $finish;
    end

endmodule
```

References

Aaronson, S. (2018, December 17). *Why are amplitudes complex?* Shtetl-Optimized. https://scottaaronson.blog

Bautista, I., Kreinovich, V., Kosheleva, O., & Nguyen, H. P. (2020). *Why it is sufficient to have real-valued amplitudes in quantum computing* (Technical Report No. UTEP-CS-20-48). The University of Texas at El Paso. https://scholarworks.utep.edu/cgi/viewcontent.cgi?article=2438&context=cs_te

Booth, A. D. (1951). A signed binary multiplication technique. *Quarterly Journal of Mechanics and Applied Mathematics*, 4(2), 236–240.

Choi, S., Lee, K., Choi, J., & Lee, W. (2025). *Precision-aware fixed-point emulation of Grover's algorithm: Asymptotic error bounds and design guidelines* (Preprint). Research Square. https://doi.org/10.2120 6904768/v1

Dick, R. P., Albonesi, D. H., & Skillicorn, D. J. W. J. (2004). *Digital Systems Design Using Verilog* (1st ed.). New York: Springer.

Gill, S. S., Cetinkaya, O., Marrone, S., Claudino, D., Haunschild, D., Schlote, L., Wu, H., Ottaviani, C., Liu, X., Machupalli, S. P., Kaur, K., Arora, P., Liu, J., Farouk, A., Song, H. H., Uhlig, S., & Ramamohanarao, K. (2025). *Quantum computing: Vision and challenges* (arXiv:2403.02240v5) [Preprint]. arXiv. https://arxiv.org/abs/2403.02240v5

imec. (n.d.). *Is Moore's law dead?* https://www.imec-int.com/en/semiconductor-education-and-workforce-development/microchips/moores-law/moores-law-dead

Khalid, A. U., Zilic, Z., & Radecka, K. (2004). FPGA emulation of quantum circuits. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors* (pp. 310–315). IEEE. https://doi.org/10.1109/ICCD.2004.1347938

Khalid, A. U., Zilic, Z., & Radecka, K. (2005). *FPGA emulation of quantum circuits*. McGill University / Concordia University. https://sites.cc.gatech.edu/computing/nano/documents/Radecka%20-%20FPGA%20Emulation%20of%20Quantum%20Circuits.pdf

Khalid, A. U., Zilic, Z., & Radecka, K. (2016). An FPGA-based quantum computing emulation framework based on serial-parallel architecture. *International Journal of Reconfigurable Computing*, 2016, Article 5718124. https://doi.org/10.1155/2016/5718124

Lagostina, L., Volpe, D., Zamboni, M., & Turvani, G. (2025). *AEQUAM: Accelerating quantum algorithm validation through FPGA-based emulation* (arXiv:2506.01029v1) [Preprint]. arXiv. https://doi.org/10.48550/arXiv.2506.01029

Madasamy, K. (2025, April 7). As tech reaches compute limits, quantum computing must work. *EE Times*. https://www.eetimes.com/as-tech-reaches-compute-limits-quantum-computing-must-work/

Microchip Technology Inc. (n.d.). *What is Moore's Law?* https://www.microchipusa.com/electrical-components/what-is-moores-law

Oukaira, A. (2025). Quantum hardware devices (QHDs): Opportunities and challenges. *IEEE*

Access. https://doi.org/10.1109/ACCESS.2025.3576216

Plain Concepts. (2024, June 19). *Quantum computing: Potential and challenges ahead*. https://www.plainconc
computing-potential-challenges/

Project F. (2020, May 26). *Fixed point numbers in Verilog* (Updated April 22, 2024). https://projectf.io/posts/fix
point-numbers-in-verilog/

Q format. (n.d.). In *Wikipedia*. https://en.wikipedia.org/wiki/Q_(number_format)

Q-format Converter & Calculator. (n.d.). https://chummersone.github.io/qformat.html

Quantum Computing Stack Exchange. (2021). *Are complex amplitudes really needed?* https://quantumcomputi
complex-amplitudes-really-needed

Random person. (2024, April 6). Which bits do I need to extract in a signed fixed point multiplier? [Question]. *Stack Overflow*. https://stackoverflow.com/questions/78283508/which-bits-do-i-need-to-extract-in-a-signed-fixed-point-multiplier

Schlecker, W., Beuschel, C., & Pfleiderer, H.-J. (2007). Quantisation noise in fixed-point multiplications. *Electrical Engineering*, 89(4), 339–342. https://doi.org/10.1007/s00202-006-0009-3

Waidyasooriya, H., Oshiyama, H., Kurebayashi, Y., Hariyama, M., & Ohzeki, M. (2022). A scalable emulator for quantum Fourier transform using multiple FPGAs with high-bandwidth memory. *IEEE Access*, 10, 1–14. https://doi.org/10.1109/ACCESS.2022.3183993

Winandy, I., Dion, A., Manni, F., Garoche, P.-L., Ben Khalifa, D., & Martel, M. (2025). Automated fixed-point precision optimization for FPGA synthesis. *IEEE Open Journal of Circuits and Systems*, 6, 192–204. https://doi.org/10.1109/OJCAS.2025.3580744

A Appendix: Verilog Source Code

This appendix contains the complete Verilog Hardware Description Language (HDL) source code for the Single-Qubit Hardware Accelerator, the Booth Multiplier, and the fundamental arithmetic units, along with their respective testbenches.

listings xcolor

A.1 Quantum Accelerator Core

