Instituto Tecnológico
y de Estudios Superiores de Occidente

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación el 29 de noviembre de 1976.

Departamento de Electrónica, Sistemas e Informática

Maestría en Diseño Electrónico

Hardware Feasibility of Neural Network Branch Predictor

TRABAJO RECEPCIONAL que para obtener el **GRADO** de
MAESTRO EN DISEÑO ELECTRÓNICO

Presenta: **Alberto Vargas Garrido**

Director: **Carlos Humberto Vázquez Tello**

Tlaquepaque, Jalisco. June 14, 2026

To my mother and father, whose example as role models made me who I am today; to my brother, who taught me responsibility and how to care for others; and to my dog Luna, for her loyalty. For their unconditional love and support, and for being there every step of the way.

Abstract

AI has taken the engineering and computer science industry by storm due to its adaptability and convenience. Its use is now an expectation, and the protagonist is the transformer, as it is used in LLMs. However, the simple perceptron still has a role to play, and not all problems need a massive neural network. With this in mind, the purpose of this project is to simulate and implement a branch predictor using perceptron-based neural networks in System Verilog for a RISC-V processor design. The goal of this project is not necessarily to prove that a neural network is better than traditional branch predictor algorithms like TAGE, but to identify its uses and strengths relative to them, if any exist.

In simple terms, a branch predictor's job, as its name implies, is to predict the result of a conditional branch in the executed code. It has the necessary elements to exploit the perceptron's ability to adjust its weights to reduce error by tracking correct and incorrect predictions. Two implementations were made: a Multi-Layer Perceptron and a Single-Layer Perceptron neural network, where the SLP surpasses the Bimodal and 2-bit branch predictors used for comparison, with results of SLP 98.72% / 98.22% vs Bimodal 97.72% / 95.13%. The caveat is less maximum frequency, as SLP has ∼30% more ALMs.

This work combines two areas of engineering: neural networks, which are state-of-the-art in computer science, and branch predictors, which are a fundamental part of computer architecture. It could serve as a didactic tool to teach students how a branch predictor works, with a fairly competent implementation, or as an embedded system that needs a small branch predictor.

Contents

Introduction

Modern branch predictors in the industry achieve impressive accuracy to remain competitive. However, their hardware requirements and complexity make them utilize a large area on silicon, which some applications can't afford.

TAGE-SC-L is the model to beat in this investigation. It won the 2016 CBP competition and is still used currently. The gap this case study addresses is to compare the NN implementation against TAGE using the CBP simulator and identify its strengths and weaknesses relative to TAGE's accuracy. The purpose is to create a simpler branch predictor; this way, it could be used in smaller designs. Another problem current processor designs face is the deep pipeline architecture required to achieve higher clock frequencies, which significantly complicates the design. On top of that, adding a complex branch predictor makes it a herculean task for an individual student to tackle.

This document presents the investigation and implementation of the Neural Network branch predictor for the CBP simulator, along with its RTL implementation in SystemVerilog. It shows results with the testbenches developed and comparisons between various branch predictors. For simulation, TAGE was used as a baseline for comparison, and a couple of implementations were tested to determine which was worth implementing in System Verilog. In the hardware design, two Neural Networks were used: MLP and SLP, and for traditional approaches, Bimodal and 2-bit branch predictors were used. The practical use has already been demonstrated by AMD's use of NNs in their Zen 1 and 2 architectures for their branch predictors [Martínez-24]; the scope of this case study is to implement it from scratch, examine its characteristics, and assess its viability as a small branch predictor for simpler designs. TAGE was not built as it was outside of the scope of this project.

The case study is organized in four chapters. The first is an overview of the concepts presented throughout the next chapters, so the reader doesn't get lost and can follow the logic. Chapter 2 provides the theoretical foundation for the hardware-conscious NN predictor design and explains where AI assistance was used in the implementation (Section 2.9). The chapter focuses on further explaining the AI concepts that will be implemented into the design. Chapter 3 describes the SystemVerilog RTL implementation, including architecture evolution, module hierarchy, and synthesis considerations. Chapter 4 analyses experimental results, quantifies the performance-feasibility trade-off, and discusses implications for constrained processor design. The Conclusions section that follows summarises findings and outlines directions for future research.

Four appendices follow. Appendix A reproduces the complete fixed-point class and sigmoid implementation, the offline weight generator, and sample hex-file excerpts. Appendix B

lists per-trace performance results across all 51 CBP traces. Appendix C documents the base OOO Tomasulo core specification, including the ISA subset, pipeline organization, and structural parameters. Appendix D reproduces the assembly source of the test programs run inside that core.

Study Case Conclusion: The results of this project show a marginal win for the SLP branch predictor over the next-best, the Bimodal branch predictor: 98.72% versus 97.72% on normal patterns, and 98.22% versus 95.13% on XOR patterns. The interesting part is that SLP does not depend on Program Counter input or an external Global History Register; it only needs the past predictions and the current one to accurately predict it. Of course, there are drawbacks for SLP: it needs, in its current state, 33% more ALMs than its competitor, and that affects the max clock frequency, making it 3 times slower. Another caveat is that SLP needs more data to surpass Bimodal. A short burst of branches or a short code execution won't benefit from the increased accuracy that longer executions do.

Special thanks to the thesis advisor and the review committee at ITESO for their technical guidance. This project benefited greatly from their knowledge. In addition, thanks to NC State University and the CBP team for the Championship Branch Prediction (CBP2025) simulator framework, a fundamental pillar of this work, and for AI assistance, which was instrumental for iterative testing, as disclosed in Section 2.9.

1. Background and Related Work

1.1. Classical Branch Predictors

A typical branch predictor has 2 components: the predictor table and the Branch Target Buffer (BTB). Both use the Program counter as input. The BTB hit with the table last time prediction selects between the BTB output and PC + 1 (Fig. 1-1). The different architectures change the table algorithm, but all use this simple concept as a base.

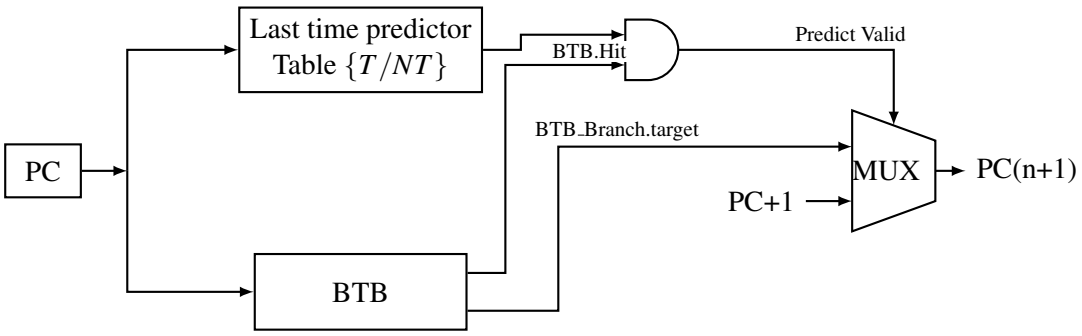

Fig. 1-1 Generic branch predictor architecture, drawn here as a 1-level Last Time Predictor. Adapted from [Martínez-24].

Classical branch predictors fall into three families ordered by increasing sophistication. The first uses static heuristics or single-bit bimodal counters. The second introduces history-indexed pattern tables. The third combines multiple predictors through a meta-predictor.

1.1.1 Static and Bimodal Predictors

Early branch predictors employed static heuristics (e.g., backward branches were predicted to be taken) or single-bit bimodal predictors [Smith-95]. These achieved 70–85% accuracy but failed to capture dynamic patterns.

1.1.2 Two-Level Adaptive Predictors

Two-level predictors [Yeh-91] introduced the concept of using branch history to index into pattern history tables (PHT). The key insight: branch outcomes correlate with recent history. Variants include:

- **Local History:** Per-branch history registers (BHR)
- **Global History:** Shared global history register (GHR)
- **Gselect:** GHR XORed with PC for indexing
- **Gshare:** GHR XORed with PC for both indexing and prediction

Gshare predictors achieve 90–95% accuracy with modest hardware (8–16KB), making them popular in commercial processors.

1.1.3 Hybrid and Tournament Predictors

Recognizing that different branches exhibit distinct patterns, hybrid predictors combine multiple prediction mechanisms with a meta-predictor that selects between them [McFarling-93]. A tournament predictor is a specific type of hybrid predictor that uses a table of 2-bit saturating counters, indexed by branch address, to dynamically select between two or more component predictors based on which has been more accurate for each branch. The Alpha 21264 [Kessler-99] pioneered this approach, combining local and global predictors with a tournament selector.

1.2. The TAGE Family

The TAGE family of predictors achieves high accuracy at substantial hardware cost. This section traces the family's origins, describes the TAGE-SC-L variant in detail, and characterizes the performance-versus-complexity trade-off it presents.

1.2.1 TAGE Origins and Evolution

The TAGE (TAgged GEometric history length) predictor, introduced by Seznec and Michaud [Seznec-06], revolutionized branch prediction through several key innovations:

Geometric History Lengths: Instead of a single history length, TAGE employs multiple tables indexed by histories of geometrically increasing lengths. This captures both short-term and long-term correlations.

Tagged Entries: Each prediction table stores partial PC tags alongside prediction counters, reducing aliasing and enabling confident predictions even with limited storage.

Useful Counters: Adaptive replacement policy tracks the usefulness of each entry, evicting unhelpful predictions.

1.2.2 TAGE-SC-L Architecture

The TAGE-SC-L predictor extends the base TAGE with statistical correction and loop prediction:

TAGE Component. The base TAGE predictor consists of 36 tagged prediction tables organized into banks, indexed by a hash of the Program Counter and the global branch history, with geometrically increasing lengths ranging from 6 to over 3000 bits. Each table entry contains a 3-bit prediction counter, a partial tag for aliasing reduction, and a useful bit for adaptive replacement. The predictor selects its prediction from the longest matching history table (provider component), with the second-longest match (altpred component) serving as backup [Seznec-06].

Statistical Corrector (SC). The SC component corrects systematic biases in the TAGE prediction using multiple GEHL predictors that index bias tables based on various features, including local, global, and path history. A signed perceptron-like summation across these tables produces a correction signal, and a threshold-based decision determines whether to override the TAGE prediction.

Loop Predictor. The loop predictor detects counted loops by tracking iteration counts and maintaining confidence levels, overriding the TAGE prediction when high-confidence loop behavior is identified.

1.2.3 TAGE Performance and Complexity

TAGE-SC-L won the Championship Branch Prediction 2016 competition with a misprediction rate of less than 1% across many traces [Seznec-16a]. However, this performance comes at substantial hardware cost: the design fully utilizes a 64KB storage budget across 36 tagged tables plus statistical corrector and loop predictor tables, requires complex indexing and hashing functions with multiple table accesses per prediction, and results in thousands of lines of synthesizable HDL that are difficult to verify.

1.3. Neural Network Branch Predictors

Neural-network branch predictors have been studied across several architectures with varying hardware feasibility. This section reviews the perceptron predictor, multi-layer per-

ceptron variants, spiking and convolutional approaches, and the small body of prior work that evaluates hardware cost rather than software accuracy alone.

1.3.1 Perceptron Predictors

Jiménez and Lin introduced perceptron predictors [Jiménez-01], applying simple neural networks to branch prediction. Subsequent work refined path-based neural prediction [Jiménez-03]. A perceptron is a linear classifier computing:

$$y = w_0 + \sum_{i=1}^{n} w_i \cdot h_i \tag{1-1}$$

where $h_i \in \{-1, +1\}$ represents global history bits, and w_i are learned weights. Prediction: taken if $y \geq 0$, not-taken otherwise.

The perceptron predictor offers several advantages: it can capture long histories spanning hundreds of bits, its training rule is simple ($w_i \leftarrow w_i + t \cdot h_i$, where t is the target outcome), and its regular structure maps well to hardware. However, as a linear model, it can only capture linearly separable branch patterns in the history-bit feature space and requires signed multiply-accumulate operations.

1.3.2 Multi-Layer Perceptron (MLP) Predictors

Extending to multi-layer networks adds a hidden layer with non-linear activations:

$$h_j = \sigma\left(\sum_i w_{ij} x_i + b_j\right) \quad \text{(hidden layer)} \tag{1-2}$$

$$y = \sigma\left(\sum_j v_j h_j + c\right) \quad \text{(output layer)} \tag{1-3}$$

where σ is a non-linear activation function (sigmoid, tanh, ReLU).

Multi-layer networks can, in principle, learn non-linear functions of their inputs [LeCun-15] through implicit feature learning in hidden layers and can be trained via gradient-based backpropagation [LeCun-98]. However, hardware implementation faces significant challenges: sigmoid and tanh activation functions require exponential computation or lookup tables, backpropagation adds substantial hardware complexity, and standard floating-point arithmetic is expensive in silicon.

1.3.3 Recurrent Neural Network (RNN) Predictors

Recurrent Neural Networks (RNNs) retain hidden state across inputs. This lets them learn temporal patterns in sequential data. The SRNN, a branch predictor explored in the literature [Zhang-20], was evaluated in floating-point software simulation without area or power estimates. As with the MLP, the recurrent weights and activation functions involve multiplications and non-linear functions that are expensive in hardware, so this case study does not pursue a recurrent architecture. Given the high cost of recurrent weights and activation functions in hardware, this case study sticks with the single-layer perceptron lineage described in Section 1.3.1.

1.3.4 Spiking Neural Network Predictors

Spiking neural networks use biologically inspired spiking neurons characterized by membrane potential accumulation, threshold-based firing, and temporal coding. Their event-driven nature offers low power potential and the ability to learn temporal patterns. The spiking branch predictor, however, has received limited attention: the only published evaluation reviewed in this case study is the spiking-style predictor of Tarsa and Pouchet [Tarsa-15]. This case study does not pursue spiking dynamics; the predictor implemented in Chapter 2 is a single-layer perceptron with hardware-friendly tanh activation, in the lineage of the Jiménez and Lin perceptron predictor described in Section 1.3.1 and constrained for synthesisability rather than for biological plausibility.

1.3.5 Convolutional Neural Network (CNN) Predictors

Recent work has explored CNN-based branch predictors that apply convolutional filters to history sequences, enabling local pattern detection and hierarchical feature extraction [Zangeneh-20; Fang-23]. However, CNNs face challenges for branch prediction, including high computational cost, large parameter counts, and limited demonstrated benefit over simpler architectures.

1.3.6 Prior Hardware NN Implementations

Prior work is limited in its ability to address the hardware feasibility of neural network branch predictors. Zangeneh et al. [Zangeneh-20] and Zhang et al. [Zhang-20] both evalu-

ate their CNN and SRNN predictors using floating-point software simulation, without reporting area or power estimates, leaving the practical implementability question unanswered. The perceptron predictor proposed by Jiménez and Lin [Jiménez-01] is the closest to a hardware-feasible design, as its dot-product computation maps naturally to signed multiply-accumulate units, but it still requires multipliers, and its linear architecture can only learn branch patterns that are linearly separable in the history-bit feature space.

Research Gap: No comprehensive hardware-conscious NN predictor design achieving near-state-of-the-art performance with dramatic complexity reduction has been reported. This case study addresses this gap by implementing both MLP and SLP branch predictors with design constraints explicitly targeting synthesizability and hardware efficiency.

2. Theoretical Foundation and Methodology

Note on implementation languages: Throughout this chapter, C++ is used for the reference software implementation (training and simulation), while SystemVerilog is used for the synthesizable RTL hardware implementation. Code listings are labeled accordingly.

2.1. Hardware-Conscious Design Philosophy

2.1.1 Design Constraints and Objectives

The current trend in neural network research is to increase the number of parameters and complexity, requiring greater computational performance [LeCun-15]. The branch predictors implemented in this investigation face the opposite. Strict limitations: be hardware-friendly; no use of complex functions like tanh or sigmoid for activation; and, if possible, no use of multiplications or divisions. For now, the design will be monolithic, as seen in Hennessy and Patterson [Hennessy-17]: the first RTL pass in SystemVerilog is as simple as possible, and the pipeline is introduced in later work and implementations. This ensures that the next restriction is met: 1 to 2 cycles of latency for synchronization between the branch predictor and the processor's fetch request, with the design simulatable in ModelSim or Verilator and validated by a timing analysis.

Restrictions on area, timing, and power consumption change the approach to implementing a neural network. In software, complex operations and algorithms such as floating-point arithmetic, dynamic memory allocation, and complex activation functions are trivial to implement compared to their hardware counterparts. This means there must be a trade-off: accuracy. Decreasing accuracy will permit the design to fit the criteria.

2.1.2 Design Objectives

As mentioned earlier, the NN branch predictor implementation needs to satisfy power, latency, and area constraints; the latter is the most difficult, as it determines whether the design is worth implementing in silicon. There is no point in implementing a 99% accurate solution that requires an area that is prohibitive. Reaching 95% accuracy would be an acceptable trade-off compared with state-of-the-art branch predictors that reach near 99% accuracy; 95% at

0.3mm^2 leaves room for errors and next steps.

2.2. Fixed-Point Arithmetic System

2.2.1 Motivation: The Cost of Floating-Point

Standard neural networks employ 32-bit (single precision) or 64-bit (double precision) floating-point arithmetic, as specified by the IEEE 754 standard [IEEE-08]. A single 32-bit floating-point multiplier requires on the order of 2000–5000 logic gates for the mantissa multiplier core, plus additional logic for exponent alignment, normalization, rounding, and special-case handling for infinity, NaN, and subnormals. The figure is scaled from the 54×54-bit double-precision implementations reported in Weste and Harris [Weste-10, Section 11.9, Table 11.16]. Scaling the published 90 nm area data of Table 11.16 to 65 nm and to a 24×24-bit mantissa multiplier places the area at roughly 0.01–0.02 mm^2, with 2–3 cycle latency typical for pipelined floating-point multipliers in this range [Hennessy-17, Appendix J].

The 256→32→1 MLP architecture developed in Section 2.7, which $(256 \times 32) + 32 + (32 \times 1) + 1 = 8{,}257$ weights and biases parameterize, requires thousands of multiply-accumulate (MAC) operations per forward pass. Even with aggressive resource sharing, the arithmetic datapath dominates area and power.

2.2.2 Fixed-Point Alternative

Fixed-point arithmetic represents numbers as scaled integers. A $Q_{m.n}$ fixed-point format allocates m bits for the integer part and n bits for the fractional part, with one sign bit.

The advantages are substantial: integer multipliers are 5–10× smaller than their floating-point equivalents [Horowitz-14], no exponent logic, normalization, or special case handling is needed, behavior is fully deterministic (simplifying verification), and power consumption is reduced. The challenges are a fixed dynamic range requiring careful selection of m and n, the need for explicit overflow and underflow handling, and the potential for quantization error accumulation across successive operations.

2.2.3 28-bit Fixed-Point Format

After exploring 8-bit, 16-bit, and 32-bit formats, a 28-bit format with 10-bit integer and 18-bit fractional parts was selected, using two's complement representation for signed values.

The 8-bit format provided insufficient precision for gradient computation (quantization errors exceeding 1%), while the 16-bit format lacked the dynamic range needed for weight accumulation during training. The 32-bit format offered adequate precision but consumed unnecessary area without measurable accuracy improvement. The 28-bit format strikes the optimal balance:

$$\text{Value} = \frac{\text{raw\_value}}{2^{18}} = \frac{\text{raw\_value}}{262144} \tag{2-1}$$

Range:

$$\text{Maximum} = \frac{2^{27}-1}{2^{18}} = 511.999996 \tag{2-2}$$

$$\text{Minimum} = \frac{-2^{27}}{2^{18}} = -512.0 \tag{2-3}$$

Precision:

$$\text{Smallest increment} = 2^{-18} \approx 3.8 \times 10^{-6} \tag{2-4}$$

This format provides sufficient range for weight values typically in $[-5, +5]$, adequate precision for gradient-based learning ($< 10^{-5}$), fits in 32-bit registers with 4 spare bits for headroom, and uses a power-of-2 scale factor that enables bit-shift conversion.

2.2.4 Fixed-Point Arithmetic Operations

Addition and Subtraction Fixed-point addition is identical to integer addition with saturation:

```
FixedPoint28 operator+(const FixedPoint28& other) const {
    int64_t result = static_cast<int64_t>(value_) +
                     static_cast<int64_t>(other.value_);

    // Saturate to [MIN_VAL, MAX_VAL]
    if (result > MAX_VAL) result = MAX_VAL;  // 2^27 - 1
    if (result < MIN_VAL) result = MIN_VAL;  // -2^27

    return FixedPoint28(static_cast<int32_t>(result));
}
```

Listing 2.1: Fixed-Point Addition (C++)

Saturation prevents overflow, which is crucial for numerical stability during weight updates.

Multiplication Fixed-point multiplication requires rescaling since multiplying two scaled values doubles the scale factor:

$$\frac{a}{2^{18}} \times \frac{b}{2^{18}} = \frac{a \times b}{2^{36}} \neq \frac{a \times b}{2^{18}} \tag{2-5}$$

To maintain the same scale, a right-shift by 18 bits is applied:

```
FixedPoint28 operator*(const FixedPoint28& other) const {
    // Multiply in 64-bit to avoid intermediate overflow
    int64_t product = static_cast<int64_t>(value_) *
                      static_cast<int64_t>(other.value_);

    // Rescale by shifting right 18 bits
    int64_t result = product >> 18;

    // Saturate to valid range
    if (result > MAX_VAL) result = MAX_VAL;
    if (result < MIN_VAL) result = MIN_VAL;

    return FixedPoint28(static_cast<int32_t>(result));
}
```

Listing 2.2: Fixed-Point Multiplication (C++)

Note that the intermediate product of two 28-bit values requires 56 bits before the right-shift, which is why the computation uses 64-bit arithmetic. Values outside the 10-bit integer range or requiring precision beyond the 18-bit fractional part are lost in this truncation. The right-shift itself is a trivial hardware operation (wire routing) with zero logic cost.

2.2.5 Quantization Error Analysis

Fixed-point arithmetic introduces quantization error at each operation. For a single multiplication:

$$\text{Error} = \left| \frac{(a \times b) \bmod 2^{18}}{2^{18}} \right| < 2^{-18} \tag{2-6}$$

For a network with L layers and W weights, the worst-case accumulated error is $O(LW \cdot 2^{-18})$. With the proposed architecture ($L = 2$, $W = 8257$), maximum accumulated error is:

$$\text{Max error} \approx 2 \times 8257 \times 2^{-18} \approx 0.063 \tag{2-7}$$

Since sigmoid outputs range $[0, 1]$ and the prediction threshold is 0.5, this error is acceptable (6.3% of full scale). This error margin is consistent with the quantization analysis for fixed-point neural network implementations described by Horowitz [Horowitz-14].

