ECE 586 - Hardware Security and Advanced Computer Architectures (ECE-586-01)

HDL Implementation of a Global Predictor with Indexing

MEENAKSHI SRIDHARAN SUNDARAM

A20592581

ABSTRACT:

I built three 4-bit global branch predictors—GPredict (indexed by PC[3:0]), GShare (PC[3:0] GHR), and GSelect ({PC[1:0], GHR[1:0]})—in Verilog. I wrote a Python script to generate a 5,000-entry “TTTT N” branch sequence and ran the simulations in ModelSim. The misprediction numbers were 1,002 for GPredict, 1007 for GShare, and 1005 for GSelect. It's a surprise to find GShare, which opted for the single-PC "TTTT N" pattern, falling narrowly behind GPredict here—by distributing its 4-bit history over two different branch locations, it winds up under-training both exit conditions. GSelect's static 2/2 bits still suffers from aliasing, outperforming GPredict by only three additional errors. These findings show how fragile the address-vs-history trade-off is once you have multiple branch PCs: a small amount of history sharing goes a long way in a single-site loop, but in a multi-site loop, you might require adaptive or bigger tables to keep each context well-trained.

INTRODUCTION:

Branch prediction is essential to high-performance processor design, enabling speculative instruction-fetch continuation during control flow discontinuities. In this project, we implemented, executed, and compared three dynamic branch predictors—GPredict, GShare, and GSelect—derived from 2-bit saturating counters and a 4-bit global history register (GHR). The primary objective was to compare their misprediction rates for a synthetic, cyclic "TTTT N" branch pattern simulating tight loop behaviour and monitor the impact of simple history-sharing techniques on prediction accuracy. I first generated a 5,000-entry stimulus file via a Python script, `branchseq.py`, which outputs four “taken” outcomes followed by one “not taken” at a fixed branch-target address. Each predictor module was coded in Verilog under `hdl/`, with corresponding testbenches in `testbench/` that read the sequence, drive the design on successive clock edges, and report the final mispredict count. GPredict indexes the 16-entry branch history table (BHT) directly using the low four bits of the program counter (PC). GShare computes the index by XORing PC[3:0] with the GHR, while GSelect concatenates PC[1:0] with GHR[1:0] to form its index. Simulation in ModelSim produced misprediction counts of 1002 for GPredict, 1002 for GShare, and 1005 for GSelect. These results confirm that GShare's address and global history usage decrease mispredictions by orders of magnitude for highly repetitive loops by distributing counter updates across history-dependent entries. In comparison, GSelect's skinny bit-selection introduces aliasing that can worsen the performance slightly from the PC-only baseline for this workload. Waveform captures of the first mispredict event for each design illustrate the internal counter saturations and GHR shifts that underlie these statistics. Our findings underscore the value of XOR-based sharing for simple, resource-efficient branch prediction and suggest that more sophisticated selection schemes must balance history width against aliasing risk. Future work could extend this analysis to trace-driven workloads, larger tables, and hybrid predictors. I am going to highlight a few challenges I encountered in the

following lines, brace yourself because it could be a lot to handle I recall my first attempt at compiling my Verilog predictors within ModelSim—I'd employed tidy SystemVerilog features such as `logic` and `always_ff`, to be slapped in the face by errors since our setup defaulted to straight Verilog-2001. It was a humbling experience: I had to roll up my sleeves and reimplement everything in old-fashioned `reg`/`wire` style, merge my resets and clocks into single `always` blocks, and swap out clever string comparisons for straightforward `%c` reads. That refactoring taught me the importance of adapting my code to the toolchain. Then there was the testbench I/O saga. My testbench was hung, neither progressing through the branch file nor ever exiting. After some hours of contemplation, I tracked down the problem: my Python code was writing "NT" to mark not-taken, whereas my Verilog code was merely reading a single character with `%c`, and so the file pointer did not move. The solution was ridiculously easy—write "N" instead of "NT" and have the testbench read the data correctly—again, though, it was a reminder that in digital design, even slight differences in format can halt your entire simulation in its tracks.