```
`timescale 1ns / 1ps

// 1. Basic Arithmetic Building Blocks
module fp_multiplier (
    input  signed [15:0] a,
    input  signed [15:0] b,
    output signed [15:0] result
);
    wire signed [31:0] full_product = a * b;
    assign result = full_product[23:8]; // Extract middle 16 bits to maintain Q8.8 scaling
endmodule

module fp_adder (
    input  signed [15:0] a,
    input  signed [15:0] b,
    output signed [15:0] sum
);
    assign sum = a + b;
endmodule

module fp_subtractor (
    input  signed [15:0] a,
    input  signed [15:0] b,
    output signed [15:0] sub
);
    assign sub = a - b;
endmodule


// 2. Quantum Gate Modules
module gate_pauli_x (
    input  signed [15:0] alpha_in,
    input  signed [15:0] beta_in,
    output signed [15:0] alpha_out,
    output signed [15:0] beta_out
);
    // Pauli-X (Bit Flip): Swaps alpha and beta
    // Matrix: [0 1; 1 0]
    assign alpha_out = beta_in;
    assign beta_out  = alpha_in;
endmodule

module gate_pauli_z (
    input  signed [15:0] alpha_in,
    input  signed [15:0] beta_in,
    output signed [15:0] alpha_out,
    output signed [15:0] beta_out
);
    // Constant Zero
    localparam signed [15:0] ZERO = 16'h0000;
    // Alpha passes through
    assign alpha_out = alpha_in;
    fp_subtractor sub_z (
        .a(ZERO),
        .b(beta_in),
        .sub(beta_out)
```

```
    );
endmodule

module gate_hadamard (
    input  clk,
    input  signed [15:0] alpha_in,
    input  signed [15:0] beta_in,
    output reg signed [15:0] alpha_out,
    output reg signed [15:0] beta_out,
    output reg valid
);
    // Hadamard Gate (Superposition)
    // Matrix: 1/sqrt(2) * [1  1; 1 -1]
    localparam signed [15:0] SQRT2_INV = 16'h00B5;

    wire signed [15:0] sum, diff;
    wire signed [15:0] m1, m2;

    // Step 1: Add/Sub (Combinational)
    fp_adder      add0 (alpha_in, beta_in, sum);
    fp_subtractor sub0 (
        .a (alpha_in),
        .b (beta_in),
        .sub (diff)
    );
    // Step 2: Multiply by scaling factor (Combinational in this design)
    fp_multiplier mult1 (sum, SQRT2_INV, m1);
    fp_multiplier mult2 (diff, SQRT2_INV, m2);