2.2.6 Implementation: FixedPoint28 Class

The complete C++ reference implementation encapsulates 28-bit fixed-point arithmetic in a class (see Listings 2.1 and 2.2). Key features include explicit conversion from double (for initialization), overloaded operators (+, -, *, +=) for natural syntax, automatic saturation on all operations, and a raw value accessor for hardware interfacing.

This C++ abstraction enables writing the neural network logic at a high level while maintaining deterministic fixed-point semantics. The same arithmetic operations are then replicated in the SystemVerilog RTL implementation (Chapter 3), ensuring bit-exact correspondence between simulation and hardware.

2.3. Divider-less Logic Design

2.3.1 The Cost of Division

Even simple division operations are expensive in hardware. Iterative dividers require n cycles for n-bit division, combinational dividers have $O(n^2)$ area complexity, and reciprocal approximation still requires a multiplier.

A 28-bit divider requires 1500–2500 logic gates and multi-cycle latency, as characterized by Horowitz [Horowitz-14], making it unacceptable for a 1-cycle prediction operation.

2.3.2 Bit-Shift Division

Division by powers of 2 is trivial in hardware - simply a right-shift (wire routing):

$$\frac{x}{2^k} = x \gg k \quad \text{(arithmetic right shift)} \tag{2-8}$$

This operation has zero logic cost (pure wiring), zero latency (combinational), and produces an exact result for integer x.

2.3.3 Application: Sigmoid Approximation Slope

One natural target for the right-shift trick is the sigmoid's linear region. A piecewise-linear sigmoid common in fixed-point hardware implementations takes the form:

$$\sigma(x) \approx 0.5 + \frac{x}{8} \quad \text{for } |x| < 4 \tag{2-9}$$

Division by 8 is equivalent to right-shift by 3:

```
// Traditional approach (requires divider):
// result = x / 8;

// Multiplier-less approach (pure wiring):
int32_t result = x >> 3;  // Right shift by 3 bits
```

Listing 2.3: Divider-less Division by 8 (C++)

This eliminates a 28-bit divider (∼2000 gates saved), produces an immediate result with zero latency, and consumes zero dynamic power since there is no switching activity.

This single optimization removes one of the most expensive operations from the critical path.

2.3.4 Generalization: Powers-of-2 Design

The design systematically uses powers of 2 for all division operations: the learning rate is 2^{-10} (right-shift by 10), weight decay is 2^{-17} (right-shift by 17), the sigmoid slope is 2^{-3} (right-shift by 3), and bias scaling uses powers of 2 wherever possible.

This philosophy eliminates all dividers and most multipliers from the design, leaving only essential weight multiplication operations.

2.4. Activation Functions in Hardware

2.4.1 The Activation Function Challenge

Neural networks rely on non-linear activation functions in their hidden layers, and the standard choices impose different hardware costs. The sigmoid $\sigma(x) = 1/(1+e^{-x})$ needs exponential computation (Taylor series, CORDIC, or lookup table), a division, and range reduction for numerical stability; the Taylor-series option demands many multipliers with high latency, CORDIC requires 15–20 iterations to converge, and lookup tables need 256–1024 entries with interpolation, placing the resulting unit at roughly 0.005–0.01 mm^2. Tanh, $\tanh(x) = (e^x - e^{-x})/(e^x + e^{-x})$, costs even more because the same machinery has to compute two exponentials. ReLU, $\max(0,x)$, is the cheap end of the spectrum, requiring only a single comparator, but it suffers from dead neurons and vanishing gradients in deep networks [Glorot-10].

2.4.2 SLP Output: Piecewise-Linear Tanh

The single-layer perceptron uses a piecewise-linear tanh in the output stage:

$$f_{\text{SLP}}(x) = \begin{cases} -1 & x \leq -4 \\ x/4 & -4 < x < 4 \\ +1 & x \geq 4 \end{cases} \tag{2-10}$$

The linear region is a right-shift by 2 in fixed-point, and the saturation regions are the comparators. The output range is $[-1,+1]$. The prediction is taken when the output is positive and not-taken when the output is negative, so the decision threshold is 0. The SystemVerilog implementation in 'snn_branch_predictor.sv' is:

```
// Tanh activation
if (output_sum.val > (4 << 18)) begin
  mlp_result = FP_ONE;
end else if (output_sum.val < -(4 << 18)) begin
  mlp_result = '{val: -FP_ONE.val};
end else begin
  mlp_result = '{val: output_sum.val >>> 2};
end
```

Listing 2.4: Piecewise-Linear Tanh Activation (SLP)

2.4.3 MLP Output: Sigmoid Look-Up Table

The multi-layer perceptron takes a different approach. A 256-entry lookup table holds the true sigmoid $\sigma(x) = 1/(1+e^{-x})$ sampled at 256 points covering the input range $[-8,+8]$, stored in Q10.18 fixed-point format. The LUT is generated offline by 'generate_weights.py' (Appendix A, Listing A.3) from the floating-point sigmoid expression and is loaded into the RTL at reset via `$readmemh("sigmoid_lut.hex", ...)`; an excerpt and the shared hex-file encoding are reproduced in Appendix A, Section A.4.

The hidden layer of the MLP does not use the LUT. Each hidden neuron applies a step activation, `(sum > 0) ? FP_ONE : FP_ZERO`, which produces binary intermediate values that drive the output layer.

The output index is computed using a right-shift and an offset, then clamped to the table bounds. The SystemVerilog in 'mlp_branch_predictor.sv' is:

```
shifted_sum    = out_temp.val >>> SIGMOID_INPUT_SHIFT;    // SHIFT = 14
lut_idx_signed = shifted_sum + SIGMOID_OFFSET;            // OFFSET =
    128

```

```
if (lut_idx_signed < 0)
  sigmoid_index = 8'd0;
else if (lut_idx_signed >= SIGMOID_LUT_SIZE)              // SIZE = 256
  sigmoid_index = 8'd255;
else
  sigmoid_index = lut_idx_signed[SIGMOID_LUT_BITS-1:0];

mlp_result = '{val: sigmoid_lut[sigmoid_index]};
```

Listing 2.5: Sigmoid LUT Lookup (MLP Output)

The LUT step is $16/256 = 0.0625$ in input, so the output is piecewise constant between samples. The prediction is taken when the sigmoid output is at or above 0.5 and not-taken below.

2.4.4 Training-Time Derivative

Backpropagation uses a constant derivative in place of the true derivative of the forward activation. The constant value avoids the vanishing-gradient effect that occurs when a true sigmoid or tanh derivative saturates near ± 4 and removes the multiplication that would otherwise sit on the training datapath. The two predictors use different constants, each matching the linear-region slope of its forward activation.

The SLP uses $1/4$, which is the slope of its piecewise-linear tanh:

```
output_delta = '{val: output_error.val >>> 2};  // 1/4
```

Listing 2.6: SLP Training Derivative

The MLP uses $1/8$:

```
output_delta = '{val: output_error.val >>> 3};  // 1/8
```

Listing 2.7: MLP Training Derivative

Both shifts are wires, so the training stage carries no multiplier at this point.

2.5. Weight Initialization and Storage

2.5.1 Pre-training and Runtime Adaptation

Both the MLP and SLP predictors use a two-stage training scheme: offline pre-training produces initial weight values in software, and runtime backpropagation refines those values during execution. This hybrid arrived through an empirical path rather than an a priori design analysis.

The RTL was first implemented as a pure runtime-learning design, with weights initialized to zero and updated only at runtime on every `train_valid` pulse. In early experiments, the pure-runtime configuration did not converge reliably on the benchmark workloads, so weights that had been trained offline in the C++ reference implementation on the CBP trace suite were exported to hex files, quantized to the Q10.18 fixed-point format, and loaded into the RTL's weight registers via `$readmemh` on reset. The runtime backpropagation path was retained; it continues to refine the weights during execution, starting from the imported operating point rather than from zero. With that warm start in place, the design produced the results reported in Chapter 4, and the same configuration is used throughout the remainder of this case study.

The offline step serves two purposes. It delivers what offline training traditionally provides, and it offers a practical initialization that sidesteps the convergence issues observed with zero initialization in the pure-runtime experiments. Offline training in this case covers training on the full 51-trace suite, hyperparameter optimization via genetic algorithms, multiple independent runs with selection of the best weights, and regularisation techniques such as dropout, weight decay, and early stopping [Han-16; Courbariaux-15]. The runtime step then provides adaptation to workload drift, phase changes, and branch patterns that are under-represented in the offline corpus. The live weights sit in flip-flop registers throughout both inference and training; the hex files are read once on reset and are not accessed afterward, so there is no ongoing ROM-versus-SRAM trade-off in either loop.

2.5.2 Weight File Format

In the C++ reference implementation that performs the offline pre-training, weights are stored as double-precision floating-point compile-time constants in a header file. These values are quantized to the Q10.18 fixed-point format at the end of the training pipeline and written to hex files, which the RTL reads on reset via `$readmemh` to populate the weight and bias registers. The double-precision format is kept in the C++ reference only to preserve maximum precision during training and during the conversion to fixed point:

```
// rom_weights.h - Generated from best training run
#ifndef ROM_WEIGHTS_H
#define ROM_WEIGHTS_H

// Input -> Hidden weights [32][256]
static constexpr double ROM_WEIGHTS_IH[32][256] = {
    { 0.012845, -0.003821, 0.028472, ... },  // Hidden neuron 0
    { -0.015293, 0.041827, -0.002134, ... }, // Hidden neuron 1
    // ... 30 more rows
};

// Hidden biases [32]
```

```
static constexpr double ROM_BIAS_H[32] = {
    0.105832, -0.082743, 0.043921, ...
};

// Hidden -> Output weights [1][32]
static constexpr double ROM_WEIGHTS_HO[1][32] = {
    { 0.283741, -0.194832, 0.371924, ... }
};

// Output bias [1]
static constexpr double ROM_BIAS_O[1] = { -0.052183 };

#endif
```

Listing 2.8: ROM Weights Header Structure (C++)

In the C++ reference, these doubles are converted to fixed-point at initialization. For the SystemVerilog synthesis flow, the quantized values are exported as hex files and consumed by `$readmemh` at reset, populating the weight registers with their pre-trained values before any runtime training pulse arrives.

2.5.3 Weight Quantization

In the C++ reference implementation, the initial weights are stored as doubles and converted to FixedPoint28 during the one-shot initialization step that precedes either offline training (when the reference network is being retrained) or a simulation run:

```
void load_rom_weights() {
    // Load Input->Hidden weights
    for (int j = 0; j < hidden_size_; ++j) {
        for (int i = 0; i < input_size_; ++i) {
            weights_ih_[j][i] = FixedPoint28(ROM_WEIGHTS_IH[j][i]);
        }
    }
    // ... (biases and output weights similarly)
}
```

Listing 2.9: Weight Loading (C++)

This conversion happens once during initialization. During operation, only fixed-point values are used.

2.5.4 Weight Storage Analysis

The total storage requirement is $8192 + 32 + 32 + 1 = 8{,}257$ parameters, comprising 8,192 input-to-hidden weights (256×32), 32 hidden biases, 32 hidden-to-output weights (32×1), and 1 output bias.

At 28 bits per parameter:

$$\text{Weight storage} = 8257 \times 28 \text{ bits} = 231{,}196 \text{ bits} \approx 29 \text{ KB} \tag{2-11}$$

Compare to TAGE-SC-L [Seznec-16a]: 64 KB (2.2× larger).

2.6. SystemVerilog Synthesis Compatibility

2.6.1 Design for Synthesis Principles

Hardware synthesis tools (Synopsys Design Compiler, Cadence Genus) impose strict constraints on synthesizable code: no dynamic allocation (`new`, `malloc`, or STL containers), array dimensions must be compile-time constants, no pointer arithmetic or dynamic dispatch, no floating-point arithmetic (unless targeting FPGAs with hard float cores), and all operations must have bounded execution time to ensure deterministic behavior.

2.6.2 Fixed-Size Array Architecture

Maximum dimensions are defined as compile-time constants in both C++ (for simulation) and SystemVerilog (for synthesis):

```
class MLP {
    // Compile-time constants (constexpr)
    static constexpr int MAX_FEATURES = 256;  // Input size
    static constexpr int MAX_HIDDEN = 32;     // Hidden layer size
    static constexpr int MAX_OUTPUT = 1;      // Output size (binary)

    // Fixed-size arrays (synthesis-friendly)
    FixedPoint28 weights_ih_[MAX_HIDDEN][MAX_FEATURES]; // [32][256]
    FixedPoint28 bias_h_[MAX_HIDDEN];                   // [32]
    FixedPoint28 weights_ho_[MAX_OUTPUT][MAX_HIDDEN];   // [1][32]
    FixedPoint28 bias_o_[MAX_OUTPUT];                   // [1]

    // Runtime dimensions (must be <= MAX_*)
```

```
    int input_size_;   // 256
    int hidden_size_;  // 32
    int output_size_;  // 1
};
```

Listing 2.10: Fixed-Size Array Declarations (C++)

With these dimensions baked in at compile time, the synthesis tool knows the exact memory footprint without runtime allocation overhead, and the arrays map directly to register files or SRAM blocks. Timing analysis is then deterministic.

2.6.3 Constexpr for Hardware Constants

The C++ reference implementation uses `constexpr` extensively for hardware constants, which map directly to `localparam` declarations in the SystemVerilog implementation:

```
class FixedPoint28 {
private:
    static constexpr int FRACTIONAL_BITS = 18;
    static constexpr int SCALE = 1 << FRACTIONAL_BITS;  // 262144
    static constexpr int MAX_VAL = 134217727;
    static constexpr int MIN_VAL = -134217728;
    // ...
};
```

Listing 2.11: Constexpr Constants (C++)

Values declared this way are computed at compile time and folded into constants by the synthesis tool, leaving no runtime computation or storage requirement and a correspondingly cleaner netlist.

2.6.4 Bit-Exact Semantics

Fixed-point arithmetic is bit-exact: the same input always produces the same output, with no rounding-mode variability (unlike floating-point). Software simulation and hardware therefore behave identically, which is what makes the simulate-then-synthesize verification flow described below reliable.

This is critical for hardware validation. The design can be simulated in software, results verified, and then synthesized with confidence that hardware behavior will match.

2.6.5 Synthesis Flow

The design methodology follows a two-track approach. First, a C++ reference implementation is developed and validated through software simulation, ensuring correct fixed-point arithmetic and neural network behavior. Then, the design is manually translated to SystemVerilog RTL, preserving bit-exact semantics. The C++ implementation confines itself to operations with direct hardware equivalents (fixed-size arrays, integer arithmetic, bit shifts, comparators), ensuring that the translation to SystemVerilog is straightforward.

The synthesizable SystemVerilog is then processed through the standard FPGA synthesis flow: RTL simulation with Verilator for functional verification, followed by logic synthesis and place-and-route using Intel Quartus targeting the Cyclone V FPGA.

2.7. Complete MLP Architecture

The architecture is a feedforward neural network consisting of an input layer, a hidden layer with a non-linear activation function, and a single output neuron that produces the taken/not-taken prediction. Each layer uses the 28-bit Q10.18 fixed-point format described in Section 2, and all division operations are implemented as bit shifts.

2.7.1 Network Topology

The theoretical MLP architecture consists of three layers. The **input layer** accepts 256 features drawn from multiple sources: 64 bits from the Global History Register (GHR), 32 bits of local per-branch history (hashed), 16 bits of path history, 16 bits of PC hash, and 128 additional bits encoding bimodal counters and confidence signals. The **hidden layer** contains 32 neurons, each fully connected to all 256 inputs and using the piecewise linear sigmoid activation described in Section 2. The **output layer** is a single neuron, fully connected to the 32 hidden neurons, also using piecewise linear sigmoid activation. A branch is predicted taken if the output value ≥ 0.5, not-taken otherwise.

Note that this theoretical 256→32→1 architecture represents the maximum-complexity design point. As discussed in Chapter 3, empirical evaluation showed that simpler architectures (8→128→1 and 128→1) achieved superior accuracy due to better generalization with limited training data.

2.7.2 Forward Propagation

Algorithm 1 Forward Propagation (Prediction)

Require: Input features $x[0..255]$, weights $W^{(1)}$, $W^{(2)}$, biases $b^{(1)}$, $b^{(2)}$
Ensure: Prediction y (taken/not-taken)

// Hidden layer computation
for $j = 0$ to 31 **do**
    $sum_j \leftarrow b_j^{(1)}$
    for $i = 0$ to 255 **do**
        $sum_j \leftarrow sum_j + W_{ji}^{(1)} \times x_i$
    end for
    $h_j \leftarrow \sigma(sum_j)$ ▷ Piecewise linear sigmoid
end for
// Output layer computation
$out \leftarrow b_0^{(2)}$
for $j = 0$ to 31 **do**
    $out \leftarrow out + W_{0j}^{(2)} \times h_j$
end for
$y \leftarrow \sigma(out)$ ▷ Piecewise linear sigmoid
return $(y \geq 0.5)$ ▷ Predict taken if $y \geq 0.5$

The computational complexity comprises 8,192 MACs for the hidden layer (32×256), 32 MACs for the output layer (1×32), 2 sigmoid evaluations, and 1 comparison, totaling 8,224 MACs per prediction.

2.7.3 Hardware Timing

For a 1-cycle prediction at 3GHz, the available time per cycle is 333 picoseconds. A highly pipelined MAC array with 8–16 stages hides the total latency of 8–16 cycles, achieving a throughput of 1 prediction per cycle once the pipeline is filled.

This matches or exceeds TAGE latency (multi-cycle table accesses).

2.8. Training Methodology

2.8.1 Offline Training Process

Training occurs during the design phase using powerful workstations and follows a seven-step process. First, 40 training traces are executed to collect (features, outcome) pairs

(the trace inventory is reproduced in Appendix B, Table XII). These are processed into 256-dimensional feature vectors. The network is initialized with small random weights using the Xavier initialization method. The training loop then iterates through forward propagation, binary cross-entropy loss computation, backpropagation, and weight updates using the Adam optimizer [Kingma-14]. Hyperparameters are optimized using a genetic algorithm [Goldberg-89] over learning rate, decay, and other parameters; the genetic search configuration is described in Section 2.8.2. The resulting float64 weights are quantized to FixedPoint28 format, and the final weights are exported to a 'rom_weights.h' header file for ROM generation.

2.8.2 Genetic Algorithm Optimization

Genetic algorithms are used to optimize hyperparameters across a multi-dimensional search space. The optimized parameters include learning rate ($[10^{-4}, 10^{-1}]$), weight decay ($[10^{-6}, 10^{-3}]$), dropout rate ($[0.0, 0.3]$), GHR length ($[16, 64]$), PHT size ($[1024, 8192]$), and the number of hidden neurons ($[16, 64]$). The genetic algorithm uses a population of 20 configurations evolved over 50 generations, with a mutation rate of 0.1 and a crossover rate of 0.8.

This approach follows the hyperparameter optimization methodology used in neural network training [Kingma-14]. The best configuration found was: learning rate 0.001 (implemented as a bit-shift operation for hardware compatibility), 32 hidden neurons, GHR length of 64 bits, and 3 full training passes over the dataset.

2.9. Use of AI in Implementation

The base out-of-order RISC-V Tomasulo core with ROB described in Appendix C was authored by hand; the remaining implementation components were produced with the assistance of generative AI tools. The per-component breakdown is given in Table I.

TABLE I
PROVENANCE OF THE IMPLEMENTATION COMPONENTS SUPPORTING THIS CASE STUDY.

Component	Provenance
Base OOO Tomasulo RISC-V core with ROB	Authored by hand
Bug fixes applied to the base core after its initial implementation	AI-assisted
Championship Branch Prediction iteration	AI-assisted
Branch-predictor RTL (Bimodal / 2-bit FSM / SLP / MLP)	AI-generated
Simulation automation and sweep scripts	AI-generated
ASM-to-HEX encoder	AI-generated
Testbenches that drive the automation flow	AI-generated

3. SystemVerilog RTL Implementation

This chapter documents the RTL design described theoretically in Chapter 2. Two implementations are showcased: the Multi-Layer Perceptron (MLP) branch predictor and its simplified derivative, the Single-Layer Perceptron (SLP), following the restrictions described in the last chapter.

Technology reference: Throughout this chapter, gate-count estimates use a 65nm CMOS process as the reference, consistent with Horowitz [Horowitz-14]. Actual synthesis targets the Intel Cyclone V FPGA (5CSEMA5F31C6, 32,070 ALMs) using Quartus Prime under the Slow 1100mV 85°C timing corner. Area estimates at 65nm assume approximately 10 gates per μm^2 for standard cell implementations.

3.0.1 Implementation Architecture Evolution

From Theory to Practice The theoretical design presented in Section 2.7 describes a 256→32→1 architecture with extensive feature extraction from multiple history sources, inspired by the perceptron branch predictor of Jiménez and Lin [Jiménez-01] and subsequent neural network approaches [Zangeneh-20; Zhang-20]. For RTL implementation, two reduced variants were carried forward to keep the design within feasible silicon limits: an 8→128→1 MLP (1,281 parameters) and a 64→1 SLP (65 parameters). Even with these reductions, the MLP does not produce a synthesis result on the Cyclone V target. The fitter runs for hours without converging, and the design's logic estimate already exceeds the device's ALM capacity (Section 4.5); only the SLP synthesizes successfully. The remainder of this chapter describes the SystemVerilog RTL for both variants and the trade-offs each entails relative to the theoretical design.

3.0.2 Module Hierarchy and Organization

Self-Contained Monolithic Architecture Both predictors are implemented as self-contained monolithic modules. Rather than decomposing the network into separate perceptron units and layer modules, each predictor contains all its logic within a single SystemVerilog module. That logic covers feature extraction, forward pass, activation, and training.