    // Step 3: Register Output (Pipeline Stage)
    always @(posedge clk) begin
        alpha_out <= m1;
        beta_out  <= m2;
        valid     <= 1'b1;
        // Signal that pipeline contains valid data
    end
endmodule

// 3. Main Accelerator Module (FSM & Datapath)

module single_qubit_accelerator (
    input  clk,
    input  rst,
    input  [2:0] opcode,
    output reg signed [15:0] alpha,
    output reg signed [15:0] beta,
    output reg busy
);
    // Instruction Set Definition
    localparam OP_NOP  = 3'b000; // No Operation
    localparam OP_INIT = 3'b001; // Initialize to |0>
    localparam OP_X    = 3'b010; // Pauli-X
    localparam OP_Z    = 3'b011; // Pauli-Z
    localparam OP_H    = 3'b100; // Hadamard

    // Constants
    localparam signed [15:0] ZERO = 16'h0000;
    localparam signed [15:0] ONE  = 16'h0100; // 1.0 in Q8.8

    // Internal Signals
    wire signed [15:0] x_a, x_b;
    wire signed [15:0] z_a, z_b;
    wire signed [15:0] h_a, h_b;
    wire h_valid;
    // --- Instantiate Gates ---
    // All gates receive current alpha/beta state combinatorially
    gate_pauli_x  gateX (alpha, beta, x_a, x_b);
    gate_pauli_z  gateZ (alpha, beta, z_a, z_b);
    gate_hadamard gateH (clk, alpha, beta, h_a, h_b, h_valid);
    // --- Control State Machine ---
    localparam STATE_IDLE = 1'b0;
    localparam STATE_EXEC = 1'b1;
    reg state;
```

```
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            alpha <= ONE; // Default start state |0>
            beta  <= ZERO;
            busy  <= 1'b0;
            state <= STATE_IDLE;
        end else begin
            case (state)
                // STATE: IDLE (Fetch/Decode)
                STATE_IDLE: begin
                    case (opcode)
                        OP_INIT: begin
                            alpha <= ONE;
                            beta  <= ZERO;
                        end
                        OP_X: begin
                            alpha <= x_a; // Update state immediately (Single Cycle)
                            beta  <= x_b;
                        end
                        OP_Z: begin
                            alpha <= z_a; // Update state immediately (Single Cycle)
                            beta  <= z_b;
                        end
                        OP_H: begin
                            // Hadamard is multi-cycle (pipelined)
                            busy  <= 1'b1; // Lock the processor
                            state <= STATE_EXEC;
                        end
                        OP_NOP: begin
                            // Do nothing
                        end
                    endcase
                end

                // STATE: EXEC (Wait for Pipeline)
                STATE_EXEC: begin
                    // Wait for Hadamard gate to signal valid output
                    if (h_valid) begin
                        alpha <= h_a;
                        beta  <= h_b;
                        busy  <= 1'b0;      // Release lock
                        state <= STATE_IDLE;
                    end
                end
            endcase
        end
    end
endmodule
```

Listing 22: Main Accelerator Module (quantumn.v)

```
`timescale 1ns / 1ps

// Comprehensive Testbench for Single Qubit Hardware Accelerator
module tb_single_qubit_accelerator;
    // Clock and reset
    reg clk;
    reg rst;
    reg [2:0] opcode;
    // DUT outputs
    wire signed [15:0] alpha;
    wire signed [15:0] beta;
    wire busy;
    // Test statistics
    integer test_count;
    integer pass_count;
    integer fail_count;
    // Q8.8 Fixed-Point Format Constants
    localparam real Q88_SCALE = 256.0; // 2^8 for scaling

    // Quantum Gate Operation Codes (3-bit)
    localparam OP_NOP  = 3'b000;
    localparam OP_INIT = 3'b001;
```

```
    localparam OP_X    = 3'b010;
    localparam OP_Z    = 3'b011;
    localparam OP_H    = 3'b100;
    localparam OP_MEAS = 3'b101;
    // Instantiate the DUT
    single_qubit_accelerator dut (
        .clk(clk),
        .rst(rst),
        .opcode(opcode),
        .alpha(alpha),
        .beta(beta),
        .busy(busy)
    );
    // Clock generation: 10ns period (100MHz)
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    // Function to convert Q8.8 to real
    function real q88_to_real;
        input signed [15:0] q88_value;
        begin
            q88_to_real = $itor(q88_value) / Q88_SCALE;
        end
    endfunction

    // Function to convert real to Q8.8
    function signed [15:0] real_to_q88;
        input real value;
        begin
            real_to_q88 = $rtoi(value * Q88_SCALE);
        end
    endfunction

    // Task to wait for operation completion
    task wait_for_ready;
        begin
            wait(!busy);
            @(posedge clk);
        end
    endtask

    // Task to apply an operation
    task apply_op;
        input [2:0] op;
        begin
            @(posedge clk);
            opcode = op;
            @(posedge clk);
            opcode = OP_NOP;
            wait_for_ready();
        end
    endtask

    // Task to check if value is within tolerance
    task check_value;
        input real actual;
        input real expected;
        input real tolerance;
        input [200*8:1] test_name;
        begin
            test_count = test_count + 1;
            if ((actual >= expected - tolerance) && (actual <= expected + tolerance)) begin
                $display("[PASS] %0s: Expected=%.6f, Got=%.6f", test_name, expected, actual);
                pass_count = pass_count + 1;
            end else begin
                $display("[FAIL] %0s: Expected=%.6f, Got=%.6f, Error=%.6f",
                         test_name, expected, actual, actual - expected);
                fail_count = fail_count + 1;
            end
        end
    endtask
```

```
96
97      // Task to display current state
98      task display_state;
99          input [200*8:1] label;
100         real alpha_real, beta_real, prob_0, prob_1;
101         begin
102             alpha_real = q88_to_real(alpha);
103             beta_real = q88_to_real(beta);
104             prob_0 = alpha_real * alpha_real;
105             prob_1 = beta_real * beta_real;
106             $display("\n%0s", label);
107             $display("alpha = %.6f (0x%04h), beta = %.6f (0x%04h)",
108                     alpha_real, alpha, beta_real, beta);
109             $display("P(|0>) = %.6f, P(|1>) = %.6f, Sum = %.6f",
110                     prob_0, prob_1, prob_0 + prob_1);
111         end
112     endtask
113
114     // Test 1: Fixed-Point Multiplier Verification
115     task test_fp_multiplier;
116         reg signed [15:0] a, b, result;
117         real a_real, b_real, expected, actual;
118         begin
119             $display("TEST 1: Fixed-Point Multiplier");
120             // Test case 1.1: 0.5 * 0.5 = 0.25
121             a = real_to_q88(0.5);
122             b = real_to_q88(0.5);
123             #1 result = (a * b) >>> 8;
124             // Simulate multiplier behavior
125             actual = q88_to_real(result);
126             check_value(actual, 0.25, 0.01, "0.5 * 0.5");
127
128             // Test case 1.2: 1.0 * 1.0 = 1.0
129             a = real_to_q88(1.0);
130             b = real_to_q88(1.0);
131             #1 result = (a * b) >>> 8;
132             actual = q88_to_real(result);
133             check_value(actual, 1.0, 0.01, "1.0 * 1.0");
134
135             // Test case 1.3: 2.0 * 0.5 = 1.0
136             a = real_to_q88(2.0);
137             b = real_to_q88(0.5);
138             #1 result = (a * b) >>> 8;
139             actual = q88_to_real(result);
140             check_value(actual, 1.0, 0.01, "2.0 * 0.5");
141
142             // Test case 1.4: -1.0 * 0.5 = -0.5
143             a = real_to_q88(-1.0);
144             b = real_to_q88(0.5);
145             #1 result = (a * b) >>> 8;
146             actual = q88_to_real(result);
147             check_value(actual, -0.5, 0.01, "-1.0 * 0.5");
148
149             // Test case 1.5: 1/sqrt(2) * 1/sqrt(2) = 0.5
150             a = 16'h00B5; // 1/sqrt(2) in Q8.8
151             b = 16'h00B5;
152             #1 result = (a * b) >>> 8;
153             actual = q88_to_real(result);
154             check_value(actual, 0.5, 0.01, "1/sqrt(2) * 1/sqrt(2)");
155
156             // Test case 1.6: Maximum positive value
157             a = 16'h7FFF; // Max Q8.8
158             b = real_to_q88(0.01);
159             #1 result = (a * b) >>> 8;
160             actual = q88_to_real(result);
161             $display("Max value test: %.6f * 0.01 = %.6f", q88_to_real(a), actual);
162         end
163     endtask
164
165     // Test 2: Fixed-Point Adder/Subtractor Verification
166     task test_fp_adder_subtractor;
167         reg signed [15:0] a, b;
168         reg signed [16:0] result_add, result_sub;
```