The original design followed a bottom-up modular approach inspired by the processor's own register file architecture: individual `register` modules (analogous to the architectural

registers x0, a1, etc.) were composed into `perceptron_unit` modules, which were in turn instantiated by `perceptron_layer` modules via `generate` blocks. This hierarchical decomposition reflected standard hardware design methodology and enabled clear separation of concerns. However, during synthesis with Intel Quartus targeting the Cyclone V FPGA, the modular design produced synthesis failures and poor quality-of-results related to multi-dimensional array indexing and complex write-enable inference across module boundaries. Consolidating the logic into monolithic modules resolved these issues and produced clean, predictable synthesis results suitable for the timing and area analysis presented in Chapter 4.

Whether the synthesis difficulties stemmed from a fundamental limitation of the hierarchical approach to neural network weight storage or from an implementation issue that could be resolved with a different module decomposition strategy remains an open question. Future work could revisit the modular design with alternative synthesis tools or different hierarchical partitioning to determine whether the bottom-up approach can achieve equivalent synthesis quality.

Both predictors maintain their own internal branch history and require no external GHR or PC signals.

The **MLP predictor** ('mlp_branch_predictor.sv') implements an 8→128→1 architecture within a single module. It maintains an internal branch history as a shift register that shifts in `train_actual` on each training event. Feature extraction converts each history bit to a fixed-point (`FP_ONE` or `FP_ZERO`) combinationally. The hidden layer instantiates 128 neurons via a `generate` block, each computing a conditional weighted sum with ReLU activation. The output layer accumulates over the 128 hidden activations and applies a sigmoid through a 256-entry lookup table loaded via `$readmemh`. A single pipeline register separates the forward pass from the output. Weights are initialized from hex files and updated via backpropagation on `train_valid`, using shift-based sigmoid and ReLU derivatives, conditional weight updates, and shift-based learning rate and weight decay. Fig. 3-1 shows the resulting RTL structure: the prediction path runs vertically down the left column from the 8-bit internal history through the 128-neuron hidden layer, ReLU, pipeline register, output layer, and the sigmoid LUT, while the right column carries the three ROM-initialized storage banks (hidden weights, output weights, and the sigmoid LUT) and the dual training loop that updates both weight banks on every `train_valid` pulse.

The **SLP predictor** ('snn_branch_predictor.sv') implements a simpler 64→1 single-layer architecture. It shares the same internal history, feature extraction, pipeline register, and hex-file weight initialization patterns as the MLP. The single neuron computes a conditional weighted sum across 64 features with tanh activation (saturation at ± 1, linear region sum$/4$). Training uses gradient descent with a tanh derivative approximation (arithmetic right shift by 2), conditional weight updates, and shift-based learning rate and weight decay. Fig. 3-2 shows the resulting RTL block structure: the prediction path runs vertically down the left column from the internal history register through feature extraction, the weighted-sum unit, and the

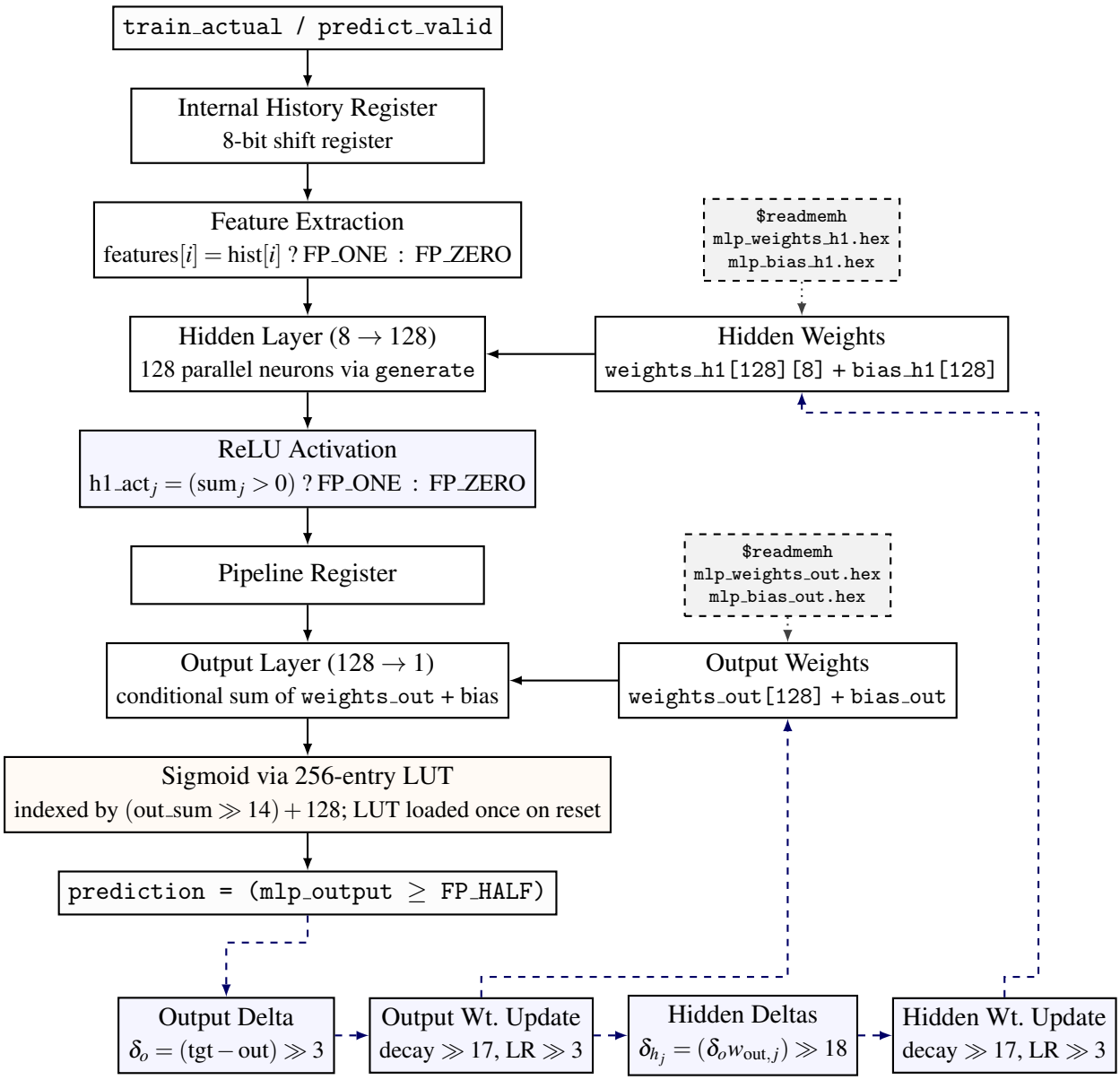

Fig. 3-1 RTL flow diagram of the Multi-Layer Perceptron (MLP) branch predictor ('src/BRANCH_PREDICTOR/mlp_branch_predictor.sv'). Solid arrows: forward prediction data path (left column). Dotted arrows: one-time `$readmemh` initialisation on reset (three ROM banks: hidden weights, output weights, and the sigmoid LUT). Dashed arrows: runtime backpropagation feedback (bottom row, active on `train_valid`). The output delta drives the output weight update and propagates through the output weights into the hidden-layer deltas, which then drive the hidden weight update; each hidden delta is gated by the corresponding ReLU sign in the actual SystemVerilog.

tanh activation, while the right column carries the one-time ROM initialization and the runtime backpropagation feedback loop that adjusts the weight registers on every `train_valid` pulse.

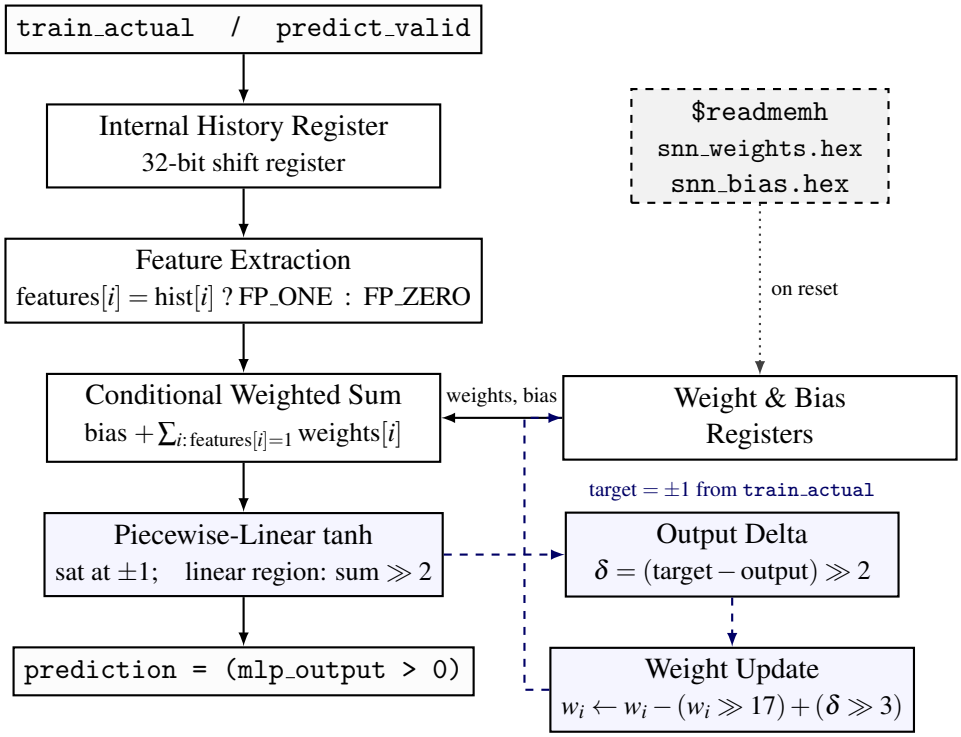

Fig. 3-2 RTL flow diagram of the Single-Layer Perceptron (SLP) branch predictor ('src/BRANCH_PREDICTOR/snn_branch_predictor.sv'). Solid arrows: forward prediction data path. Dotted arrow: one-time `$readmemh` weight initialisation on reset. Dashed arrows: runtime backpropagation feedback (active on `train_valid`).

Interface Specifications Top-Level SLP/MLP Predictor Interface:

```
module snn_branch_predictor #(
  parameter int NUM_FEATURES = 32,   // Past outcomes to track
  parameter int GHR_LEN = 32         // Unused - interface
   compatibility
)(
  input  logic clk,
  input  logic rst_n,

  // Global History input (UNUSED - tracks its own)
  input  logic [GHR_LEN-1:0] global_history,

  // Prediction interface
  input  logic        predict_valid,
  input  logic [63:0] pc,              // UNUSED
  output logic        prediction,
  output logic        prediction_valid,
  output fixed_point_t mlp_output,
```

```
17
18    // Training interface
19    input  logic        train_valid,
20    input  logic [63:0] train_pc,        // UNUSED
21    input  logic        train_actual,    // The actual branch outcome
22
23    // Control
24    output logic        ready
25 );
```

Listing 3.1: Branch Predictor Interface (from snn_branch_predictor.sv)

The interface exposes `pc`, `global_history`, and `train_pc` ports for compatibility with the processor's branch prediction infrastructure, but the self-contained predictors leave them unused. Prediction is requested via `predict_valid`, with results returned via `prediction` (binary taken/not-taken) and `mlp_output` (28-bit fixed-point confidence). Training triggers runtime weight updates and internal history shifts via `train_valid` and `train_actual`. The internal history register is updated on every training event, replacing the need for an external GHR.

3.0.3 Forward Pass

Feature Extraction and Forward Pass Both predictors use the same feature-extraction approach: each bit of the internal history register is converted to a fixed-point value (`FP_ONE` or `FP_ZERO`) for use in the weighted-sum computation. This is implemented inline within each predictor module rather than as a separate feature extractor:

```
1  // Internal branch history - tracks actual outcomes
2  logic [NUM_FEATURES-1:0] internal_history;
3
4  // Feature extraction - combinational
5  fixed_point_t features [NUM_FEATURES];
6  always_comb begin
7    for (int i = 0; i < NUM_FEATURES; i++) begin
8      features[i] = internal_history[i] ? FP_ONE : FP_ZERO;
9    end
10 end
```

Listing 3.2: Inline Feature Extraction (from snn_branch_predictor.sv)

The SLP forward pass computes a single weighted sum with conditional accumulation, exploiting the binary nature of features to eliminate multipliers entirely:

```
1  // Single-layer forward pass
2  fixed_point_t output_sum;
3  always_comb begin
```

```
  automatic fixed_point_t sum_temp;
  sum_temp = bias;
  for (int i = 0; i < NUM_FEATURES; i++) begin
    if (features[i] == FP_ONE) begin
      sum_temp = fp_add(sum_temp, weights[i]);
    end
  end
  output_sum = sum_temp;
end

// Tanh activation with saturation
fixed_point_t mlp_result;
always_comb begin
  if (output_sum.val > (4 << 18)) begin
    mlp_result = FP_ONE;                          // Saturate to +1
  end else if (output_sum.val < -(4 << 18)) begin
    mlp_result = '{val: -FP_ONE.val};             // Saturate to -1
  end else begin
    mlp_result = '{val: output_sum.val >>> 2};    // Divide by 4 (shift
   )
  end
end
```

Listing 3.3: SLP Forward Pass with Conditional Accumulation

Instead of full 28-bit multiplication, weights are conditionally accumulated only when the corresponding feature equals 1. A traditional implementation would require N multipliers plus N adders, whereas this binary optimization reduces the hardware to N adders and N comparators, yielding approximately 50% area reduction.

MLP Hidden Layer with Generate Block The MLP predictor uses a `generate` block to instantiate 128 hidden neurons in parallel, each computing its own weighted sum and ReLU activation:

```
// Hidden Layer (8->128) + ReLU
genvar h1;
generate
  for (h1 = 0; h1 < HIDDEN1_NEURONS; h1++) begin : gen_h1
    always_comb begin
      automatic fixed_point_t sum_temp;
      sum_temp = bias_h1[h1];
      for (int i = 0; i < NUM_FEATURES; i++) begin
        if (features[i] == FP_ONE) begin
          sum_temp = fp_add(sum_temp, weights_h1[h1][i]);
        end
      end
      h1_sums[h1] = sum_temp;
```

```
      h1_activations[h1] = (sum_temp.val > 0) ? FP_ONE : FP_ZERO;
    end
  end
endgenerate
```

Listing 3.4: MLP Hidden Layer (from mlp_branch_predictor.sv)

All 128 neurons compute their weighted sums in parallel within the same cycle. The ReLU activation is a binary step function (output is `FP_ONE` or `FP_ZERO`), requiring only a single sign-bit comparator per neuron.

MLP Output Layer with Sigmoid LUT The MLP output layer uses a 256-entry lookup table for the sigmoid activation, loaded from a hex file at initialization:

```
// Output layer (128->1) + Sigmoid via LUT
always_comb begin
  automatic fixed_point_t out_temp;
  automatic logic signed [31:0] shifted_sum;
  automatic logic signed [31:0] lut_idx_signed;

  out_temp = bias_out;
  for (int i = 0; i < HIDDEN1_NEURONS; i++) begin
    if (h1_activations[i] == FP_ONE) begin
      out_temp = fp_add(out_temp, weights_out[i]);
    end
  end
  output_sum = out_temp;

  // Sigmoid LUT indexing
  shifted_sum = out_temp.val >>> SIGMOID_INPUT_SHIFT;
  lut_idx_signed = shifted_sum + SIGMOID_OFFSET;

  if (lut_idx_signed < 0)
    sigmoid_index = 8'd0;
  else if (lut_idx_signed >= SIGMOID_LUT_SIZE)
    sigmoid_index = 8'd255;
  else
    sigmoid_index = lut_idx_signed[SIGMOID_LUT_BITS-1:0];

  mlp_result = '{val: sigmoid_lut[sigmoid_index]};
end
```

Listing 3.5: MLP Output Layer with Sigmoid LUT (from mlp_branch_predictor.sv)

The sigmoid LUT approach trades 256 ROM entries for zero computational cost at inference time, avoiding both the piecewise linear approximation and the expensive exponential logic.

Internal History Register Rather than using a separate `global_history_register` module, both predictors maintain their internal history inline, shifting in the actual branch outcome on each training event:

```
// Internal history update - shift in actual branch outcome
always_ff @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    internal_history <= '0;
  end else if (train_valid) begin
    internal_history <= {internal_history[NUM_FEATURES-2:0],
   train_actual};
  end
end
```

Listing 3.6: Internal History Update (shared pattern)

Hardware cost: an N-bit shift register (64 flip-flops for SLP, 8 for MLP). The history updates only on `train_valid`, ensuring that the predictor's view of branch history matches the actual outcomes returned by the pipeline.

3.0.4 Training Path and Arithmetic

Weight Initialization from Hex Files Weights are pre-trained offline and loaded on reset from hex files using `$readmemh`, which gives the predictor a useful operating point from the first cycle. The hex-loaded values seed the weight registers; runtime backpropagation (shown later in this chapter) then refines them as training pulses arrive on `train_valid`. A full 'snn_weights.hex' is reproduced in Appendix A, Listing A.5:

```
// ROM arrays for initial weights
logic signed [27:0] weights_rom [NUM_FEATURES];
logic signed [27:0] bias_rom [1];

initial begin
  $readmemh("snn_weights.hex", weights_rom);
  $readmemh("snn_bias.hex", bias_rom);
end

// Weight initialization on reset
always_ff @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    for (int f = 0; f < NUM_FEATURES; f++) begin
      weights[f] <= '{val: weights_rom[f]};
    end
    bias <= '{val: bias_rom[0]};
  end
```

```
  // ... training logic follows
end
```

Listing 3.7: Weight Loading from Hex Files (from snn_branch_predictor.sv)

The hex files are generated by the offline training pipeline, which trains on the full trace suite using powerful workstations and exports the best weights as fixed-point constants. The `$readmemh` mechanism is standard in both simulation and FPGA synthesis flows, ensuring that the same weight values are used in Verilator simulation and Quartus synthesis.

Backpropagation Training Logic The MLP predictor implements full backpropagation with gradient descent in an `always_ff` block triggered by `train_valid`:

```
end else if (train_valid) begin
  automatic fixed_point_t target;
  automatic fixed_point_t output_error;
  automatic fixed_point_t output_delta;
  automatic fixed_point_t h1_errors [HIDDEN1_NEURONS];
  automatic fixed_point_t h1_deltas [HIDDEN1_NEURONS];
  automatic fixed_point_t decayed_weight;
  automatic fixed_point_t updated_weight;

  target = train_actual ? FP_ONE : FP_ZERO;
  output_error = fp_sub(target, saved_mlp_result);
  output_delta = '{val: output_error.val >>> 3};

  // Backpropagate to hidden layer
  for (int j = 0; j < HIDDEN1_NEURONS; j++) begin
    h1_errors[j] = '{val: (output_delta.val *
                    weights_out[j].val) >>> 18};
    h1_deltas[j] = (saved_h1_sums[j].val > 0) ?
                   h1_errors[j] : FP_ZERO;
  end

  // Update output weights with decay
  for (int j = 0; j < HIDDEN1_NEURONS; j++) begin
    decayed_weight = fp_sub(weights_out[j],
                     '{val: weights_out[j].val >>> WEIGHT_DECAY_SHIFT})
   ;
    if (saved_h1_activations[j] == FP_ONE) begin
      updated_weight = fp_add(decayed_weight,
                       '{val: output_delta.val >>> LEARNING_RATE_SHIFT
   });
    end else begin
      updated_weight = decayed_weight;
    end
    weights_out[j] <= updated_weight;
```

```
  end
  bias_out <= fp_add(bias_out,
              '{val: output_delta.val >>> LEARNING_RATE_SHIFT});

  // Update hidden layer weights similarly...
end
```

Listing 3.8: MLP Backpropagation Training (from mlp_branch_predictor.sv)

Several optimizations keep the training logic hardware-friendly. The sigmoid derivative is approximated as a constant $1/8$ (arithmetic right shift by 3), eliminating the need for multiplication during backpropagation. The ReLU derivative is binary (0 or 1) based on the sign of the weighted sum, requiring only a single comparator. Weight updates are conditional on feature values: only active inputs (feature $= 1$) receive gradient updates, while inactive inputs (feature $= 0$) receive only weight decay, reducing switching activity and dynamic power consumption. The learning rate $2^{-3} = 0.125$ and weight decay $2^{-17} \approx 7.6 \times 10^{-6}$ are both implemented as arithmetic right shifts, incurring zero additional hardware cost beyond the shift logic already present in the datapath.