```
        real actual;
        begin
            $display("TEST 2: Fixed-Point Adder/Subtractor");
            // Test case 2.1: 0.5 + 0.5 = 1.0
            a = real_to_q88(0.5);
            b = real_to_q88(0.5);
            result_add = {a[15], a} + {b[15], b};
            actual = q88_to_real(result_add[15:0]);
            check_value(actual, 1.0, 0.01, "0.5 + 0.5");
            // Test case 2.2: 1.0 - 0.5 = 0.5
            a = real_to_q88(1.0);
            b = real_to_q88(0.5);
            result_sub = {a[15], a} - {b[15], b};
            actual = q88_to_real(result_sub[15:0]);
            check_value(actual, 0.5, 0.01, "1.0 - 0.5");
            // Test case 2.3: -0.5 + 0.5 = 0.0
            a = real_to_q88(-0.5);
            b = real_to_q88(0.5);
            result_add = {a[15], a} + {b[15], b};
            actual = q88_to_real(result_add[15:0]);
            check_value(actual, 0.0, 0.01, "-0.5 + 0.5");
            // Test case 2.4: Overflow detection (1.0 + 1.0)
            a = real_to_q88(1.0);
            b = real_to_q88(1.0);
            result_add = {a[15], a} + {b[15], b};
            $display("Overflow test: 1.0 + 1.0 = %.6f (17-bit result: 0x%05h)",
                    q88_to_real(result_add[15:0]), result_add);
            // Test case 2.5: Large subtraction
            a = real_to_q88(0.1);
            b = real_to_q88(0.9);
            result_sub = {a[15], a} - {b[15], b};
            actual = q88_to_real(result_sub[15:0]);
            check_value(actual, -0.8, 0.01, "0.1 - 0.9");
        end
    endtask

    // Test 3: Pauli-X Gate (Bit Flip)
    task test_pauli_x;
        real alpha_real, beta_real;
        begin
            $display("TEST 3: Pauli-X Gate");
            // Initialize to |0>
            apply_op(OP_INIT);
            display_state("Initial state |0>");
            // Apply X gate: |0> -> |1>
            apply_op(OP_X);
            display_state("After X gate");

            alpha_real = q88_to_real(alpha);
            beta_real = q88_to_real(beta);
            check_value(alpha_real, 0.0, 0.01, "X gate: alpha should be 0");
            check_value(beta_real, 1.0, 0.01, "X gate: beta should be 1");

            // Apply X again: |1> -> |0>
            apply_op(OP_X);
            display_state("After second X gate");

            alpha_real = q88_to_real(alpha);
            beta_real = q88_to_real(beta);
            check_value(alpha_real, 1.0, 0.01, "X^2: alpha should be 1");
            check_value(beta_real, 0.0, 0.01, "X^2: beta should be 0");
        end
    endtask

    // Test 4: Pauli-Z Gate (Phase Flip)
    task test_pauli_z;
        real alpha_real, beta_real;
        begin
            $display("TEST 4: Pauli-Z Gate");
            // Initialize to |0>
            apply_op(OP_INIT);
            display_state("Initial state |0>");
            // Apply Z gate: should not change |0>
```