SLP Training (Simpler):

The SLP variant uses a simplified single-layer training algorithm with a Tanh derivative:

```
target = train_actual ? FP_ONE : FP_NEG_ONE;  // Target: +1 or -1
output_error = fp_sub(target, saved_mlp_result);
output_delta = '{val: output_error.val >>> 2};  // Tanh deriv approx

// Update bias
bias <= fp_add(bias, '{val: output_delta.val >>> LEARNING_RATE_SHIFT});

// Update weights with conditional add and decay
for (int i = 0; i < NUM_FEATURES; i++) begin
  decayed_weight = fp_sub(weights[i],
                   '{val: weights[i].val >>> WEIGHT_DECAY_SHIFT});
  if (saved_features[i] == FP_ONE) begin
    updated_weight = fp_add(decayed_weight,
                     '{val: output_delta.val >>> LEARNING_RATE_SHIFT});
  end else begin
    updated_weight = decayed_weight;
  end
  weights[i] <= updated_weight;
end
```

Listing 3.9: SLP Training with Tanh Derivative (from snn_branch_predictor.sv)

Fixed-Point Arithmetic Package All implementations use a shared `fixed_point_pkg` SystemVerilog package defining the Q10.18 format (1 sign bit + 9 integer bits + 18 fractional bits,

as described in Section 2.2 of Chapter 2) and arithmetic operations. This package is the SystemVerilog equivalent of the C++ `FixedPoint28` class presented in Section 2.2.6 of Chapter 2:

```
package fixed_point_pkg;

  parameter int TOTAL_BITS = 28;
  parameter int FRAC_BITS = 18;
  localparam signed [27:0] MAX_VAL = 28'h7FFFFFF;
  localparam signed [27:0] MIN_VAL = 28'h8000000;

  typedef struct packed {
    logic signed [27:0] val;
  } fixed_point_t;

  // Constants
  localparam fixed_point_t FP_ZERO    = '{val: 28'h0000000};
  localparam fixed_point_t FP_ONE     = '{val: 28'h0040000}; // 1 << 18
  localparam fixed_point_t FP_NEG_ONE = '{val: 28'hFFC0000}; // -1.0
  localparam fixed_point_t FP_HALF    = '{val: 28'h0020000}; // 1 << 17

  // Addition with saturation
  function automatic fixed_point_t fp_add(
    fixed_point_t a, fixed_point_t b);
    logic signed [TOTAL_BITS:0] result;
    result = {a.val[TOTAL_BITS-1], a.val}
           + {b.val[TOTAL_BITS-1], b.val};
    if (result > MAX_VAL) return '{val: MAX_VAL};
    else if (result < MIN_VAL) return '{val: MIN_VAL};
    else return '{val: result[TOTAL_BITS-1:0]};
  endfunction

  // Subtraction with saturation
  function automatic fixed_point_t fp_sub(
    fixed_point_t a, fixed_point_t b);
    logic signed [TOTAL_BITS:0] result;
    result = {a.val[TOTAL_BITS-1], a.val}
           - {b.val[TOTAL_BITS-1], b.val};
    if (result > MAX_VAL) return '{val: MAX_VAL};
    else if (result < MIN_VAL) return '{val: MIN_VAL};
    else return '{val: result[TOTAL_BITS-1:0]};
  endfunction

  // Hardware-friendly sigmoid, ReLU, shift operations...
endpackage
```

Listing 3.10: Fixed-Point Package (from fixed_point_pkg.sv)

3.0.5 Synthesis Considerations

Synthesis Challenges with Arrays During development, several synthesis challenges were encountered with array-based weight storage:

Problem: Synthesis tools can struggle with multi-dimensional arrays in `always_ff` blocks, and Intel Quartus is particularly affected. The `for` loops inside `always_ff` are unrolled at synthesis time into parallel hardware, but when they index into multi-dimensional arrays, the tool must infer complex multiplexing logic for the write-enable and data paths of each element. This often results in synthesis failure or extremely poor-quality results (QoR). The following pattern caused synthesis errors:

```
// BAD: Multi-dimensional array in always_ff
always_ff @(posedge clk) begin
  for (int i = 0; i < NUM_NEURONS; i++) begin
    for (int j = 0; j < NUM_INPUTS; j++) begin
      weights[i][j] <= new_weights[i][j];  // Synthesis may fail or
   produce poor QoR
    end
  end
end
```

Listing 3.11: Unsynthesizable Pattern (Avoid)

Solution: In the MLP hidden layer, `generate` blocks create distinct combinational logic instances at elaboration time. For weight storage, the current implementation uses standard arrays in `always_ff` blocks (which modern Quartus handles correctly for the array sizes used), while the hidden layer forward pass uses generate blocks for parallel neuron instantiation:

```
// GOOD: Explicit register instantiation
genvar i, j;
generate
  for (i = 0; i < NUM_NEURONS; i++) begin : gen_neurons
    for (j = 0; j < NUM_INPUTS; j++) begin : gen_inputs
      register #(.n(28)) weight_reg (
        .clk(clk),
        .reset(rst_n),
        .enable(weight_update_en),
        .datainput(new_weights[i][j].val),
        .dataoutput(weight_registers[i][j])
      );
    end
  end
endgenerate
```

Listing 3.12: Synthesizable Pattern (Preferred)

This pattern guarantees that each neuron's computation is independent, enabling parallel evaluation with a total combinational delay equal to a single neuron rather than $N\times$.

Generate Blocks for Parallelism As shown in Listing 3.4, the MLP hidden layer uses a `generate` block to create 128 parallel neuron instances, each with its own `always_comb` block for weighted sum computation and ReLU activation. After synthesis, all 128 neurons compute in parallel in a single cycle, with area scaling linearly with the neuron count.

Avoiding Unsynthesizable Constructs **No Dynamic Allocation:**

```
// BAD: Dynamic allocation (unsynthesizable)
reg [27:0] weights[];
initial weights = new[NUM_INPUTS];

// GOOD: Static allocation
reg [27:0] weights [NUM_INPUTS];
```

Listing 3.13: Static vs. Dynamic Allocation

No Real Arithmetic:

```
// BAD: Real arithmetic (unsynthesizable without HLS)
real weight_val = 0.5 * feature;

// GOOD: Fixed-point arithmetic
fixed_point_t weight_val = '{val: feature.val >>> 1};  // Divide by 2
```

Listing 3.14: Fixed-Point vs. Real

Constexpr for Compile-Time Constants:

```
// Constants computed at synthesis time
localparam int LEARNING_RATE_SHIFT = 3;          // LR = 2^-3 = 0.125
localparam int WEIGHT_DECAY_SHIFT = 17;          // Decay = 2^-17
localparam int FP_HALF = 131072;                 // 0.5 in Q10.18
```

Listing 3.15: Compile-Time Constants

Hardware Resource Implications The gate-count estimates presented below are analytical figures in the 65nm CMOS domain, assuming 10 gates per flip-flop (including clock distribution and reset logic) and adder/comparator gate counts from standard cell library characterization at 65nm [Horowitz-14; Hennessy-17]; multiplier-accumulate complexity follows the analysis by Jiménez and Lin [Jiménez-01] for perceptron branch predictor hardware. They are not directly comparable to the Cyclone V ALM counts reported in Chapter 4, because an ALM aggregates gates, flip-flops, and routing into a single coarser-grained unit that also carries the overhead of FPGA interconnect. The SLP estimate below ($\sim$7,000 gates) can at least be cross-checked against its synthesized ALM count (see Table VI), and the two line up to within

the expected ALM-aggregate-more-than-gates discrepancy. The MLP estimate, by contrast, is reported analytically only: as documented in Section 4.5, Quartus place-and-route does not converge on the MLP design within practical runtime, and the analytical estimate already exceeds the Cyclone V's ALM capacity, so its ∼80,000-gate figure has no silicon counterpart against which to validate it and should be read strictly as a pre-synthesis design-space estimate.

TABLE II
MLP PREDICTOR (8→128→1) GATE-COUNT ESTIMATE (65NM CMOS ANALYTICAL). TOTALS APPROXIMATELY 80,000 GATES AND ∼0.25–0.35 MM^2. SYNTHESIS TO CYCLONE V DID NOT COMPLETE FOR THIS DESIGN, SO THIS ESTIMATE IS NOT CORROBORATED BY SILICON.

Component	Quantity	Est. Gates	Total Gates
GHR (64-bit shift reg)	1	64 FFs	640
Feature extractor	8 mux	8 × 2	16
Hidden layer neurons	128	500 ea.	64 000
- Weight registers	128 × 8	28 FFs ea.	28 672
- Bias register	128	28 FFs	3 584
- Adder tree	128	200 ea.	25 600
- ReLU activation	128	10 ea.	1 280
Output layer neuron	1	5000	5 000
- Weight registers	128	28 FFs	3 584
- Bias register	1	28 FFs	28
- Adder tree	1	800	800
- Sigmoid activation	1	100	100
Training logic	1	10 000	10 000
Total			**∼80 000 gates**

TABLE III
SLP PREDICTOR (64→1) GATE-COUNT ESTIMATE (65NM CMOS ANALYTICAL). TOTALS APPROXIMATELY 7,000 GATES AND ∼0.025–0.04 MM^2, ROUGHLY 11× SMALLER THAN THE MLP ANALYTICAL ESTIMATE. THE SLP DID SYNTHESISE ON CYCLONE V (SEE CHAPTER 4).

Component	Quantity	Est. Gates	Total Gates
GHR (64-bit shift reg)	1	64 FFs	640
Feature extractor	64 mux	64 × 2	128
Output layer neuron	1	3000	3 000
- Weight registers	64	28 FFs	1 792
- Bias register	1	28 FFs	28
- Adder tree	1	600	600
- Tanh activation	1	80	80
Training logic	1	3000	3 000
Total			**∼7 000 gates**

For comparison, TAGE-SC-L [Seznec-16a] requires approximately 150,000 gates and

0.6–0.8 mm^2 at 65nm in published analytical estimates. Within the same analytical framework, the SLP at $\sim$7,000 gates ($\sim$0.025–0.04 mm^2) would be approximately 20$\times$ smaller than TAGE-SC-L. The MLP estimate at $\sim$80,000 gates ($\sim$0.25–0.35 mm^2) suggests a 2–3$\times$ analytical reduction, but because the MLP did not synthesize, this last comparison should be interpreted as a pre-synthesis projection rather than a validated result.

4. Results and Discussion

4.1. Experimental Setup

4.1.1 Simulation Framework

This case study uses the Championship Branch Prediction (CBP) simulation framework [Seznec-16b], which uses trace-driven simulation with instruction traces captured from SPEC CPU workloads. The suite has 51 traces that stress the branch predictors. The metrics of focus are MPKI (Mispredictions Per Kilo Instructions) and CycWPPKI (Cycles on Wrong-Path Per Kilo Instructions). As per the competition rules, the storage limit is 64KB. TAGE-SC-L uses the full budget, and the NN implementations use 27KB as a heuristic refinement after multiple design iterations.

4.1.2 Benchmark Traces

The CBP suite has 51 traces (disclosed in Appendix B, Table XII), divided into 40 traces for training and 11 for evaluation. Out of the 51 traces, 37 are integer (INT), and 14 are floating-point (FP). As in any traditional machine learning training, it combines both because diversity is necessary for better results.

4.1.3 Predictor Configurations

NN Predictor Variants The base MLP design is 'mlp_v8_clean'. Its architecture has 256 input features, 1 hidden layer with 32 neurons, and 1 output neuron, since it is a binary decision (taken or not-taken). Each parameter is a 28-bit fixed-point word. Building on this, 'mlp_v8_clean_lr_shift_3' adds an optimized shift-based learning rate. 'mlp_v8_clean_lr_shift_3_confidence' introduces prediction confidence tracking. The best performer is 'mlp_v8_clean_lr_shift_3_confidence_hist', which combines confidence with history features, achieving the best trace-simulation performance. Two SLP variants ('snn' and 'snn_tanh') provide single-layer perceptron baselines based on the latter hardware implementations. This happened because the SLP was tested heuristically later in the RTL implementation, way after the simulation part.

4.2. Sample Trace Results

Table IV presents results on two representative sample traces, providing a controlled initial comparison between all predictor variants.

TABLE IV
SAMPLE TRACE PERFORMANCE COMPARISON (2 TRACES) – CBP BENCHMARK

Predictor	MPKI	CycWPPKI	vs. TAGE
TAGE-SC-L (Baseline)	**0.709**	48.39	–
MLP (confidence + hist)	1.231	62.01	+73.6%
MLP (confidence only)	1.348	63.74	+90.1%
SLP (tanh activation)*	1.362	62.47	+92.1%
MLP (lr_shift_3)	1.366	66.94	+92.7%
SLP (basic)*	1.635	68.55	+130.6%
MLP (base clean)	2.002	80.17	+182.4%

**Note: SLP results in trace simulation were affected by training synchronization bugs later fixed in the hardware implementation. See Section 4.9 for corrected hardware results where SLP outperforms MLP.*

4.2.1 Impact on IPC

To assess how the full-dataset MPKI gap translates into real-world performance, consider a processor with a baseline IPC of 3.5, a 20-cycle misprediction penalty, and a branch frequency of 17% of instructions (typical for general-purpose workloads).

IPC with TAGE-SC-L:

$$\text{IPC}_{\text{TAGE-SC-L}} = \frac{3.5}{1 + 4.51 \times 20/1000} = \frac{3.5}{1.0902} = 3.210 \tag{4-1}$$

IPC with MLP:

$$\text{IPC}_{\text{MLP}} = \frac{3.5}{1 + 8.62 \times 20/1000} = \frac{3.5}{1.1724} = 2.985 \tag{4-2}$$

The resulting IPC difference is:

$$\frac{3.210 - 2.985}{3.210} = 7.01\% \text{ IPC loss} \tag{4-3}$$

The 91.1% higher MPKI translates to approximately 7% real-world IPC loss across the full trace suite. This is a more substantial penalty than the sample-trace analysis suggested,

reflecting the greater diversity of branch behaviors in the complete benchmark. The trade-off remains viable for resource-constrained applications where the area and power savings justify this performance gap.

4.3. Full Dataset Results

Table V summarizes results across all 51 traces (40 training + 11 test).

TABLE V
FULL DATASET PERFORMANCE (51 TRACES)

Predictor	Workload	Count	Avg. MPKI	Avg. CycWPPKI
TAGE-SC-L	INT	37	4.70	182.77
TAGE-SC-L	FP	14	4.01	167.14
TAGE-SC-L	**Overall**	**51**	**4.51**	**178.48**
MLP (best)	INT	37	9.07	285.06
MLP (best)	FP	14	7.43	265.65
MLP (best)	**Overall**	**51**	**8.62**	**279.74**
SLP	INT	37	10.76	317.07
SLP	FP	14	9.82	312.39
SLP	**Overall**	**51**	**10.50**	**315.78**

4.3.1 Workload Characterization

Integer vs. Floating-Point Divergence INT workloads exhibit 22.0% higher MPKI than FP workloads for the MLP predictor (9.07 vs. 7.43), and a similar trend holds for TAGE (4.70 vs. 4.01) and SLP (10.76 vs. 9.82). This divergence reflects a fundamental difference in control-flow structure: integer workloads feature more irregular branches driven by pointer chasing, data-dependent conditions, and complex conditional logic, whereas floating-point workloads tend toward structured loops from vector operations and matrix traversals. The consistency of this trend across all three predictors confirms that the difficulty is intrinsic to the workloads rather than an artifact of any particular predictor design.

Trace-by-Trace Variability Fig. 4-1 illustrates the MPKI distribution across all 51 traces for TAGE-SC-L, MLP, and SLP. The spread is wide, ranging from below 1 MPKI on the most predictable traces to nearly 29 MPKI on the most challenging cases, underscoring that no single accuracy number captures predictor behavior across diverse workloads. TAGE consistently achieves the lowest MPKI across all traces, but the gap varies substantially.

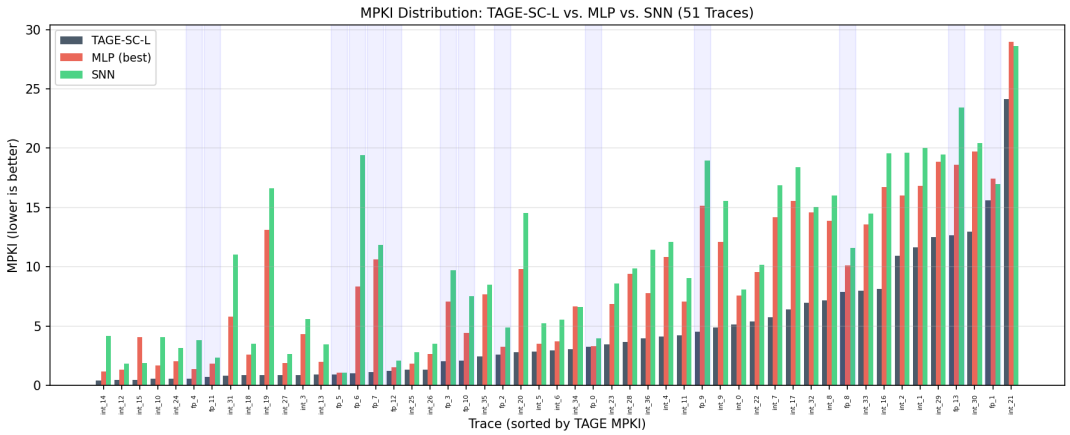

Fig. 4-1 MPKI Distribution: TAGE-SC-L vs. MLP vs. SLP across all 51 traces, sorted by TAGE MPKI. Blue-shaded columns indicate FP workloads. TAGE consistently achieves the lowest MPKI, with MLP and SLP showing larger gaps on difficult traces.

4.3.2 Best and Worst Case Analysis

With the trace (`fp_5_trace`), the MLP is able to shrink the gap with TAGE: 1.06 MPKI with MLP and 0.94 MPKI with TAGE. This test has highly regular, repetitive patterns, unlike the most challenging trace (`int_21_trace`). The MLP reaches 28.94 MPKI, against TAGE's 24.13 MPKI, still not a massive gap, but it widens it. This shows that any predictor will have problems with this test. Finally, MLP cannot surpass TAGE's accuracy on any trace, demonstrating that the TAGE architecture is still well thought out.

4.4. Hardware Complexity Analysis

4.4.1 FPGA Synthesis Results

Table VI presents the actual Quartus synthesis results for the complete OOO Tomasulo processor (described in Appendix C) with each branch predictor variant, targeting the Intel Cyclone V FPGA (5CSEMA5F31C6, 32,070 ALMs) under the Slow 1100mV 85°C timing corner.

TABLE VI
QUARTUS SYNTHESIS RESULTS: OOO PROCESSOR WITH DIFFERENT BRANCH PREDICTORS

Predictor	ALMs	ALM share	Registers	Fmax (MHz)	Synthesized
Bimodal	12706	40%	13008	13.44	Yes
Simple 2-bit	12587	39%	12991	13.63	Yes
SLP	**16858**	**53%**	**12696**	**4.48**	**Yes**
MLP	—	—	—	—	**No – did not converge**

Note: Synthesis results vary ±3% between Quartus runs due to non-deterministic place-and-route algorithms. TAGE-SC-L was not synthesized as it exists only as a software reference in the CBP benchmark framework and was outside the scope of this case study.

The SLP predictor synthesizes successfully but at a high cost: it requires 33% more ALMs than Bimodal (16,858 vs. 12,706) and operates at approximately one-third the clock frequency (4.48 MHz vs. 13.44 MHz). The register count is comparable across all predictors (∼12,700–13,000), indicating that the SLP's additional ALM usage comes primarily from combinational logic, namely the multiplexers and adder trees required for the weighted sum computation across 32 input features.

4.4.2 Frequency Impact Analysis

The SLP's lower operating frequency (4.48 MHz vs. Bimodal's 13.44 MHz) stems from its combinational critical path: the weighted sum must accumulate across all 32 features in a single cycle, creating a deep adder chain. In a pipelined ASIC implementation, this could be split across multiple pipeline stages to recover frequency, but on an FPGA, the single-cycle combinational design limits Fmax. The throughput remains 1 prediction per cycle once the pipeline is filled, so the frequency reduction translates directly into lower prediction bandwidth rather than increased prediction latency.

4.5. The Performance-Feasibility Trade-Off

4.5.1 Quantifying the Trade-Off

Fig. 4-2 plots accuracy against hardware cost for the predictors that were synthesized.

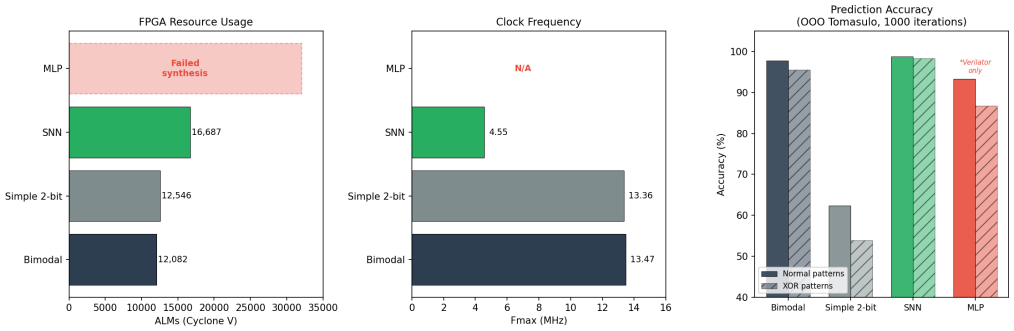

Fig. 4-2 Performance-Feasibility Trade-Off: ALM utilization vs. prediction accuracy on the OOO Tomasulo processor (Cyclone V FPGA). The SLP achieves higher accuracy on hardware test patterns at the cost of 33% more ALMs and 3× lower Fmax. The MLP exceeds device capacity and its synthesis does not converge in practical runtime, so it has no on-silicon point. TAGE-SC-L exists only as a CBP software reference.

4.5.2 Trade-Off Discussion

The MLP could not be compiled for the target Cyclone V FPGA because it exceeded the FPGA's capabilities. It did not crash, but it compiled for hours without finishing, and it was forcibly terminated, which disqualifies it as a practical option. The SLP synthesized correctly with 16,858 ALMs.

Converting the math for inference of a neural network to RTL has the downside of generating a lot of multiplexers, one for each "`if (feature == FP_ONE)`". The count grows with parameters: 64 features per neuron in the SLP, or 8×128 weights in the MLP, driving up ALM usage. Traditional branch predictors like Bimodal avoid this by using a table indexed by the PC (program counter) to increment a counter, which maps better to an FPGA implementation.

The SLP still achieves the highest accuracy on both categories of patterns given sufficient data — that is, enough branches to learn the pattern (Section 4.9). However, the trade-off is understandable: the SLP runs at a third of Bimodal's clock frequency and uses 33% more ALMs, which makes Bimodal the overall winner.

4.6. Error Analysis

4.6.1 Correlation with Branch Entropy

Branch entropy quantifies the intrinsic predictability of a branch:

$$H = -p\log_2(p) - (1-p)\log_2(1-p) \tag{4-4}$$

where p is the taken probability. Fig. 4-3 plots MPKI against branch entropy for both predictors.

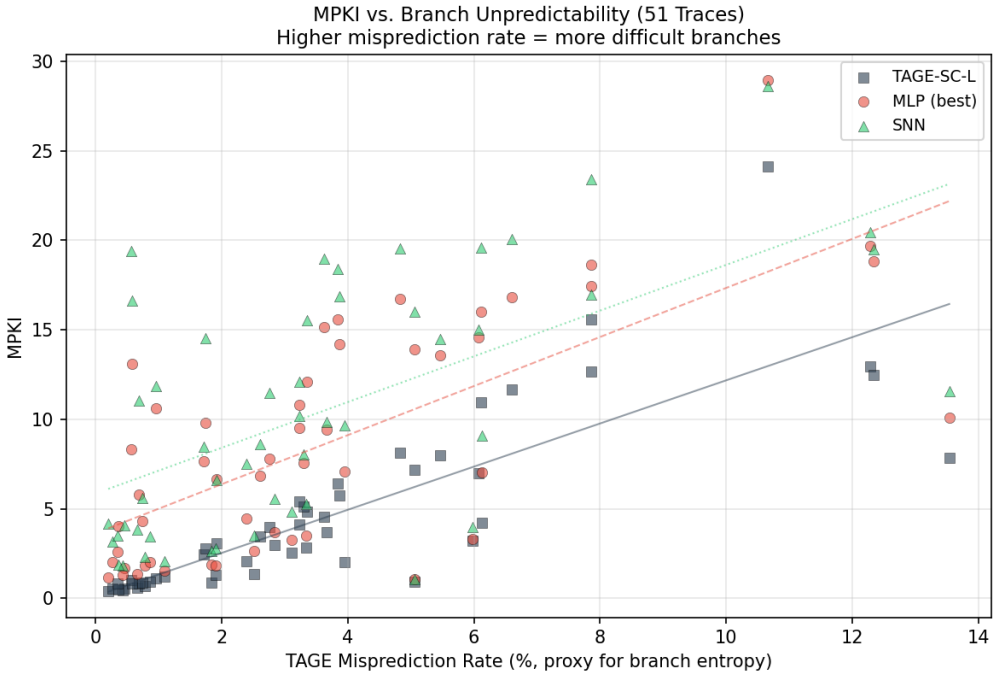

Fig. 4-3 MPKI vs. Branch Unpredictability (NN Predictor, 51 Traces). Misprediction rate serves as a proxy for branch entropy. INT workloads (red) cluster at higher unpredictability, while FP workloads (blue) tend toward more predictable behavior.

Both predictors degrade as entropy increases, but the NN predictor's MPKI rises more steeply. At low entropy (highly biased branches), both predictors perform comparably, as even simple heuristics suffice. At high entropy (branches near 50/50 taken probability), the gap widens because TAGE's multi-table architecture can exploit subtle long-range correlations that the NN's single feedforward pass may miss. This suggests that the NN predictor is best suited for workloads with moderate branch entropy, where its pattern-learning capability provides clear value without being overwhelmed by near-random branch behavior.

4.7. Position Relative to Prior Work

Table VII compares this work's CBP-run predictors against the prior literature. For consistency, MPKI is mispredictions per 1K instructions. Using MPKI for everything makes the comparison easier to read and easier to see the differences: a bigger number, a worse predictor.

The design journey began with the mention in Jiménez [Jiménez-05] of XOR patterns, which would require another model capable of handling non-linear patterns. The first instinct was to explore a Multi-Layer Perceptron and find more architectures to try with the CBP sim-

ulator and compare. The other neural-branch-prediction families surveyed at this stage were the spiking-style perceptron [Tarsa-15], CNN-based predictors [Zangeneh-20; Fang-23], and RNN/SRNN approaches [Zhang-20]. The biggest problem was that the complex math didn't translate well to hardware; multiplications and divisions already take up too much space compared to addition. And the activation functions needed at minimum an approximation that would make them lose a lot of information, or use a LUT. All those changes made them perform even worse. At the end, MLP was the only one scaling well with the precision reductions, thanks to a more hardware-friendly design; the only caveat was that it still used multiplication. The next step was implementing it in RTL; however, it was too complex to compile in a reasonable time, and Bimodal won by a large margin. Heuristically, the design was simplified until SLP emerged, and both architectures, MLP($8 \rightarrow 128 \rightarrow 1$) and SLP($64 \rightarrow 1$), were the best performers in the tests for RTL.

TABLE VII
THIS CASE STUDY'S CBP-RUN MPKI VALUES FOR THE PREDICTORS DEVELOPED HERE, AGAINST TAGE-SC-L AS THE ALGORITHMIC BASELINE.

Predictor	Type	MPKI (CBP run)
This Work — MLP (256→32→1 2-trace sample)	Multi-layer NN	1.231–2.002
This Work — MLP (256→32→1 51-trace suite)	Multi-layer NN	8.62 avg
This Work — SLP (128→1 51-trace suite)	Single-layer NN	10.50 avg
TAGE-SC-L (2016) [Seznec-16a]	Algorithmic	4.51 avg

TAGE-SC-L is our baseline, achieving the best accuracy (4.51 MPKI) across all 51 traces in the CBP simulator test. MLP misses almost double, with a score of 8.62 MPKI. SLP is the worst performer on the simulator tests, demonstrating that the tests are not stressing the same things, and a more sophisticated architecture does help the overall score of the predictor. On the two-trace sample subset, the MLP reaches a range of 1.231 (best variant) to 2.002 (worst variant) MPKI (Table IV), and TAGE gets a staggering 0.7 MPKI, or less than 1 miss per 1000 instructions. Re-running the simulator for the other NN branch predictor implementations will require hours of rework outside the scope of this work, especially after finding heuristically viable options that are actually implementable in RTL.

4.8. Discussion

4.8.1 When is the NN Predictor Superior?

Compared to the TAGE-SC-L branch predictor, a Neural Network option becomes a better choice when the area budget is below 0.5 mm^2, according to the simulation. The true

restraints appear once RTL implementation comes into play. Here, in the simulation, the NN performs well with acceptable accuracy, as the CBP traces are long enough for the NN to learn the patterns. In addition, its simplicity allows for multiple iterations, unlike TAGE's complex multi-table architecture. That being said, TAGE remains undefeated on raw accuracy, so the NN didn't put a dent in its already established implementations on more complex CPUs.

4.8.2 Hybrid Architectures

Hybrid architectures combining the MLP with TAGE through a confidence-based selector were explored during this case study. The selector collapsed to almost always choosing TAGE, and the MLP failed on the few branches it was given the chance to predict, so the hybrid performed worse than pure TAGE. The implementation was removed; alternative selector strategies (different confidence metrics, branch-type routing, or training-time selector co-training) remain open as future work.

4.8.3 Limitations and Threats to Validity

The CBP simulator is the standard, as it offers various capabilities that make it ideal for testing branch predictors. It can simulate and capture cache misses, cycle-accurate timing, IPC, and wrong-path cycles. The only limitation is that it has a fixed pattern of branch directions. Another reason that carries a lot of weight is that it is a simulator used by investigators for their papers, such as Seznec [Seznec-16a] and Zangeneh et al. [Zangeneh-20], so it is already battle-tested by prior investigations.

The estimations will use the old 65nm technology, but the comparison between implementations still holds up to discern which one uses less area, even when scaled to modern 7nm nodes.

Unfortunately, the current SLP implementation needs 1000 or more iterations to beat the Bimodal predictor. Section 4.9 reflects this conclusion. Its marginal accuracy depends entirely on the length of the program, which is typically thousands of loops and branches, and this is still a limitation. In essence, this is the warm-up time, or the number of iterations it takes to finally achieve a meaningful difference in accuracy.

4.9. Hardware Implementation Results: OOO Tomasulo Processor

The Tomasulo out-of-order design developed by the author is the basis for this investigation; it will provide a platform for running branch predictors in a more realistic test. Its design

is based on a RISC-V ISA (the base core's ISA subset, pipeline structure, and queue/buffer depths are documented in Appendix C), using Verilator as an open-source alternative to Questa and ModelSim, which were used as well. The two tests that were executed with this RTL design are normal branch patterns and XOR patterns.

4.9.1 Self-Contained Neural Network Architecture

The decision that sets apart both NN designs from a traditional branch predictor is that they don't use the Program Counter or Global History Table; the reason those inputs are declared anyway is to make the testbench easier to use when switching between branch predictors. But internally, those signals are not used; it only needs its own table of past branch results to learn the pattern.

```
module snn_branch_predictor #(
  parameter int NUM_FEATURES = 32,   // Past outcomes to track
  parameter int GHR_LEN = 32         // Unused - interface
   compatibility
)(
  // Global History input (UNUSED - tracks its own)
  input  logic [GHR_LEN-1:0] global_history,
  input  logic [63:0] pc,              // UNUSED
  input  logic        train_actual,    // The actual branch outcome
  // ...
);
  // Internal branch history - tracks actual outcomes
  logic [NUM_FEATURES-1:0] internal_history;

  // History update - shift in actual outcome on train_valid
  always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) internal_history <= '0;
    else if (train_valid)
      internal_history <= {internal_history[NUM_FEATURES-2:0],
   train_actual};
  end
```

Listing 4.1: Self-Contained SLP Design (from snn_branch_predictor.sv)

This self-contained design isolates the branch predictor from the processor, allowing both to be developed in parallel. However, synchronization still needs to be considered; the predictor must deliver the prediction when the processor needs it. This makes it perfect as a didactic design for students to develop both separately, without changing the processor design.

4.9.2 Test Suite Design

The normal pattern test (`BRANCH_PREDICTOR_TEST`) runs eight patterns: alternating taken/not-taken (50/50), two-taken-one-not-taken (67/33), 90% biased taken, a burst of 8 taken followed by 8 not-taken (50/50 overall), a nested 3-taken-1-not-taken loop (75/25), a Fibonacci-based pattern (∼67/33), always taken (100/0), and always not-taken (0/100). Each pattern runs for 1000 branches, so the suite covers 8000 branches in total.

Keeping the 50/50 bias as much as possible is important here, so the predictors don't just learn to always be taken or always not-taken and still achieve high accuracy. If the 50/50 bias is maintained, the branch predictor is forced to learn the pattern; otherwise, it only achieves 50-60% accuracy. This was observed in past tests where 70 or 80% of the branches were TAKEN, the branch predictors just predicted TAKEN, and the differences in accuracy were not as dramatic.

The XOR test is based on the concept introduced by Jiménez [Jiménez-05] to make an XOR pattern; unfortunately, any repetitive pattern is linear and can be learned by any branch predictor. It has 6 patterns, 1000 iterations each, and tries to maintain a 50/50 bias across the TAKEN and NOT TAKEN branches. The assembly source for both programs is in Appendix D, alongside a second normal-pattern variant used in the same runs.

4.9.3 Normal Pattern Results

Table VIII reports the accuracy of each predictor on the normal patterns.

TABLE VIII
NORMAL BRANCH PATTERN RESULTS (8000 BRANCHES / 1000 ITERATIONS)

Predictor	Correct	Mispredictions	Accuracy	Requires PC/GHR
SLP	**7898**	**102**	**98.72%**	**No**
Bimodal	7818	182	97.72%	Yes
MLP	7465	535	93.31%	**No**
Simple 2-bit	4987	3013	62.34%	Yes

SLP surpasses Bimodal, achieving 98.72% accuracy compared to 97.72% with Bimodal, provided at least 1000 iterations. Neural Network predictors need that number of iterations to learn the pattern, and with the same number of iterations, the SLP predictor surpasses the second-best, showing a larger ceiling. Unfortunately, the MLP predictor is too heavy and slow to learn the pattern, so its accuracy is lower, making it third place with an accuracy of 93.31%, and the 2-bit branch predictor seems to just get stuck on TAKEN, which is why it is close to 50%, with an accuracy of 62.34%.

4.9.4 Per-Pattern Breakdown: Normal Test

Table IX breaks the comparison down pattern by pattern between Bimodal and the SLP.

TABLE IX
PER-PATTERN ACCURACY: BIMODAL VS. SLP (NORMAL TEST / 1000 ITERATIONS)

Pattern	Branches	Bimodal	SLP	Winner
Alternating T/NT	1000	99.60%	**99.40%**	Bimodal
Two taken / one NT	1000	99.80%	**99.00%**	Bimodal
Biased 90% taken	1000	90.00%	**98.00%**	SLP
Burst pattern	1000	93.50%	**97.50%**	SLP
Nested loop	1000	99.70%	**98.90%**	Bimodal
Fibonacci-based	1000	99.80%	**97.90%**	Bimodal
Always taken	1000	100.00%	**99.70%**	Bimodal
Always not-taken	1000	99.40%	**99.40%**	Tie
Mean	**8000**	**97.72%**	**98.72%**	**SLP**

Bimodal still converges faster than SLP, as demonstrated by the simple patterns, which use fewer iterations, and Bimodal wins marginally on those patterns. Its traditional PC-indexed counters behave better in this scenario. SLP still has an edge on the more complex pattern tests, with 98.0% compared with the 90.0% of the Bimodal predictor on the biased-90% pattern, and 97.5% in contrast with 93.5% on the burst pattern. The perceptron's learning algorithm wins in these cases. The overall numbers are 98.72% for SLP and 97.72% for Bimodal.

Figs. 4-4 and 4-5 visualize the per-pattern accuracy comparison across all four predictors on the normal test suite.

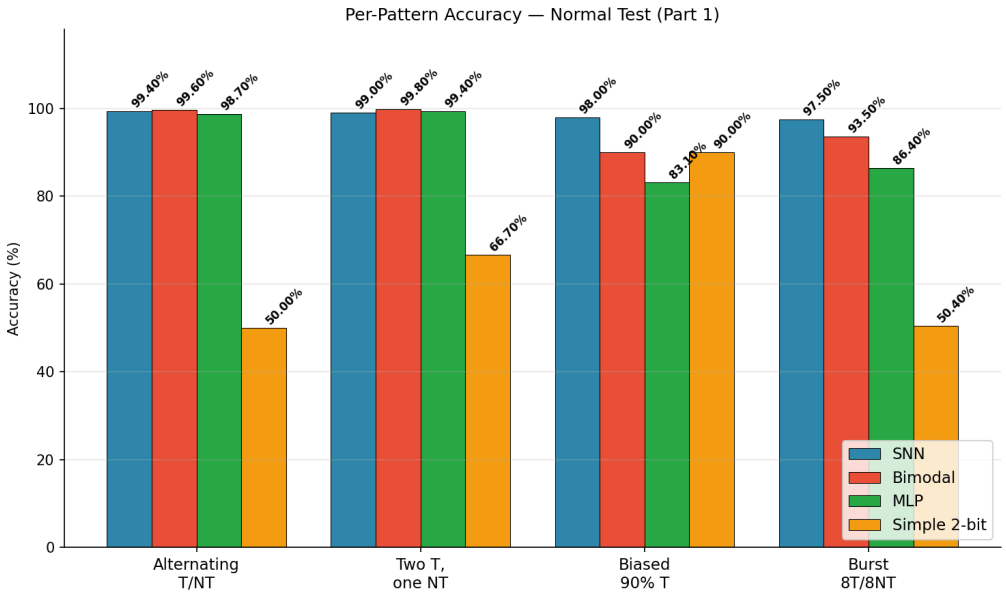

Fig. 4-4 Per-pattern accuracy comparison on normal branch patterns (Part 1): Alternating, Two-taken, Biased 90%, and Burst patterns. SLP dominates on patterns with non-trivial structure (Biased 90%, Burst), while Bimodal leads on simple repetitive patterns.

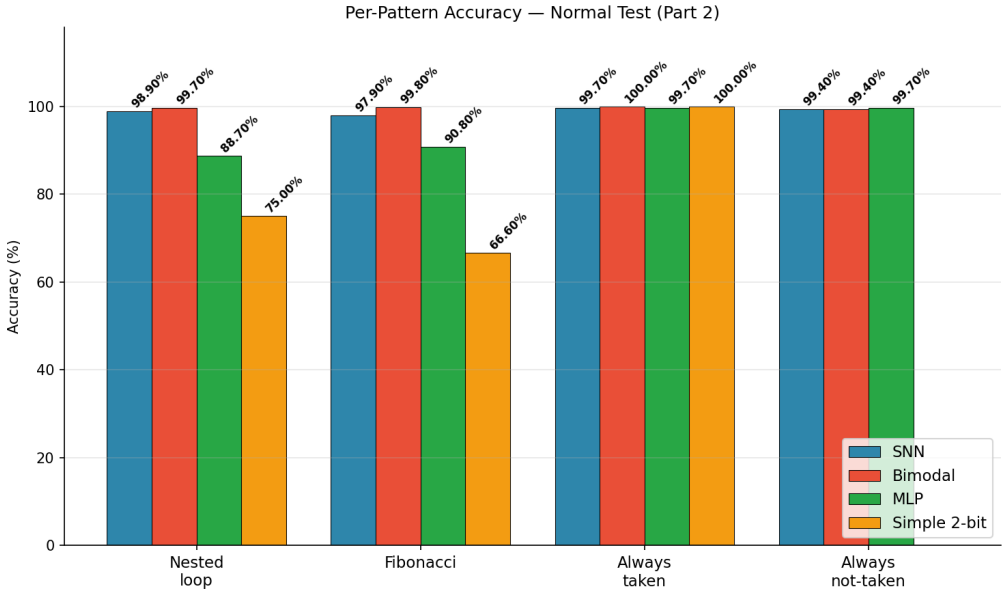

Fig. 4-5 Per-pattern accuracy comparison on normal branch patterns (Part 2): Nested loop, Fibonacci, Always taken, and Always not-taken patterns. Bimodal's PC-indexed counters excel on deterministic patterns, while SLP remains competitive across all categories.

4.9.5 XOR Pattern Results

The XOR test patterns translate the textbook XOR-defeats-perceptron concept [Jiménez-05] into ASM. That framing does not hold here: any periodic pattern with N iterations is learnable linearly with at least N history bits.

Table X presents results on the XOR pattern suite.

TABLE X
XOR PATTERN RESULTS (6000 BRANCHES / 1000 ITERATIONS)

Predictor	Branches	Correct	Accuracy	Requires PC/GHR
SLP	6000	**5893**	**98.22%**	**No**
Bimodal	6000	5708	95.13%	Yes
MLP	6000	5021	83.68%	**No**
Simple 2-bit	6000	3000	50.00%	Yes

Here we can see the problems MLP has, its complex architecture and low resolution on the input layer (only 8 input bits) make it converge slower, making it the third place overall of the predictors with a 83.68% as opposed to the 98.22% of its simpler sibling that has 64 input bits as the input layer, fitting comfortably the pattern periods (4 for 2-bit XOR and XNOR, 8 for 3-bit XOR, 16 for 4-bit Parity, 32 for Non-adjacent XOR). To reemphasize, the two NN models were made heuristically, so the MLP was shrunk to run the tests in an acceptable time, match the size of the other predictors more or less, and achieve good accuracy, and it is still third. 2-bit predictor got stuck on 1 decision, TAKEN, so it gets 50% right.

Figs. 4-6 and 4-7 show the per-pattern accuracy breakdown across all four predictors on the XOR test suite.

4.9.6 Summary: Pattern-Dependent Performance

TABLE XI
PREDICTOR EFFECTIVENESS BY PATTERN TYPE

Predictor	Normal	XOR	ALMs	Fmax (MHz)	Self-Contained
Bimodal	97.72%	95.13%	∼12 706	∼13.4	No
SLP	**98.72%**	**98.22%**	**∼16 858**	**∼4.5**	**Yes**
MLP	93.31%	83.68%	—	—	**Yes**
Simple 2-bit	62.34%	50.00%	∼12 587	∼13.6	No

Note: Synthesis results vary ±3% between Quartus runs due to non-deterministic place-and-route.

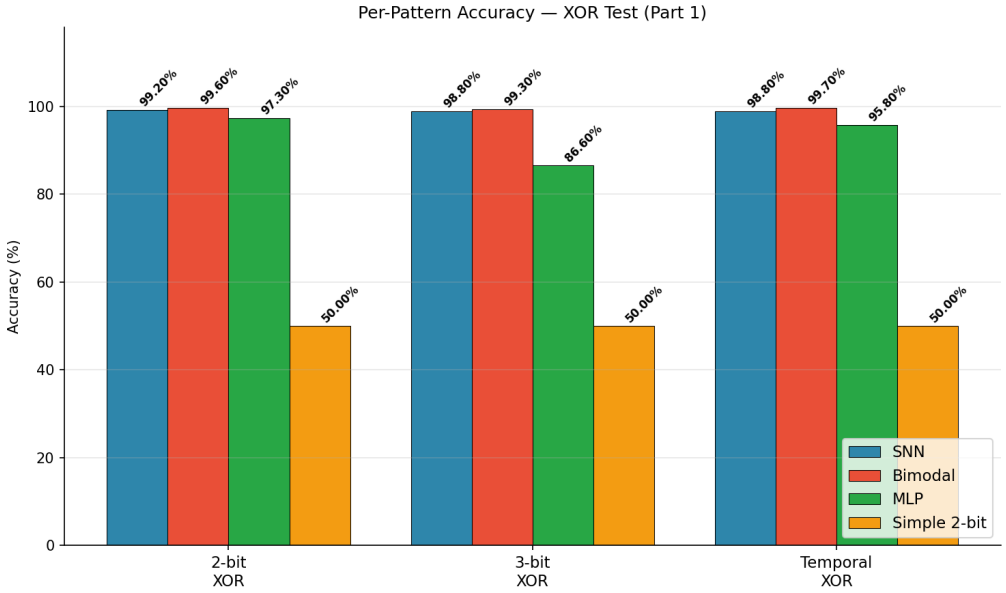

Fig. 4-6 Per-pattern accuracy comparison on XOR patterns (Part 1): 2-bit XOR, 3-bit XOR, and Temporal XOR. The SLP and Bimodal both achieve high accuracy on 2-bit and 3-bit XOR, while the simple 2-bit predictor collapses to 50% (random guessing).

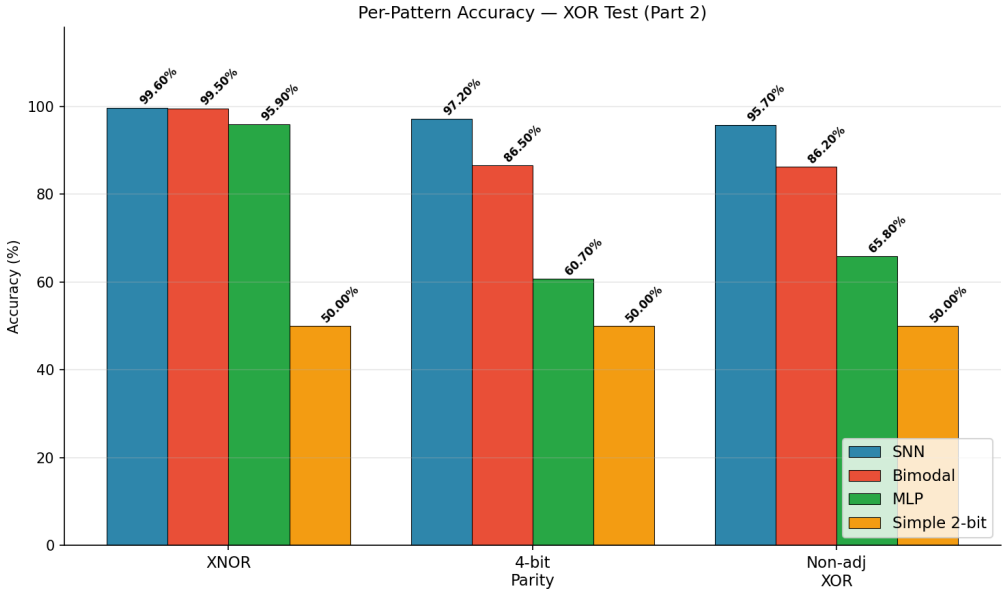

Fig. 4-7 Per-pattern accuracy comparison on XOR patterns (Part 2): XNOR, 4-bit Parity, and Non-adjacent XOR. The SLP maintains high accuracy across all patterns, while MLP degrades substantially on 4-bit Parity and Non-adjacent XOR due to its limited 8-bit history resolution.