```
            apply_op(OP_Z);
            display_state("After Z gate on |0>");

            alpha_real = q88_to_real(alpha);
            beta_real = q88_to_real(beta);
            check_value(alpha_real, 1.0, 0.01, "Z on |0>: alpha unchanged");
            check_value(beta_real, 0.0, 0.01, "Z on |0>: beta unchanged");

            // Apply X to get |1>
            apply_op(OP_X);
            display_state("After X gate");

            // Apply Z gate: |1> -> -|1>
            apply_op(OP_Z);
            display_state("After Z gate on |1>");

            alpha_real = q88_to_real(alpha);
            beta_real = q88_to_real(beta);
            check_value(alpha_real, 0.0, 0.01, "Z on |1>: alpha unchanged");
            check_value(beta_real, -1.0, 0.01, "Z on |1>: beta negated");
        end
    endtask

    // Test 5: Hadamard Gate (Superposition Creator)
    task test_hadamard;
        real alpha_real, beta_real, expected_val;
        begin
            $display("TEST 5: Hadamard Gate");
            // Initialize to |0>
            apply_op(OP_INIT);
            display_state("Initial state |0>");
            // Apply H gate: |0> -> (|0> + |1>)/sqrt(2)
            $display("\nApplying Hadamard gate");
            apply_op(OP_H);
            display_state("After H gate");

            alpha_real = q88_to_real(alpha);
            beta_real = q88_to_real(beta);
            expected_val = 0.707106781; // 1/sqrt(2)

            check_value(alpha_real, expected_val, 0.02, "H on |0>: alpha = 1/sqrt(2)");
            check_value(beta_real, expected_val, 0.02, "H on |0>: beta = 1/sqrt(2)");

            // Apply H again: Should return to |0> (H^2 = I)
            $display("\nApplying Hadamard gate again");
            apply_op(OP_H);
            display_state("After H^2 (should be |0>)");

            alpha_real = q88_to_real(alpha);
            beta_real = q88_to_real(beta);
            check_value(alpha_real, 1.0, 0.02, "H^2: alpha should be 1");
            check_value(beta_real, 0.0, 0.02, "H^2: beta should be 0");
        end
    endtask

    // Test 6: Hadamard on |1>
    task test_hadamard_on_one;
        real alpha_real, beta_real;
        begin
            $display("TEST 6: Hadamard Gate on |1>");
            // Initialize to |0> then flip to |1>
            apply_op(OP_INIT);
            apply_op(OP_X);
            display_state("Initial state |1>");

            // Apply H gate: |1> -> (|0> - |1>)/sqrt(2)
            apply_op(OP_H);
            display_state("After H on |1>");

            alpha_real = q88_to_real(alpha);
            beta_real = q88_to_real(beta);

            check_value(alpha_real, 0.707106781, 0.02, "H on |1>: alpha = 1/sqrt(2)");
```

```
            check_value(beta_real, -0.707106781, 0.02, "H on |1>: beta = -1/sqrt(2)");
        end
    endtask

    // Main Test Sequence
    initial begin
        // Initialize
        test_count = 0;
        pass_count = 0;
        fail_count = 0;
        rst = 1;
        opcode = OP_NOP;

        $display("\n");
        $display("  Single-Qubit Hardware Accelerator - Comprehensive Test");
        $display("Q8.8 Fixed-Point Format:");
        $display("  Resolution: 2^-8 = %.8f", 1.0/256.0);
        $display("  Range: [-128.0, 127.996]");

        // Reset sequence
        #20;
        rst = 0;
        #20;
        // Run all tests
        test_fp_multiplier();
        test_fp_adder_subtractor();
        test_pauli_x();
        test_pauli_z();
        test_hadamard();
        test_hadamard_on_one();
        // Final summary
        $display("\n");
        $display("TEST SUMMARY");
        $display("Total Tests:  %0d", test_count);
        $display("Passed:       %0d", pass_count);
        $display("Failed:       %0d", fail_count);
        $display("Pass Rate:    %.1f%%", (pass_count * 100.0) / test_count);

        $finish;
    end

    // Timeout watchdog
    initial begin
        #100000;
        $display("\n[ERROR] Testbench timeout!");
        $finish;
    end

endmodule
```

Listing 23: Accelerator Testbench (quantumn_testbench.v)

A.2 Booth Multiplier