The story repeats with SLP scoring higher than every other predictor. The SLP predictor continuously refines its weights to achieve greater accuracy over time, whereas Bimodal saturates and stops after a fixed number of iterations. Bimodal still has the advantage of being smaller (∼12K ALMs) and faster (∼13.5 MHz). If the priority is good accuracy and performance, Bimodal is a sweet spot compared to SLP, and MLP is overall not feasible in its current state.

4.9.7 Decision Boundary Visualization

To plot how the neural network predictors classify branches, the internal history is reduced to two dimensions using Principal Component Analysis (PCA). The SLP has a 64-bit internal history, and the MLP has an 8-bit one, so each prediction lives as a point in either a 64- or an 8-dimensional space. PCA finds the two orthogonal directions of largest variance and uses them as the axes PCA-1 and PCA-2. For a history vector $\mathbf{x} = [b_0, b_1, \ldots, b_{n-1}]$ with $b_i \in \{0, 1\}$:

$$\text{PCA-1} = \sum_{i=0}^{n-1} v_{1,i} \cdot b_i \quad \text{(weighted combination of history bits)} \tag{4-5}$$

$$\text{PCA-2} = \sum_{i=0}^{n-1} v_{2,i} \cdot b_i \quad \text{(different weighted combination)} \tag{4-6}$$

where $\mathbf{v}_1$ and $\mathbf{v}_2$ are the eigenvectors corresponding to the two largest eigenvalues of the data covariance matrix.

Fig. 4-8 shows the resulting decision boundaries from actual hardware execution.

The top row shows that the SLP learns linear decision boundaries (straight dashed lines) in PCA space, which is what a single-layer perceptron should produce. Even with the linearity constraint, the 64-bit history gives enough dimensions for the linear boundary to separate the classes on most XOR patterns. The bottom row shows the MLP's non-linear boundaries (curved lines) from its hidden layer. The MLP encounters a different problem: with only 8 history bits, many distinct branch sequences converge to the same point after the PCA projection. This is visible in patterns P5 (4-bit Parity) and P6 (Non-adj XOR), where the larger dots correspond to repeated overlapping samples. That overlap is the main reason the MLP loses to the SLP, even though its hidden layer is, in principle, more expressive than a single-layer linear classifier. The 2D PCA plot is also a projection that discards information; the underlying classifier uses all history bits, not just the two principal directions shown here.

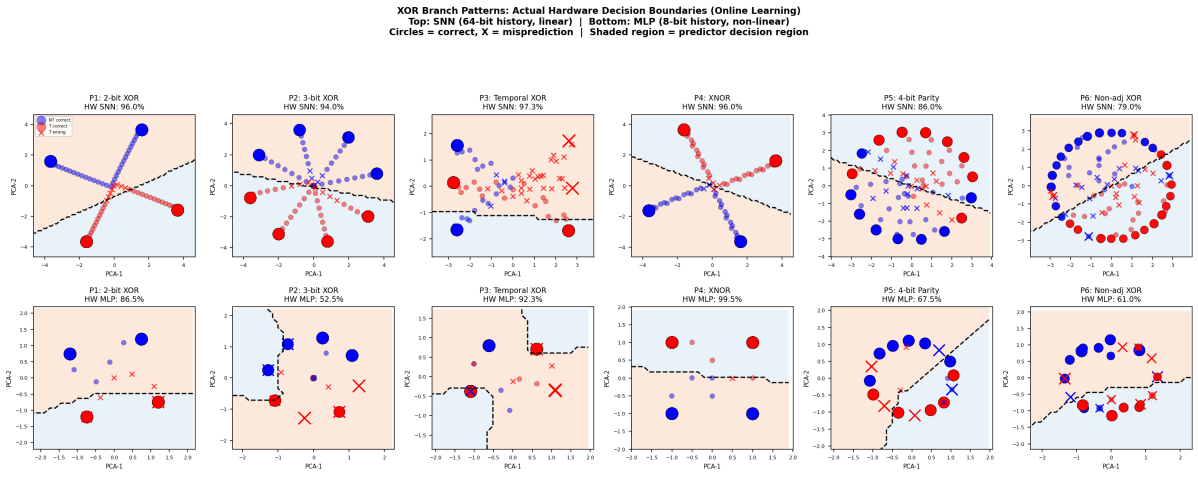

Fig. 4-8 Decision Boundary Visualization from actual hardware signals (runtime learning): Top row shows SLP (64-bit history, linear classifier), bottom row shows MLP (8-bit history, classifier with a non-linear hidden layer). Red points = taken branches, blue points = not-taken branches. Point size is proportional to the number of overlapping branches at each location, scaled with a tanh saturation curve to keep sizes bounded. Circles = correct predictions, X = mispredictions. Shaded regions indicate classifier decision regions.

4.9.8 Prediction Distribution Analysis

Fig. 4-9 plots the prediction distribution of the SLP, which produces balanced output instead of collapsing onto one class.

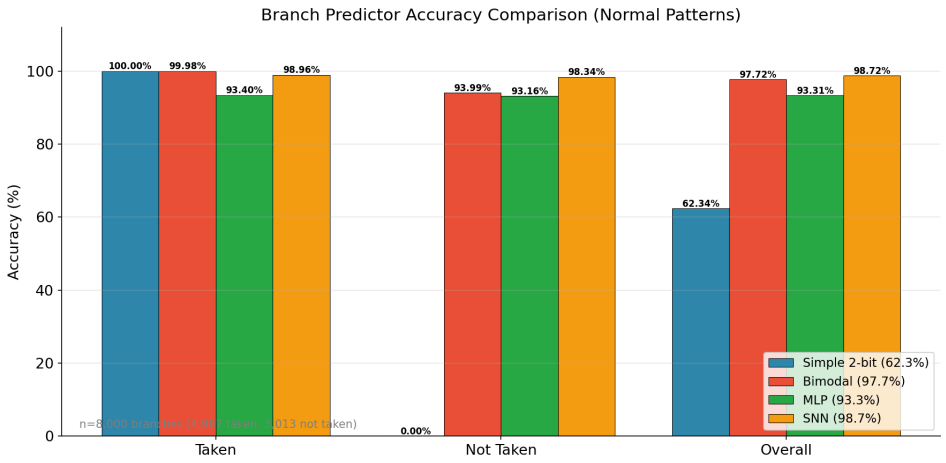

Fig. 4-9 SLP Branch Prediction Summary on Normal Patterns (Hardware Implementation)

The SLP produces 3013 not-taken and 4987 taken predictions, which match the actual distribution of 3013 not-taken and 4987 taken branches exactly. Its per-class accuracy is also balanced: 98.96% on taken branches and 98.34% on not-taken ones, so the learned weights are not biased toward either outcome. The MLP follows the same prediction distribution but with

lower per-class accuracy (93.40% taken, 93.16% not-taken), in line with its lower overall score.

Fig. 4-10 shows the same plot on the XOR suite. The SLP stays balanced and accurate on both taken and not-taken classes there as well.

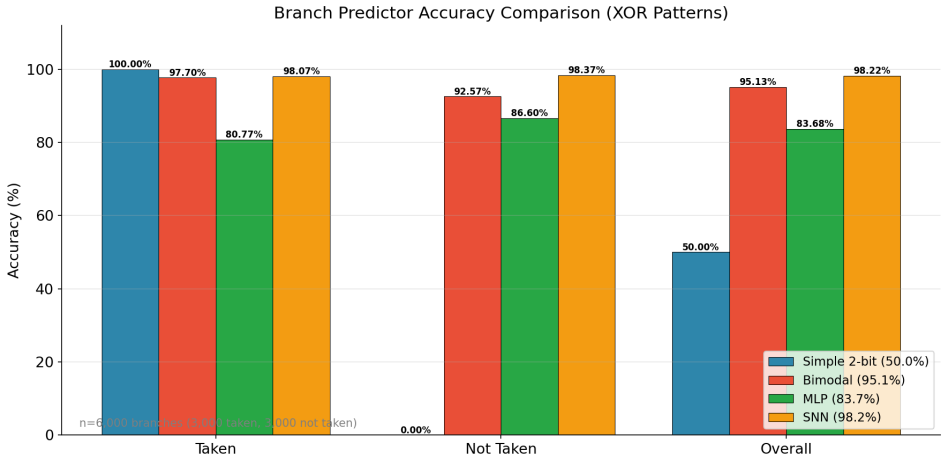

Fig. 4-10 Branch Predictor Accuracy Comparison on XOR Patterns (Hardware Implementation)

4.9.9 Learning Progression

Fig. 4-11 shows the SLP's accuracy rising over time during hardware execution on the normal patterns. With 1000 iterations per pattern, it settles at 98.72% overall, the ceiling that neural network predictors reach once they have enough branches to learn from.

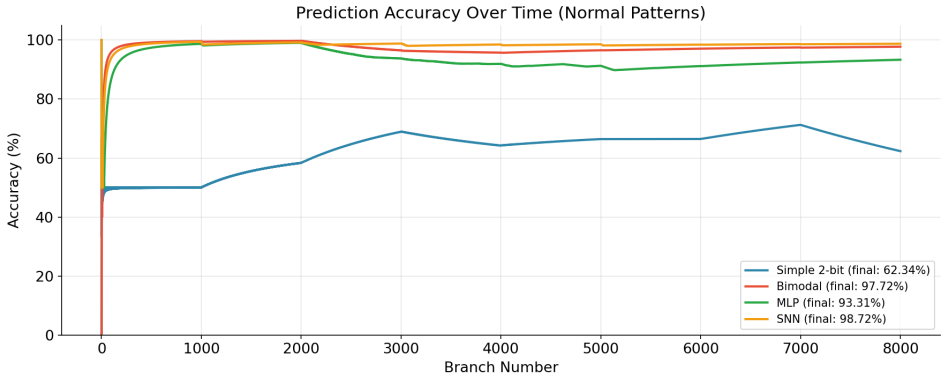

Fig. 4-11 SLP Accuracy Over Time on Normal Patterns (Hardware Implementation)

Fig. 4-12 is the same plot for the XOR suite, where the SLP rises along a similar curve to 98.22%.

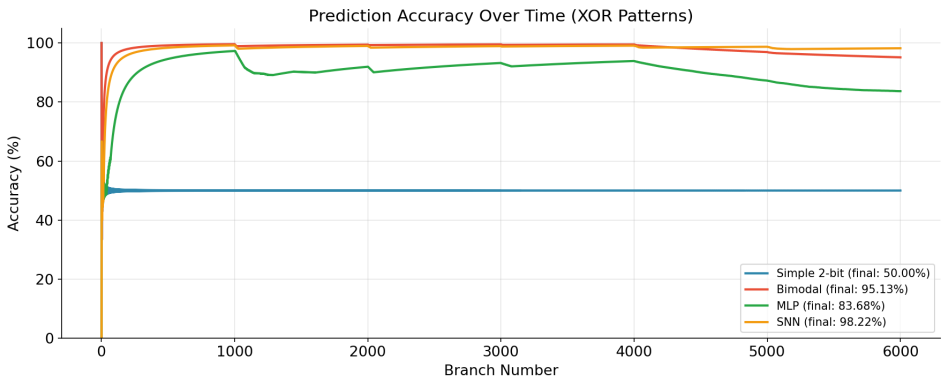

Fig. 4-12 Prediction Accuracy Over Time on XOR Patterns (Hardware Implementation)

4.9.10 Comparison: Trace Simulation vs. Hardware Implementation

Three conclusions emerge from comparing trace simulation (Table V, with the per-trace breakdown in Appendix B) against hardware execution (Table XI). First, neural network predictors can learn in actual hardware, not just in software simulation. Second, SLP outperforms MLP by having fewer layers, collapsing to a simple 1-perceptron layer, and converging faster. Third, it is possible to use only bit shifts, and the trade-off in accuracy doesn't negatively affect the prediction.

Conclusion and Future Work

This work demonstrates that it is possible to adapt the perceptron's mathematical model to be hardware-friendly by using only additions and shifts. This is the most shocking result: after losing a lot of precision, it can still predict the next branch outcome, thanks to the binary nature of the decision. Unfortunately, one of its best features, the lack of dependency on Program Counter indexing or an external Global History Register or Pattern History Table infrastructure, is also a disadvantage, as it forces the traditional architecture to change, and if the design already has all the infrastructure for a traditional branch predictor, it won't be plug-and-play with this particular RTL.

It is evident that simpler designs and neural network architectures benefit from this type of work, as they don't need more than two layers to start learning meaningful pattern recognition. The MLP implementation was a good prototype for the more efficient SLP, as it proved too slow to learn, too large to fit on the target FPGA, and overall worse than Bimodal by a large margin. A hybrid combining the MLP with TAGE via a confidence-based selector was also attempted during this work, but the selector collapsed, almost always choosing TAGE, and the MLP failed on the few branches it was actually given the chance to predict. The implementation was removed; alternative selector strategies remain open for future work.

The SLP could surpass the second-best branch predictor as long as it had enough history to learn the pattern. With a score of 98.72% on normal patterns and 98.22% on XOR patterns, winning over Bimodal by 1% and just over 3%, respectively. The disadvantages are area and maximum clock frequency: classical PC-indexed predictors such as Bimodal retain clear advantages in hardware efficiency, offering the smallest footprint (12,706 ALMs) and the fastest clock (13.44 MHz). The SLP represents a 33% increase in ALMs and, as a result, runs at almost a third of the clock frequency of Bimodal. In addition, Bimodal converges faster on short-lived workloads, where immediate PC-indexed tracking is valuable. TAGE-SC-L still has the best overall accuracy among all implementations at 4.51 MPKI, compared to the MLP's 8.62 and the SLP's 10.50. These NN branch predictors have a long way to go to replace TAGE entirely, let alone newer implementations. That said, the results do not conclusively demonstrate that SLP is the overall best implementation, as the tests are limited. Finally, the monolithic network architecture hurts overall performance and may not specialize in different branch types (loops, indirect branches, conditionals), potentially leaving accuracy on the table.

There are ways to follow up on this project. First, more cases are needed, or even a complete testbench based on the CBP simulator. Another is MLP optimizations; at present, its accuracy is diminished too much by removing the full multiplication, and addressing this would either make the MLP a worthwhile candidate or make it clear that it is not a path to

take. A different architecture for the MLP should be explored, since the input layer is too low-resolution to yield meaningful data from the branches. Specializing for INT and FP branches with a selector and two SLPs that are each experts on one type is one option, since the MLP variant already shows 22.0% higher MPKI on INT workloads than on FP (9.07 vs. 7.43; see Table V). However, the current work shows that the models learn at runtime, so an adaptive feature selection could address the observed workload sensitivity. More sophisticated runtime-learning strategies could improve both convergence behavior and hardware cost beyond the current fixed-learning-rate scheme, and lighter-weight update mechanisms could reduce the training-path gate count. Power was not measured here, so performance-per-watt remains a future-work axis rather than a finding of this case study.

Appendices

A. MLP IMPLEMENTATION DETAILS

This appendix reproduces the full C++ implementations referenced from Chapter 2: the FixedPoint28 arithmetic class and the piecewise-linear sigmoid that uses it.

A.1. Complete Fixed-Point Class

The complete implementation of the FixedPoint28 class is shown below:

```
class FixedPoint28 {
private:
    static constexpr int FRACTIONAL_BITS = 18;
    static constexpr int SCALE = 1 << FRACTIONAL_BITS; // 262,144
    static constexpr int MAX_VAL = 134217727;    // 2^27 - 1
    static constexpr int MIN_VAL = -134217728;  // -2^27

    int32_t value_;

public:
    FixedPoint28() : value_(0) {}

    explicit FixedPoint28(double d) {
        int64_t scaled = static_cast<int64_t>(d * SCALE + (d >= 0 ? 0.5
    : -0.5));
        value_ = static_cast<int32_t>(
            std::max(static_cast<int64_t>(MIN_VAL),
                    std::min(static_cast<int64_t>(MAX_VAL), scaled)));
    }

    double to_double() const {
        return static_cast<double>(value_) / SCALE;
    }

    FixedPoint28 operator+(const FixedPoint28& other) const {
        int64_t result = static_cast<int64_t>(value_) +
                         static_cast<int64_t>(other.value_);
        return FixedPoint28(static_cast<int32_t>(
            std::max(static_cast<int64_t>(MIN_VAL),
                    std::min(static_cast<int64_t>(MAX_VAL), result))));
    }

```

```
    FixedPoint28 operator*(const FixedPoint28& other) const {
        int64_t result = (static_cast<int64_t>(value_) *
                          static_cast<int64_t>(other.value_)) >>
   FRACTIONAL_BITS;
        return FixedPoint28(static_cast<int32_t>(
            std::max(static_cast<int64_t>(MIN_VAL),
                    std::min(static_cast<int64_t>(MAX_VAL), result))));
    }
};
```

Listing A.1: FixedPoint28 Fixed-Point Arithmetic Class

A.2. Piecewise Linear Sigmoid Implementation

```
FixedPoint28 sigmoid(const FixedPoint28& x) const {
    int32_t raw_val = x.raw();

    // Saturation limits (scaled to 28-bit fixed point)
    static constexpr int32_t SAT_HIGH = (4 << 18);   // 4.0
    static constexpr int32_t SAT_LOW = -(4 << 18);   // -4.0
    static constexpr int32_t HALF = (1 << 17);       // 0.5
    static constexpr int32_t ONE = (1 << 18);        // 1.0

    if (raw_val >= SAT_HIGH) return FixedPoint28(ONE);
    if (raw_val <= SAT_LOW) return FixedPoint28(0);

    // Linear approximation: sigmoid(x) = 0.5 + x/8
    // Divide by 8 using bit shift (>> 3) - multiplier-less!
    int32_t linear_part = raw_val >> 3;
    int32_t result = HALF + linear_part;

    // Clamp to [0, 1]
    if (result > ONE) result = ONE;
    if (result < 0) result = 0;

    return FixedPoint28(result);
}
```

Listing A.2: Hardware-Friendly Sigmoid Activation Function

A.3. Offline Weight Generator

The script that produces the initial weight hex files consumed by the SystemVerilog RTL on reset is reproduced below. It implements the same Galois LFSR formula used by the C++ reference, walks the LFSR state once for the MLP and once for the SLP, quantizes each draw to the Q10.18 fixed-point format described in Chapter 2, and writes one hex word per line into the files referenced by the `$readmemh` calls in Listing 3.7.

```
#!/usr/bin/env python3
"""
Generate weight initialization files for neural network branch predictors.

This script generates hex files that can be loaded by SystemVerilog using $readmemh.
The weights are initialized using the same LFSR formula as the original C++ reference.

Usage:
    python3 generate_weights.py [--output-dir DIR]

Output files:
    - mlp_weights_h1.hex    : Hidden layer weights (128 neurons × 8 features = 1024 values)
    - mlp_bias_h1.hex       : Hidden layer biases (128 values)
    - mlp_weights_out.hex   : Output layer weights (128 values)
    - mlp_bias_out.hex      : Output layer bias (1 value)
    - snn_weights.hex       : Single layer weights (128 values)
    - snn_bias.hex          : Single layer bias (1 value)
"""

import argparse
import os
from pathlib import Path


# Fixed-point parameters (Q10.18 format)
FRAC_BITS = 18
SCALE = 1 << FRAC_BITS  # 262144
MAX_VAL = (1 << 27) - 1   # 0x7FFFFFF
MIN_VAL = -(1 << 27)      # -0x8000000


def double_to_fixed(d: float) -> int:
    """Convert floating-point value to Q10.18 fixed-point integer."""
    scaled = int(d * SCALE + (0.5 if d >= 0 else -0.5))
```

```
    # Clamp to valid range
    if scaled > MAX_VAL:
        scaled = MAX_VAL
    if scaled < MIN_VAL:
        scaled = MIN_VAL
    return scaled


def fixed_to_hex(val: int, bits: int = 28) -> str:
    """Convert signed fixed-point integer to hex string."""
    if val < 0:
        # Two's complement for negative values
        val = (1 << bits) + val
    return f"{val:07X}"


class LFSR:
    """Galois LFSR matching the C++/SystemVerilog implementation."""

    def __init__(self, seed: int = 0xACE1):
        self.state = seed & 0xFFFFFFFF

    def next(self) -> int:
        """Generate next LFSR value and return it."""
        # Galois LFSR: seed = (seed >> 1) ^ (-(seed & 1) & 0xB400)
        lsb = self.state & 1
        self.state = (self.state >> 1) ^ (0xB400 if lsb else 0)
        return self.state

    def get_weight(self, scale: float = 0.1) -> float:
        """Generate a weight value in range [-scale, +scale)."""
        self.next()
        # Match C++ formula: ((seed & 0xFFFF) / 32768.0 - 1.0) * scale
        raw = (self.state & 0xFFFF) / 32768.0 - 1.0
        return raw * scale


def generate_mlp_weights(output_dir: Path,
                         num_features: int = 8,
                         hidden_neurons: int = 128,
                         seed: int = 0xACE1):
    """Generate weight files for MLP branch predictor (8→128→1)."""

    lfsr = LFSR(seed)

    # Hidden layer 1 weights: [hidden_neurons][num_features] flattened
   to 1D
```

```
    # Order: neuron 0 feature 0, neuron 0 feature 1, ..., neuron 1 feature 0, ...
    weights_h1 = []
    for h in range(hidden_neurons):
        for f in range(num_features):
            w = lfsr.get_weight(0.1)
            weights_h1.append(double_to_fixed(w))

    # Hidden layer 1 biases
    bias_h1 = [double_to_fixed(0.1)] * hidden_neurons

    # Output layer weights
    weights_out = []
    for h in range(hidden_neurons):
        w = lfsr.get_weight(0.1)
        weights_out.append(double_to_fixed(w))

    # Output bias
    bias_out = [double_to_fixed(0.0)]

    # Write files
    with open(output_dir / "mlp_weights_h1.hex", "w") as f:
        f.write(f"// MLP Hidden Layer Weights: {hidden_neurons} neurons × {num_features} features\n")
        f.write(f"// Format: Q10.18 fixed-point, 28 bits, hex\n")
        f.write(f"// Index = neuron * {num_features} + feature\n")
        for i, w in enumerate(weights_h1):
            f.write(f"{fixed_to_hex(w)}\n")

    with open(output_dir / "mlp_bias_h1.hex", "w") as f:
        f.write(f"// MLP Hidden Layer Biases: {hidden_neurons} values\n")
        f.write(f"// Format: Q10.18 fixed-point, 28 bits, hex\n")
        for b in bias_h1:
            f.write(f"{fixed_to_hex(b)}\n")

    with open(output_dir / "mlp_weights_out.hex", "w") as f:
        f.write(f"// MLP Output Layer Weights: {hidden_neurons} values\n")
        f.write(f"// Format: Q10.18 fixed-point, 28 bits, hex\n")
        for w in weights_out:
            f.write(f"{fixed_to_hex(w)}\n")

    with open(output_dir / "mlp_bias_out.hex", "w") as f:
        f.write("// MLP Output Bias: 1 value\n")
        f.write("// Format: Q10.18 fixed-point, 28 bits, hex\n")
        for b in bias_out:
```

```
            f.write(f"{fixed_to_hex(b)}\n")

    print(f"Generated MLP weights:")
    print(f"  - mlp_weights_h1.hex  ({len(weights_h1)} values)")
    print(f"  - mlp_bias_h1.hex     ({len(bias_h1)} values)")
    print(f"  - mlp_weights_out.hex ({len(weights_out)} values)")
    print(f"  - mlp_bias_out.hex    ({len(bias_out)} values)")

    return weights_h1, bias_h1, weights_out, bias_out


def generate_snn_weights(output_dir: Path,
                         num_features: int = 64,
                         seed: int = 0xACE1):
    """Generate weight files for SNN branch predictor (128→1)."""

    lfsr = LFSR(seed)

    # Single layer weights
    weights = []
    for f in range(num_features):
        w = lfsr.get_weight(0.1)
        weights.append(double_to_fixed(w))

    # Bias
    bias = [double_to_fixed(0.0)]

    # Write files
    with open(output_dir / "snn_weights.hex", "w") as f:
        f.write(f"// SNN Weights: {num_features} values\n")
        f.write(f"// Format: Q10.18 fixed-point, 28 bits, hex\n")
        for w in weights:
            f.write(f"{fixed_to_hex(w)}\n")

    with open(output_dir / "snn_bias.hex", "w") as f:
        f.write("// SNN Bias: 1 value\n")
        f.write("// Format: Q10.18 fixed-point, 28 bits, hex\n")
        for b in bias:
            f.write(f"{fixed_to_hex(b)}\n")

    print(f"Generated SNN weights:")
    print(f"  - snn_weights.hex ({len(weights)} values)")
    print(f"  - snn_bias.hex    ({len(bias)} values)")

    return weights, bias


```

```python
def generate_sigmoid_lut(output_dir: Path,
                         table_size: int = 256,
                         input_range: float = 8.0):
    """Generate sigmoid lookup table for MLP output activation.

    The sigmoid function: sig(x) = 1 / (1 + exp(-x))

    Input is quantized to table_size entries covering [-input_range, +input_range].
    Output is in Q10.18 fixed-point format, range [0, 1].

    To use: index = ((x >> INPUT_SHIFT) + OFFSET) & (TABLE_SIZE - 1)
    where INPUT_SHIFT and OFFSET are computed based on input_range and table_size.
    """
    import math

    sigmoid_values = []

    for i in range(table_size):
        # Map index to input value: [-input_range, +input_range]
        x = (i / (table_size - 1)) * 2 * input_range - input_range
        # Compute sigmoid
        sig = 1.0 / (1.0 + math.exp(-x))
        sigmoid_values.append(double_to_fixed(sig))

    # Write LUT file (no comments for $readmemh compatibility)
    with open(output_dir / "sigmoid_lut.hex", "w") as f:
        for val in sigmoid_values:
            f.write(f"{fixed_to_hex(val)}\n")

    # Also generate derivative LUT for backprop: sig'(x) = sig(x) * (1 - sig(x))
    sigmoid_deriv_values = []
    for i in range(table_size):
        x = (i / (table_size - 1)) * 2 * input_range - input_range
        sig = 1.0 / (1.0 + math.exp(-x))
        deriv = sig * (1.0 - sig)
        sigmoid_deriv_values.append(double_to_fixed(deriv))

    with open(output_dir / "sigmoid_deriv_lut.hex", "w") as f:
        for val in sigmoid_deriv_values:
            f.write(f"{fixed_to_hex(val)}\n")

    print(f"Generated sigmoid LUTs:")
    print(f"  - sigmoid_lut.hex       ({table_size} entries, range [-{input_range}, +{input_range}])")
```

```
    print(f"  - sigmoid_deriv_lut.hex ({table_size} entries, for backprop)")

    # Print useful constants for RTL
    input_shift = int(math.log2(SCALE * 2 * input_range / table_size))
    print(f"  RTL constants: INPUT_SHIFT={input_shift}, OFFSET={table_size // 2}, TABLE_SIZE={table_size}")

    return sigmoid_values, sigmoid_deriv_values


def verify_weights():
    """Print sample weights to verify the generation matches expected values."""
    print("\nVerification - First 5 MLP hidden weights:")
    lfsr = LFSR(0xACE1)
    for i in range(5):
        w = lfsr.get_weight(0.1)
        fixed = double_to_fixed(w)
        print(f"  [{i}] float={w:+.6f}, fixed={fixed:+8d}, hex={fixed_to_hex(fixed)}")


def main():
    parser = argparse.ArgumentParser(description="Generate neural network weight files")

    # Default output directory is proj_modelsim (where simulations run)
    script_dir = Path(__file__).parent.resolve()
    default_output = script_dir.parent.parent / "proj_modelsim"

    parser.add_argument("--output-dir", type=str, default=str(default_output),
                        help="Output directory for hex files (default: proj_modelsim)")
    parser.add_argument("--snn-features", type=int, default=64,
                        help="Number of SNN features/weights (default: 64)")
    parser.add_argument("--verify", action="store_true",
                        help="Print verification info")
    args = parser.parse_args()

    output_dir = Path(args.output_dir)
    output_dir.mkdir(parents=True, exist_ok=True)

    print("=" * 60)
    print("Neural Network Branch Predictor Weight Generator")
```

```
    print("=" * 60)
    print(f"Output directory: {output_dir.absolute()}")
    print(f"Fixed-point format: Q10.18 (28 bits total, 18 fractional)")
    print()

    generate_mlp_weights(output_dir)
    print()
    generate_snn_weights(output_dir, num_features=args.snn_features)
    print()
    generate_sigmoid_lut(output_dir)

    if args.verify:
        verify_weights()

    print()
    print("=" * 60)
    print("Done! Weight files generated successfully.")
    print("=" * 60)


if __name__ == "__main__":
    main()
```

Listing A.3: Weight File Generator (‘src/BRANCH_PREDICTOR/generate_weights.py’)

A.4. Hex File Format

All hex files loaded by the RTL via `$readmemh` share the same encoding. Each data line contains one 7-digit hex word representing a 28-bit Q10.18 fixed-point value, with negative numbers in two’s complement. Lines beginning with `//` are comments and are ignored by `$readmemh`. The file order matches the index order expected by the destination register array: for the MLP hidden weights, the index is neuron $\times$ num_features $+$ feature; for the sigmoid LUT, the index is $((\text{input} \gg 14) + 128)$ & 255, mapping the input range $[-8, +8]$ onto 256 sample points.

Listing A.4 reproduces an excerpt of ‘sigmoid_lut.hex’ (16 of its 256 data entries) to illustrate the format. The full file ships with the RTL at ‘src/BRANCH_PREDICTOR/sigmoid_lut.hex’.

```
// Sigmoid LUT: 256 entries
// Input range: [-8.0, +8.0]
// Output: Q10.18 fixed-point sigmoid values [0, 1]
// Index calculation: ((input >>> 14) + 128) & 255
0000058
000005E
```

```
0000064
000006A
0000071
0000078
0000080
0000088
...
003FF78
003FF80
003FF88
003FF8F
003FF96
003FF9C
003FFA2
003FFA8
```

Listing A.4: Excerpt from ‘sigmoid_lut.hex’ (8 leading + 8 trailing of 256 entries)

A.5. Sample Pre-Trained Weight File

To make concrete what `$readmemh` consumes at reset, an excerpt of ‘snn_weights.hex’ is reproduced below (first 32 of 128 entries followed by the last 8). The 128 hex words are the SLP single-neuron weights produced by Listing A.3 and follow the encoding documented in Section A.4. The complete file ships at ‘src/BRANCH_PREDICTOR/snn_weights.hex’. The MLP weight files follow the same pattern, scaled to their respective parameter counts (‘mlp_weights_h1.hex’: 1024 words for 128×8 hidden weights; ‘mlp_weights_out.hex’: 128 words for the output layer; the bias files hold one word per neuron).

```
// SNN Weights: 128 values
// Format: Q10.18 fixed-point, 28 bits, hex
0004EC0
FFFF42D
FFFC6E3
FFFB03E
FFFA4EC
00028DC
00057A1
000356A
FFFE782
FFFC08E
000237A
FFFDE8A
FFFBC12
FFFAAD6
```

```
0003237
00055E8
FFFF7C1
0000BE0
FFFD2BD
FFFB62B
FFFA7E2
00030BE
000552B
FFFF762
0000BB1
0005C3E
FFFFAEC
000070F
00059EE
0003CF6
FFFEB48
FFFC271
...
0006149
0003A3E
0004052
FFFECF6
0000014
0005CD6
FFFFB38
FFFCA69
```

Listing A.5: Excerpt from ‘snn_weights.hex’ (32 leading + 8 trailing of 128 entries, Q10.18 fixed-point)

B. COMPLETE RESULTS TABLES

This appendix contains the complete per-trace breakdown referenced in Chapter 4. The table that follows lists MPKI and accuracy for each of the 51 CBP traces evaluated.

B.1. Full 51-Trace Results

TABLE XII
PER-TRACE RESULTS ACROSS ALL 51 CBP TRACES (37 INT, 14 FP) FOR THE 'MLP_V8_CLEAN_LR_SHIFT_3_CONFIDENCE_HIST' PREDICTOR. MPKI IS THE 50%-WARMUP MISPREDICTION-PER-KILO-INSTRUCTION FIGURE; ACCURACY IS 100% − MR ON THE SAME WINDOW. SOURCE RUN: 'DATA/CBP2025_RUNS/RESULTS_V8_CONFIDENCE_HIST.CSV'; REBUILT BY 'SCRIPTS/AUDIT/BUILD_FULL_RESULTS_TABLE.PY'. THE ACTIVE SEED WEIGHTS ARE ARCHIVED ALONGSIDE THE RUN AS 'ROM_WEIGHTS_2026-04-03.H'; AN ALTERNATIVE 'ROM_WEIGHTS_V2.H' REGENERATED FROM 'MLP_WEIGHTS_FINAL_2026-05-11.BIN' VIA 'SCRIPTS/AUDIT/CONVERT_WEIGHTS_BIN_TO_ROM.PY' IS ALSO ARCHIVED IN THAT DIRECTORY, DEMONSTRATING THAT THE BIN-TO-HEADER CHAIN IS REPRODUCIBLE.

Workload	Trace	Size MB	MPKI	Accuracy
INT	int_0_trace	137.1	7.47	95.14%
INT	int_10_trace	204.9	1.63	98.60%
INT	int_11_trace	203.2	7.17	89.58%
INT	int_12_trace	204.6	2.23	98.47%
INT	int_13_trace	32.3	2.07	98.04%
INT	int_14_trace	177.3	1.11	99.42%
INT	int_15_trace	98.2	7.74	94.20%
INT	int_16_trace	128.3	16.66	90.10%
INT	int_17_trace	128.6	15.94	90.25%
INT	int_18_trace	108.3	2.66	98.88%
INT	int_19_trace	184.3	13.18	91.00%
INT	int_1_trace	92.8	16.80	90.46%
INT	int_20_trace	187.0	10.32	93.65%
INT	int_21_trace	91.6	29.71	86.81%
INT	int_22_trace	96.7	9.63	94.19%
INT	int_23_trace	182.4	6.78	94.92%
INT	int_24_trace	145.2	1.32	99.43%

Continued on next page

Table XII – *continued from previous page*

Workload	Trace	Size MB	MPKI	Accuracy
INT	int_25_trace	169.0	2.96	96.16%
INT	int_26_trace	197.2	2.63	95.07%
INT	int_27_trace	219.7	1.74	96.29%
INT	int_28_trace	185.9	9.25	90.72%
INT	int_29_trace	181.9	18.95	81.34%
INT	int_2_trace	94.1	16.03	91.04%
INT	int_30_trace	182.9	19.71	81.23%
INT	int_31_trace	118.1	5.65	95.21%
INT	int_32_trace	317.2	14.41	87.46%
INT	int_33_trace	310.4	13.94	90.05%
INT	int_34_trace	112.4	6.69	95.69%
INT	int_35_trace	119.9	7.78	94.53%
INT	int_36_trace	201.8	7.48	94.96%
INT	int_3_trace	96.9	6.00	94.67%
INT	int_4_trace	188.2	11.16	91.48%
INT	int_5_trace	138.5	3.46	96.00%
INT	int_6_trace	130.3	3.64	96.52%
INT	int_7_trace	128.2	14.29	90.48%
INT	int_8_trace	137.6	13.23	90.59%
INT	int_9_trace	152.6	12.27	91.52%
FP	fp_0_trace	64.7	3.41	93.67%
FP	fp_10_trace	270.8	4.35	95.03%
FP	fp_11_trace	297.8	1.84	97.93%
FP	fp_12_trace	48.5	1.44	98.71%
FP	fp_13_trace	84.6	18.70	88.43%
FP	fp_1_trace	42.7	17.39	91.20%
FP	fp_2_trace	114.0	3.36	95.91%
FP	fp_3_trace	223.0	7.08	86.21%
FP	fp_4_trace	106.3	1.15	98.66%
FP	fp_5_trace	332.9	1.06	94.26%
FP	fp_6_trace	63.6	8.35	95.25%
FP	fp_7_trace	159.1	10.61	91.14%
FP	fp_8_trace	211.1	13.96	79.28%
FP	fp_9_trace	143.5	15.19	87.85%

C. BASE OOO TOMASULO CORE SPECIFICATION

The branch-predictor variants evaluated in Chapter 4 run inside a custom out-of-order RISC-V core. The core uses Tomasulo's algorithm with register renaming and a reorder buffer, built along the canonical lines described in Patterson and Hennessy [Patterson-20, §4.11]. It is not meant to be a competitive general-purpose processor; it is just a synthesizable host pipeline that exposes a branch-prediction interface, so each predictor can be tested under realistic front-end pressure. The branch-predictor signal interface itself is documented in Listing 3.1.

Pipeline Structure

Fig. C-1 reproduces the host pipeline. The Branch Predictor supplies a speculative next PC into the I-fetch stage; the Reorder Buffer compares each retiring branch's actual outcome against its prediction and, on a mismatch, triggers the same flush path used by precise exceptions [Patterson-20, §4.11], which empties the front-end and the issue queues. The retire bus from the ROB commits results into the architectural register file in program order, and the CDB returns functional-unit results to both the ROB and any queue entry waiting on the same tag.

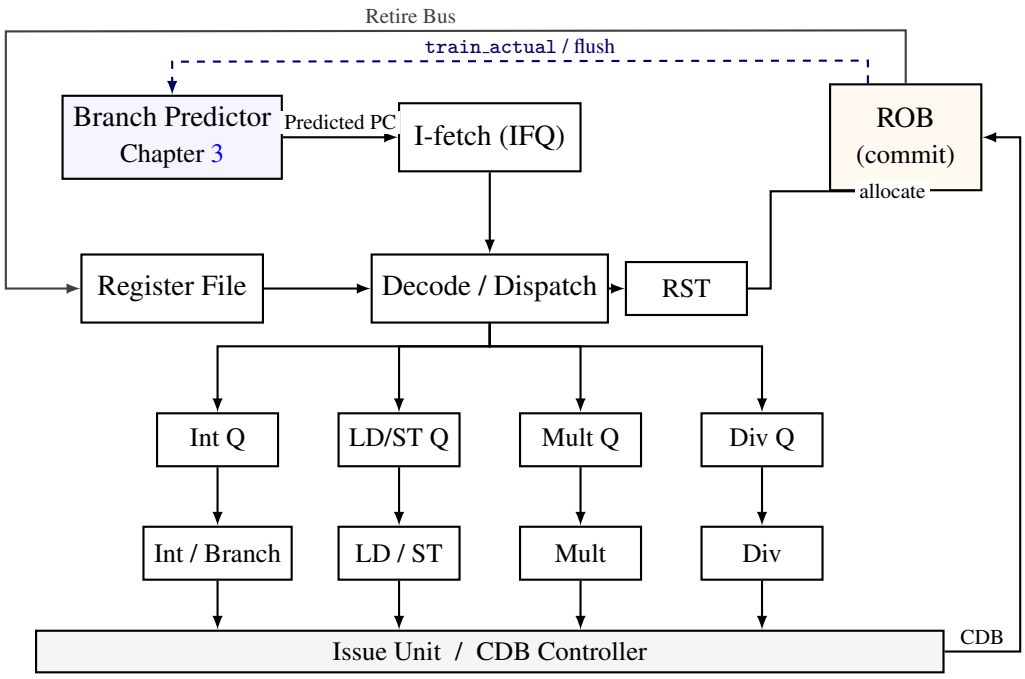

Fig. C-1 Base OOO Tomasulo host pipeline with the branch-predictor variants of Chapter 3 attached.

ISA Subset

The subset implemented here is a custom slice of RISC-V (Table XIII). It covers integer ALU, multiply/divide, load/store, and control-flow classes. Control flow uses the conditional branch instructions `BEQ` and `BNE`, as well as the `JAL` unconditional jump-and-link instruction. `SLT` is an ALU compare that puts a comparison result into a register, which the test programs then route into a `BEQ` or `BNE`. Only `BEQ` and `BNE` are routed to the branch-prediction interface. The ISA is intentionally minimal. It exists to exercise the ROB, reservation stations, and a common data bus, not to claim RISC-V conformance, and instruction encodings outside this subset are not implemented. The assembly test programs are reproduced in Appendix D; the `BEQ` and `BNE` branches in those listings are the conditionals the predictor encounters during simulation.

TABLE XIII
SUPPORTED INSTRUCTION SET SPANNING INTEGER ALU, MULTIPLY/DIVIDE, CONTROL-FLOW, AND LOAD/STORE CLASSES. `BEQ` AND `BNE` DRIVE THE BRANCH-PREDICTION INTERFACE; `SLT` IS AN ALU COMPARE AND IS NOT ROUTED TO THE PREDICTOR.

Mnemonic	Operation Class
ADD	Integer ALU
SUB	Integer ALU
XOR	Integer ALU
OR	Integer ALU
AND	Integer ALU
SLL	Integer ALU (shift left logical)
SRL	Integer ALU (shift right logical)
SLT	Integer ALU (set less than)
BEQ	Conditional branch (equal)
BNE	Conditional branch (not equal)
JAL	Unconditional jump and link
MUL	Multiplier
DIV	Divider
LD	Memory load
WR	Memory store

Structural Parameters

Buffer sizes throughout the pipeline are reported in Table XIV. The ROB order queue is 32 entries deep; the architectural register file holds 32 registers, each 32 bits; and the data memory is sized at 1024 words for the test programs in this case study.

TABLE XIV
PIPELINE BUFFER AND STORAGE SIZES FOR THE BASE OOO TOMASULO CORE. VALUES ARE READ FROM THE SYSTEMVERILOG PARAMETERS IN ‘SRC/’.

Structure	Capacity
Instruction Fetch Queue (IFQ)	4 entries
Program memory	64 lines (4 instructions per 128-bit line)
Integer reservation station	4 entries
Multiplier reservation station	4 entries
Divider reservation station	4 entries
Load/store queue	4 entries
ROB order queue (commit FIFO)	32 entries
ROB shadow register file	64 entries (6-bit tags)
Architectural register file	32 entries (32-bit)
Data memory	1024 words (32-bit)

D. TEST PROGRAM ASSEMBLY

The three programs reproduced here are the workloads run inside the OOO Tomasulo core in Section 4.9. They are written against the ISA subset of Appendix C (Table XIII) and assembled to hex by 'assembly_code/tools/assembly_2_hex.py'. Comments use the -- prefix. Register $24 is the harness's counted-branch flag, set to one immediately before each pattern's conditional and to zero immediately after, which is what the simulation accuracy counters scope on.