```
module booth_multiplier_16bit(
    input clk,
    input start,
    input signed [15:0] multiplicand,
    input signed [15:0] multiplier,
    output reg signed [31:0] product,
    output reg done
);
    reg signed [15:0] M;
    reg signed [15:0] Q;
    reg signed [15:0] A;
    reg Qm1;
    // Q(-1)
    reg [4:0] count;        // 16 iterations (0-15)

    always @(posedge clk) begin
        if (start) begin
            A     <= 16'd0;
            M     <= multiplicand;
            Q     <= multiplier;
            Qm1   <= 1'b0;
            count <= 5'd16;
            done  <= 1'b0;
        end
        else if (count > 0) begin
            case ({Q[0], Qm1})
                2'b01: A <= A + M; // Add M
                2'b10: A <= A - M; // Subtract M
                default: ;         // Do nothing
            endcase

            // Arithmetic shift right {A, Q, Qm1}
            Qm1 <= Q[0];
            Q   <= {A[0], Q[15:1]};
            A   <= {A[15], A[15:1]};

            count = count - 1;
        end
        else begin
            done    <= 1'b1;
            product <= {A, Q};       // Final result
        end
    end
endmodule
```

Listing 24: 16-bit Booth Multiplier (booth_multiplier_16bit.v)

```
`timescale 1ns/1ps

module tb_booth_multiplier_16bit;

    reg clk;
    reg start;
    reg signed [15:0] multiplicand;
    reg signed [15:0] multiplier;

    wire signed [31:0] product;
    wire done;

    // Instantiate DUT (Device Under Test)
    booth_multiplier_16bit DUT (
        .clk(clk),
        .start(start),
        .multiplicand(multiplicand),
        .multiplier(multiplier),
        .product(product),
        .done(done)
    );
    // Clock generation
    always #5 clk = ~clk; // 10ns period
```

```
24
25      initial begin
26          $display("=== Booth Multiplier 16-bit Testbench Started ===");
27          // Initial values
28          clk = 0;
29          start = 0;
30          multiplicand = 0;
31          multiplier = 0;
32
33          // TEST 1: 5 * 7 = 35
34          #10;
35          multiplicand = 5;
36          multiplier = 7;
37          start = 1;
38          #10 start = 0;
39
40          wait(done == 1);
41          #5;
42          $display("Test 1: %d * %d = %d", 5, 7, product);
43          // TEST 2: (-8) * 6 = -48
44          #20;
45          multiplicand = -8;
46          multiplier = 6;
47          start = 1;
48          #10 start = 0;
49
50          wait(done == 1);
51          #5;
52          $display("Test 2: %d * %d = %d", -8, 6, product);
53          // TEST 3: (-12) * (-9) = 108
54          #20;
55          multiplicand = -12;
56          multiplier = -9;
57          start = 1;
58          #10 start = 0;
59
60          wait(done == 1);
61          #5;
62          $display("Test 3: %d * %d = %d", -12, -9, product);
63          // TEST 4: Max*Max (stress test)
64          #20;
65          multiplicand = 16'sh7FFF; // 32767
66          multiplier   = 16'sh7FFF; // 32767
67          start = 1;
68          #10 start = 0;
69
70          wait(done == 1);
71          #5;
72          $display("Test 4: %d * %d = %d", multiplicand, multiplier, product);
73
74          $display("=== Testbench Finished ===");
75          $stop;
76      end
77  endmodule
```

Listing 25: Booth Multiplier Testbench (tb_booth_multiplier_16bit.v)

A.3 Fixed-Point Arithmetic Units

```
`timescale 1ns / 1ps

// Fixed-point ADD unit
// RESULT = A + B
module add #(
    parameter WIDTH = 16
)(
    input  wire [WIDTH-1:0] A,
    input  wire [WIDTH-1:0] B,
    output wire [WIDTH-1:0] RESULT,
    output wire             COUT    // Carry-out (overflow)
);
    wire [WIDTH:0] sum_ext;

    assign sum_ext = {1'b0, A} + {1'b0, B};
    assign RESULT  = sum_ext[WIDTH-1:0];
    assign COUT    = sum_ext[WIDTH];
endmodule
```

Listing 26: Fixed-Point Adder (add.v)