Normal branch patterns

The eight normal patterns characterized in Section 4.9 (alternating, two-taken/one-not-taken, biased-90%, burst, nested loop, Fibonacci, always-taken, always-not-taken):

```
-- Balanced Branch Predictor Test (using JAL for loop control)
-- Uses $24 as counting flag: set to 1 before pattern branch, 0 after
-- sim_main.cpp only counts branches/flushes when $24 == 1

-- Initialize
addi $1, $0, 0
addi $2, $0, 0
addi $3, $0, 0
addi $13, $0, 0      -- Pattern counter
addi $24, $0, 0      -- Counting flag

-- Constants
addi $25, $0, 1000   -- Iteration limit
addi $26, $0, 1      -- Constant 1
addi $27, $0, 2      -- Constant 2
addi $28, $0, 3      -- Constant 3

-- ============================================================
-- Pattern 1: Alternating T/NT
-- 50T/50NT expected
-- ============================================================
addi $13, $0, 1
addi $5, $0, 0

-- P1_LOOP: 10 instr
and $14, $5, $26     -- 1
addi $0, $0, 0       -- 2: NOP
addi $24, $0, 1      -- 3: FLAG ON
bne $14, $0, 8       -- 4: COUNTED
addi $0, $0, 0       -- 5: NOP
addi $24, $0, 0      -- 6: FLAG OFF
addi $0, $0, 0       -- 7: NOP
addi $5, $5, 1       -- 8
slt $12, $5, $25     -- 9
beq $12, $0, 8       -- 10
jal $0, -40          -- 10*4=40

-- ============================================================
-- Pattern 2: Two taken, one not-taken
-- 67T/33NT expected
-- ============================================================
addi $13, $0, 2
addi $5, $0, 0
addi $6, $0, 0

```

```
-- P2_LOOP: 12 instr
slt $14, $6, $27      -- 1
addi $24, $0, 1       -- 2: FLAG ON
bne $14, $0, 8        -- 3: COUNTED
addi $0, $0, 0        -- 4: NOP
addi $24, $0, 0       -- 5: FLAG OFF
addi $6, $6, 1        -- 6
slt $15, $6, $28      -- 7
bne $15, $0, 8        -- 8
addi $6, $0, 0        -- 9
addi $5, $5, 1        -- 10
slt $12, $5, $25      -- 11
beq $12, $0, 8        -- 12
jal $0, -48           -- 12*4=48

-- ============================================================
-- Pattern 3: Strongly biased taken (90%)
-- ============================================================
addi $13, $0, 3
addi $5, $0, 0
addi $6, $0, 0
addi $7, $0, 9
addi $8, $0, 10

-- P3_LOOP: 12 instr
slt $14, $6, $7       -- 1
addi $24, $0, 1       -- 2: FLAG ON
bne $14, $0, 8        -- 3: COUNTED
addi $0, $0, 0        -- 4: NOP
addi $24, $0, 0       -- 5: FLAG OFF
addi $6, $6, 1        -- 6
slt $15, $6, $8       -- 7
bne $15, $0, 8        -- 8
addi $6, $0, 0        -- 9
addi $5, $5, 1        -- 10
slt $12, $5, $25      -- 11
beq $12, $0, 8        -- 12
jal $0, -48           -- 12*4=48

-- ============================================================
-- Pattern 4: Burst pattern (8T, 8NT)
-- ============================================================
addi $13, $0, 4
addi $5, $0, 0
addi $6, $0, 0
addi $7, $0, 8
addi $8, $0, 16

-- P4_LOOP: 14 instr
slt $14, $6, $7       -- 1
addi $0, $0, 0        -- 2: NOP
addi $24, $0, 1       -- 3: FLAG ON
bne $14, $0, 8        -- 4: COUNTED
addi $0, $0, 0        -- 5: NOP
addi $24, $0, 0       -- 6: FLAG OFF
addi $0, $0, 0        -- 7: NOP
addi $6, $6, 1        -- 8
slt $15, $6, $8       -- 9
bne $15, $0, 8        -- 10
addi $6, $0, 0        -- 11
addi $5, $5, 1        -- 12
slt $12, $5, $25      -- 13
beq $12, $0, 8        -- 14
jal $0, -56           -- 14*4=56

-- ============================================================
-- Pattern 5: Nested loop (3T, 1NT)
-- ============================================================
addi $13, $0, 5
addi $5, $0, 0
addi $6, $0, 0
addi $7, $0, 3
addi $8, $0, 4

-- P5_LOOP: 12 instr
slt $14, $6, $7       -- 1
addi $24, $0, 1       -- 2: FLAG ON
bne $14, $0, 8        -- 3: COUNTED
addi $0, $0, 0        -- 4: NOP
addi $24, $0, 0       -- 5: FLAG OFF
```

```
addi $6, $6, 1       -- 6
slt $15, $6, $8      -- 7
bne $15, $0, 8       -- 8
addi $6, $0, 0       -- 9
addi $5, $5, 1       -- 10
slt $12, $5, $25     -- 11
beq $12, $0, 8       -- 12
jal $0, -48          -- 12*4=48

-- ============================================================
-- Pattern 6: Fibonacci-based
-- ============================================================
addi $13, $0, 6
addi $5, $0, 0
addi $6, $0, 1       -- Fib1
addi $7, $0, 1       -- Fib2

-- P6_LOOP: 11 instr
add $8, $6, $7       -- 1
and $14, $8, $26     -- 2
addi $24, $0, 1      -- 3: FLAG ON
bne $14, $0, 8       -- 4: COUNTED
addi $0, $0, 0       -- 5: NOP
addi $24, $0, 0      -- 6: FLAG OFF
add $6, $7, $0       -- 7
add $7, $8, $0       -- 8
addi $5, $5, 1       -- 9
slt $12, $5, $25     -- 10
beq $12, $0, 8       -- 11
jal $0, -44          -- 11*4=44

-- ============================================================
-- Pattern 7: Always taken
-- ============================================================
addi $13, $0, 7
addi $5, $0, 0

-- P7_LOOP: 10 instr
addi $14, $0, 1      -- 1
addi $0, $0, 0       -- 2: NOP
addi $24, $0, 1      -- 3: FLAG ON
bne $14, $0, 8       -- 4: COUNTED
addi $0, $0, 0       -- 5: NOP (never executed)
addi $24, $0, 0      -- 6: FLAG OFF
addi $0, $0, 0       -- 7: NOP
addi $5, $5, 1       -- 8
slt $12, $5, $25     -- 9
beq $12, $0, 8       -- 10
jal $0, -40          -- 10*4=40

-- ============================================================
-- Pattern 8: Always not-taken
-- ============================================================
addi $13, $0, 8
addi $5, $0, 0

-- P8_LOOP: 10 instr
addi $14, $0, 0      -- 1
addi $0, $0, 0       -- 2: NOP
addi $24, $0, 1      -- 3: FLAG ON
bne $14, $0, 8       -- 4: COUNTED
addi $0, $0, 0       -- 5: NOP
addi $24, $0, 0      -- 6: FLAG OFF
addi $0, $0, 0       -- 7: NOP
addi $5, $5, 1       -- 8
slt $12, $5, $25     -- 9
beq $12, $0, 8       -- 10
jal $0, -40          -- 10*4=40

-- ============================================================
-- End
-- ============================================================
addi $13, $0, 9
beq $0, $0, 0        -- Infinite loop
```

Additional normal patterns

A second normal-pattern program used alongside the first in the hardware test runs:

```
addi ra, zero, 1 # -- Initialize counter
addi sp, zero, 0 # -- Initialize taken counter
addi gp, zero, 0 # -- Initialize not-taken counter

add tp, ra, zero # -- Copy counter for pattern check
addi t0, zero, 2 # -- Divisor value
sub t1, tp, t0 # -- Test if counter >= 2
BNE4:
slt t2, t1, zero # -- Check if result is negative (counter < 2)
bne t2, zero, BNE1  # -- Branch 6 instructions ahead if counter < 2
sub tp, tp, t0 # -- Subtract 2 from counter
sub t1, tp, t0 # -- Test again
slt t2, t1, zero # -- Check if result is negative
bne t2, zero, BNE1 # -- Branch 2 instructions ahead if counter < 2
sub tp, tp, t0 # -- Subtract 2 again
beq tp, zero, BNE2 # -- If even (0), branch 2 ahead (taken pattern)
BNE1:
addi gp, gp, 1 # -- Increment not-taken counter
beq zero, zero, BNE3 # -- Unconditional branch to continue
BNE2:
addi sp, sp, 1 # -- Increment taken counter
BNE3:
addi ra, ra, 1 # -- Increment main counter
addi s0, zero, 20 # -- Loop limit
slt s1, ra, s0 # -- Check if we should continue
bne s1, zero, BNE4 # -- Branch back to start of loop

addi a0, zero, 0 # -- Reset counter for burst test
addi a1, zero, 0 # -- Burst pattern state
add a2, a0, zero # -- Copy counter
addi a3, zero, 4 # -- Cycle length (3 taken + 1 not taken)
sub a4, a2, a3 # -- Test modulo
slt a5, a4, zero # -- Check if negative
BNE6:
bne a5, zero, BNE5 # -- Branch if done with modulo
BNE9:
sub a2, a2, a3 # -- Continue modulo
beq zero, zero, BNE6 # -- Loop back for modulo
BNE5:
addi a6, zero, 3 # -- Threshold for taken branches
slt a7, a2, a6 # -- Check if position < 3
bne a7, zero, BNE7 # -- Branch if should be taken
addi gp, gp, 1 # -- Increment not-taken
beq zero, zero, BNE8 # -- Skip taken increment
BNE7:
addi sp, sp, 1 # -- Increment taken
BNE8:
addi a0, a0, 1 # -- Increment burst counter
addi s2, zero, 16 # -- Burst test limit
slt s3, a0, s2 # -- Check if should continue
bne s3, zero, BNE9 # -- Branch back to burst test

addi s4, zero, 0 # -- Reset for bimodal test
addi s5, zero, 10 # -- Half cycle length
slt s6, s4, s5 # -- Check if in first half
BNE12:
bne s6, zero, BNE10 # -- Branch if in taken phase
addi gp, gp, 1 # -- Increment not-taken
beq zero, zero, BNE11 # -- Skip taken increment
BNE10:
addi sp, sp, 1 # -- Increment taken
BNE11:
addi s4, s4, 1 # -- Increment bimodal counter
addi s7, zero, 20 # -- Bimodal test limit
slt s8, s4, s7 # -- Check if should continue
bne s8, zero, BNE12 # -- Branch back to bimodal test

addi s9, zero, 1 # -- Random seed LSB
addi s10, zero, 0 # -- Random counter
addi t4, zero, 1 # -- Mask for LSB
add s11, s9, zero # -- Copy seed
BNE15:
add s9, s11, zero # -- Update seed
```

```
and t3, s9, t4 # -- Get LSB for branch decision
bne t3, zero, BNE13 # -- Branch if LSB is 1
addi gp, gp, 1 # -- Increment not-taken
beq zero, zero, BNE14 # -- Skip taken increment
BNE13:
addi sp, sp, 1 # -- Increment taken
BNE14:
addi s10, s10, 1 # -- Increment random counter
addi t5, zero, 15 # -- Random test limit
slt t6, s10, t5 # -- Check if should continue
bne t6, zero, BNE15 # -- Branch back to random test

addi ra, zero, 0 # -- Final counter
beq ra, zero, BNE16 # -- Always taken branch
addi gp, gp, 1 # -- This should not execute
beq zero, zero, BNE17 # -- Skip
BNE16:
addi sp, sp, 1 # -- Increment taken counter
BNE17:

bne ra, zero, BNE18 # -- Always not-taken branch
addi sp, sp, 1 # -- This should not execute
beq zero, zero, BNE19 # -- Skip
BNE18:
addi gp, gp, 1 # -- Increment not-taken counter
BNE19:

beq zero, zero, BNE19 # -- Infinite loop to end simulation
```

XOR branch patterns

The six XOR patterns characterized in Section 4.9 (2-bit XOR, 3-bit XOR, temporal XOR, XNOR, 4-bit parity, non-adjacent XOR):

```
-- XOR Pattern Branch Predictor Test (using JAL for loop control)
-- Uses $24 as counting flag: set to 1 before XOR branch, 0 after
-- sim_main.cpp only counts branches/flushes when $24 == 1
-- This ensures only XOR test branches are counted, not loop control

-- Initialize
addi $1, $0, 0
addi $2, $0, 0
addi $3, $0, 0
addi $13, $0, 0     -- Pattern counter
addi $24, $0, 0     -- Counting flag (0 = don't count)

-- Constants
addi $25, $0, 1000  -- Iteration limit
addi $26, $0, 1     -- Constant 1
addi $27, $0, 2     -- Constant 2
addi $28, $0, 4     -- Constant 4
addi $29, $0, 8     -- Constant 8
addi $30, $0, 16    -- Constant 16

-- ============================================================
-- Pattern 1: 2-bit XOR (bit0 XOR bit1) -- 13 instr loop
-- ============================================================
addi $13, $0, 1
addi $5, $0, 0

-- P1_LOOP: 13 instr
and $14, $5, $26    -- 1: bit0
and $15, $5, $27    -- 2: bit1_raw
srl $15, $15, $26   -- 3: bit1
xor $16, $14, $15   -- 4: result
addi $0, $0, 0      -- 5: NOP
addi $24, $0, 1     -- 6: FLAG ON
beq $16, $0, 8      -- 7: XOR branch (COUNTED)
addi $0, $0, 0      -- 8: NOP
addi $24, $0, 0     -- 9: FLAG OFF
addi $0, $0, 0      -- 10: NOP
```

```
addi $5, $5, 1      -- 11
slt $12, $5, $25    -- 12
beq $12, $0, 8      -- 13
jal $0, -52         -- 13*4=52

-- ============================================================
-- Pattern 2: 3-bit XOR -- 16 instr loop
-- ============================================================
addi $13, $0, 2
addi $5, $0, 0

-- P2_LOOP: 16 instr
and $14, $5, $26    -- 1
and $15, $5, $27    -- 2
srl $15, $15, $26   -- 3
and $17, $5, $28    -- 4
srl $17, $17, $27   -- 5
xor $16, $14, $15   -- 6
xor $16, $16, $17   -- 7
addi $0, $0, 0      -- 8: NOP
addi $24, $0, 1     -- 9: FLAG ON
beq $16, $0, 8      -- 10: XOR branch
addi $0, $0, 0      -- 11: NOP
addi $24, $0, 0     -- 12: FLAG OFF
addi $0, $0, 0      -- 13: NOP
addi $5, $5, 1      -- 14
slt $12, $5, $25    -- 15
beq $12, $0, 8      -- 16
jal $0, -64         -- 16*4=64

-- ============================================================
-- Pattern 3: Temporal XOR -- 13 instr loop
-- Prev computed arithmetically: slt $18, $0, $16
--   $16=0 (taken) -> prev=0, $16=1 (not-taken) -> prev=1
-- ============================================================
addi $13, $0, 3
addi $5, $0, 0
addi $18, $0, 0     -- Previous outcome

-- P3_LOOP: 13 instr
and $14, $5, $26    -- 1: current_bit
xor $16, $14, $18   -- 2: current XOR previous
slt $18, $0, $16    -- 3: prev = ($16 != 0) ? 1 : 0
addi $0, $0, 0      -- 4: NOP
addi $24, $0, 1     -- 5: FLAG ON
beq $16, $0, 8      -- 6: XOR branch (COUNTED)
addi $0, $0, 0      -- 7: NOP
addi $24, $0, 0     -- 8: FLAG OFF
addi $0, $0, 0      -- 9: NOP
addi $5, $5, 1      -- 10
slt $12, $5, $25    -- 11
beq $12, $0, 8      -- 12
jal $0, -52         -- 13*4=52

-- ============================================================
-- Pattern 4: XNOR -- 14 instr loop
-- ============================================================
addi $13, $0, 4
addi $5, $0, 0

-- P4_LOOP: 14 instr
and $14, $5, $26    -- 1
and $15, $5, $27    -- 2
srl $15, $15, $26   -- 3
xor $16, $14, $15   -- 4
xor $16, $16, $26   -- 5: XNOR
addi $0, $0, 0      -- 6: NOP
addi $24, $0, 1     -- 7: FLAG ON
beq $16, $0, 8      -- 8: XNOR branch
addi $0, $0, 0      -- 9: NOP
addi $24, $0, 0     -- 10: FLAG OFF
addi $0, $0, 0      -- 11: NOP
addi $5, $5, 1      -- 12
slt $12, $5, $25    -- 13
beq $12, $0, 8      -- 14
jal $0, -56         -- 14*4=56

-- ============================================================
-- Pattern 5: 4-bit Parity -- 20 instr loop
-- ============================================================
```

```
addi $13, $0, 5
addi $5, $0, 0

-- P5_LOOP: 20 instr
and $14, $5, $26    -- 1
and $15, $5, $27    -- 2
srl $15, $15, $26   -- 3
and $17, $5, $28    -- 4
srl $17, $17, $27   -- 5
and $19, $5, $29    -- 6
addi $20, $0, 3     -- 7
srl $19, $19, $20   -- 8
xor $16, $14, $15   -- 9
xor $16, $16, $17   -- 10
xor $16, $16, $19   -- 11
addi $0, $0, 0      -- 12: NOP
addi $24, $0, 1     -- 13: FLAG ON
beq $16, $0, 8      -- 14: Parity branch
addi $0, $0, 0      -- 15: NOP
addi $24, $0, 0     -- 16: FLAG OFF
addi $0, $0, 0      -- 17: NOP
addi $5, $5, 1      -- 18
slt $12, $5, $25    -- 19
beq $12, $0, 8      -- 20
jal $0, -80         -- 20*4=80

-- ============================================================
-- Pattern 6: Non-adjacent XOR -- 17 instr loop
-- ============================================================
addi $13, $0, 6
addi $5, $0, 0

-- P6_LOOP: 17 instr
and $14, $5, $26    -- 1
and $15, $5, $28    -- 2
srl $15, $15, $27   -- 3
and $17, $5, $30    -- 4
addi $20, $0, 4     -- 5
srl $17, $17, $20   -- 6
xor $16, $14, $15   -- 7
xor $16, $16, $17   -- 8
addi $0, $0, 0      -- 9: NOP
addi $24, $0, 1     -- 10: FLAG ON
beq $16, $0, 8      -- 11: XOR branch
addi $0, $0, 0      -- 12: NOP
addi $24, $0, 0     -- 13: FLAG OFF
addi $0, $0, 0      -- 14: NOP
addi $5, $5, 1      -- 15
slt $12, $5, $25    -- 16
beq $12, $0, 8      -- 17
jal $0, -68         -- 17*4=68

-- ============================================================
-- End
-- ============================================================
addi $13, $0, 7
beq $0, $0, 0       -- Infinite loop
```

BIBLIOGRAPHY

[Courbariaux-15] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David, "Binaryconnect: Training deep neural networks with binary weights during propagations," *Advances in Neural Information Processing Systems*, vol. 28, 2015, Low-precision neural networks.

[Fang-23] Juan Fang, Jingming Guo, Yubin He, Min Cai, and Xiaobin Xu, "An attention-based cnn algorithm to predict hard-to-predict branches," in *2023 4th International Conference on Computer, Big Data and Artificial Intelligence (ICCBD+AI)*, Attention-based CNN for branch prediction, IEEE, 2023, pp. 116–121. DOI: 10.1109/ICCBD-AI62252.2023.00028

[Glorot-10] Xavier Glorot and Yoshua Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Xavier/Glorot initialization, 2010, pp. 249–256.

[Goldberg-89] David E Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[Han-16] Song Han, Xingyu Liu, Huizi Mao, et al., "Eie: Efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, Hardware-efficient neural network implementation, IEEE, 2016, pp. 243–254.

[Hennessy-17] John L Hennessy and David A Patterson, *Computer Architecture: A Quantitative Approach*, 6th. Morgan Kaufmann, 2017.

[Horowitz-14] Mark Horowitz, "Computing's energy problem (and what we can do about it)," *IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 10–14, 2014, Energy cost of computation analysis.

[IEEE-08] IEEE, *IEEE standard for floating-point arithmetic*, IEEE Std 754-2008, IEEE 754 floating-point standard, IEEE Microprocessor Standards Committee, IEEE Computer Society, 2008.

[Jiménez-01] Daniel A Jiménez and Calvin Lin, "Dynamic branch prediction with perceptrons," *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA-7)*, pp. 197–206, 2001, Original perceptron branch predictor paper.

[Jiménez-03] Daniel A Jiménez, "Fast path-based neural branch prediction," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36)*, IEEE, 2003, pp. 243–252.

[Jiménez-05] Daniel A. Jiménez, "Piecewise linear branch prediction," in *32nd International Symposium on Computer Architecture (ISCA'05)*, Introduces piecewise linear branch prediction; uses 2D XOR as the canonical linearly-inseparable example (Figure 1)., IEEE, 2005, pp. 382–393.

[Kessler-99] Richard E Kessler, "The alpha 21264 microprocessor," *IEEE Micro*, vol. 19, no. 2, pp. 24–36, 1999.

[Kingma-14] Diederik P Kingma and Jimmy Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014, Adam optimizer.

[LeCun-15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[LeCun-98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998, Classic CNN paper.

[Martínez-24] Braulio Martínez, *Branch prediction*, Course handout, ITESO, 2024.

[McFarling-93] Scott McFarling, "Combining branch predictors," in *TN-36, Digital Western Research Laboratory*, Introduced hybrid predictors, 1993.

[Patterson-20] David A Patterson and John L Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 2nd. Morgan Kaufmann, 2020, ISBN: 9780128203316.

[Seznec-06] André Seznec and Pierre Michaud, "A case for (partially) tagged geometric history length branch prediction," in *Journal of Instruction Level Parallelism*, Original TAGE predictor paper, vol. 8, 2006, pp. 1–23.

[Seznec-16a] André Seznec, "Tage-sc-l branch predictors again," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, CBP 2016 Championship Winner, 2016.

[Seznec-16b] André Seznec and Daniel A Jiménez, "The 5th championship branch prediction competition (cbp-5)," JILP Workshop on Computer Architecture Competitions, Tech. Rep., 2016, CBP 2016 competition specifications.

[Smith-95] James E Smith and Gurindar S Sohi, "The microarchitecture of superscalar processors," *Proceedings of the IEEE*, vol. 83, no. 12, pp. 1609–1624, 1995.

[Tarsa-15] Stephen J Tarsa and Louis-Noël Pouchet, "Spiking neural network branch prediction," in *Proceedings of the International Conference on Computational Science*, Elsevier, 2015, pp. 418–427.

[Weste-10] Neil H. E. Weste and David M. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th. Addison-Wesley, 2010, Chapter 11 "Datapath Subsystems"; multiplier gate/area data in Table 11.16 (54x54-bit multipliers)., ISBN: 978-0-321-54774-3.

[Yeh-91] Tse-Yu Yeh and Yale N Patt, "Two-level adaptive training branch prediction," in *Proceedings of the 24th Annual International Symposium on Microarchitecture (MICRO-24)*, IEEE/ACM, 1991, pp. 51–61.

[Zangeneh-20] Siavash Zangeneh, Stephen Pruett, Sangkug Lym, and Yale N. Patt, "Branchnet: A convolutional neural network to predict hard-to-predict branches," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, CNN branch predictor for hard-to-predict branches, IEEE, 2020, pp. 118–130. DOI: 10.1109/MICRO50266.2020.00022

[Zhang-20] Lei Zhang, Ning Wu, Fen Ge, Fang Zhou, and Mahammad Rehan Yahya, "A dynamic branch predictor based on parallel structure of srnn," *IEEE Access*, vol. 8, pp. 86 884–86 894, 2020, Sliced RNN branch predictor with parallel structure. DOI: 10.1109/ACCESS.2020.2992643

Index