```
`timescale 1ns/1ps

module tb_add;
    reg  [15:0] A, B;
    wire [15:0] RESULT;
    wire        COUT;
    add tb_add (
        .A(A),
        .B(B),
        .RESULT(RESULT),
        .COUT(COUT)
    );
    initial begin
        $dumpfile("add.vcd");
        $dumpvars(0, tb_add);

        A = 16'd10; B = 16'd5;
        #10;
        $display("ADD: %0d + %0d = %0d  COUT=%b", A, B, RESULT, COUT);

        A = 16'd0; B = 16'd0;
        #10;
        $display("ADD: %0d + %0d = %0d  COUT=%b", A, B, RESULT, COUT);

        $finish;
    end
endmodule
```

Listing 27: Adder Testbench (tb_add.v)

```
`timescale 1ns / 1ps

// Fixed-point SUB unit
// RESULT = A - B (two's complement)
module sub #(
    parameter WIDTH = 16
)(
    input  wire [WIDTH-1:0] A,
    input  wire [WIDTH-1:0] B,
    output wire [WIDTH-1:0] RESULT,
    output wire             COUT    // 1 = no borrow, 0 = borrow
);
    wire [WIDTH:0] diff_ext;

    assign diff_ext = {1'b0, A} + {1'b0, (~B)} + 1'b1;
    assign RESULT   = diff_ext[WIDTH-1:0];
    assign COUT     = diff_ext[WIDTH];
endmodule
```

Listing 28: Fixed-Point Subtractor (sub.v)

```
`timescale 1ns / 1ps

module tb_sub;
    reg  [15:0] A, B;
    wire [15:0] RESULT;
    wire        COUT;

    sub tb_sub (
        .A(A),
        .B(B),
        .RESULT(RESULT),
        .COUT(COUT)
    );
    initial begin
        $dumpfile("sub.vcd");
        $dumpvars(0, tb_sub);

        A = 16'd10; B = 16'd5;
        #10;
        $display("SUB: %0d - %0d = %0d  COUT=%b", A, B, RESULT, COUT);

        A = 16'd5; B = 16'd10;
        #10;
        $display("SUB: %0d - %0d = %0d  COUT=%b", A, B, RESULT, COUT);

        $finish;
    end
endmodule
```

Listing 29: Subtractor Testbench (tb_sub.v)

B Contribution Summary

The successful completion of the Single-Qubit Hardware Accelerator project was the result of a collaborative effort. The specific contributions of each team member are outlined below:

Nguyen Son Tung
Responsible for developing the theoretical foundation of the project. He conducted research on the underlying principles relevant to the system, synthesized key concepts into a coherent framework, and designed the overall project structure. His work ensured that the project had a solid academic grounding and a logical, well-organized architecture.

Nguyen Truong Son
Contributed by implementing the Q8.8 fixed-point format and developing the decoder application. He worked on converting numerical values into the Q8.8 representation and ensured accurate decoding functionalities. His contributions strengthened the project's practical computation and data-handling capabilities.

Tran Vu Gia Huy
Focused on the theoretical explanation of Booth's Algorithm and created the corresponding Verilog demonstration. He provided a clear analysis of how Booth's multiplication algorithm works and implemented a functional hardware-level model. His work bridged theory and practice, allowing the team to validate algorithmic behavior through simulation. Prepare final report for a good format.

Tran Dang Khoa
Responsible for simulating quantum gates. He studied the operational characteristics of fundamental quantum logic gates and developed simulations that visualize their behavior. His contribution added a modern computing perspective and expanded the technical scope of the project.

Mai Tran Hung
Implemented the addition and subtraction modules. He designed the logic for arithmetic operations, tested their correctness, and ensured compatibility with other components of the system. His work formed a crucial part of the computational pipeline within the project.