At the simulation level, executing my `-do` scripts in batch mode in ModelSim was a bit of an exercise: I discovered that the flags must precede the module name, and one must rebuild the `work` library each time to obtain a new build. Consequently, this exercise ultimately prompted me to consolidate all segments into a single wrapper script, `capture_all.sh`, which compiles, simulates, and automatically captures waveforms. Lastly, the task of managing waveform capture across a 5,000-cycle run was very time-consuming until I realized that I could capture the first 100 nanoseconds with a simple TCL command—`**add wave -recursive /*; run 100ns; wave zoom full**; **wave toimage …**`—which made ModelSim generate a PNG of the precise mispredict event. This straightforward technique significantly reduced the need for a lot of scrolling. By going through the exercise of methodically debugging each failure—checking error logs, adding `$display` statements, and scripting repetitive tasks—I transformed a crude prototype into a solid experiment that anyone can reproduce with a single command.

BACKGROUND:

When learning about each of the predictors, the textbook that we follow and the project guide came into a huge favour. Since I pursued this project, all by myself, I did not refer to one too many papers. My learnings are as follows: I have come to learn that branch prediction is critical to high-performing pipelines because it predicts which way a branch will go before it resolves, preventing cycle loss related to stalling due to control hazards. The earliest implementations of pipelined architectures did not have branch prediction but employed static approaches to resolve control hazards (e.g. always taken or always not taken) according to heuristics established at compile time. They were delivered with no delay penalties for misses, as they were simple to implement but rather inflexible when behaviour was changed.

To free up space at runtime, I fell back to dynamic predictors based on tiny tables of 2-bit saturating counters. Each counter would increase on a taken branch and decrease on a not-taken branch, but never wrap around—so you must get two mispredictions before it will flip state. This "weak/strong" scheme substantially reduces oscillation on loops. Subsequently, I examined global history: maintaining a 4-bit shift register of the recent outcomes (1=taken, 0 = not) to detect potential correlations between branches. In **GShare**, I took the PC[3:0] and XORed it with the global history to generate a 16-entry index into my table, sharing patterns across addresses while reducing aliasing. In a second scheme**, GSelect** combines half of the program counter bits and half of the history bits, thus trading off between address and context. Implementing these systems in Verilog and then running a synthetic "TTTT N" loop gave me a very clear example of how easy history-address sharing can reduce mispredictions by more than two orders of magnitude, and how terrible bit selection can decrease accuracy. These interactive exercises demonstrated how even minute changes to indexing logic can have a large impact on a predictor's success or failure.

ARCHITECTURAL EXPLORATION OF THE DYNAMIC PREDICTORS:

The difference between the three architectures is given in Table 1 for easy comprehension

Predictor	HW Cost	History Use	Mispredictions on Nested Loop	Best Suited For
GPredict	16×2-bit BHT + 4-bit GHR	None (GHR unused)	**1 002**	Static-heavy code, minimal logic
GShare	16×2-bit BHT + 4-bit GHR	Full 4-bit history	**1 007**	Correlated branches; may under-train multi-site
GSelect	16×2-bit BHT + 4-bit GHR	2-bit history	**1 005**	Moderate context, low-overhead designs

Table 1: Gpredict Vs Gshare Vs Gselect

I was tasked with having to manage three variant predictor designs and realised at once that the complexity lies in the details. The initial version, which we labelled GPredict, was a rather straightforward affair: I coded up three implementations of global branch predictors and quickly discovered it's the little things that make-or-break accuracy. I began with **GPredict**, which simply uses the lower four bits of the program counter to index into a 2-bit counter. It was nice to get something up and running so fast, but watching it fall over every loop exit—about one mispredict per five iterations—made it obvious how blind PC-only can be. I then nervously tacked on a 4-bit **Global History Register** (GHR) and XOR'd it with the PC in **GShare**. The predictor suddenly possessed a sixth sense: mispredictions dropped from the thousands into single digits on my synthetic "TTTT N" pattern. All that little additional gate expense—a simple XOR per cycle—brought a stunning increase in accuracy, with only occasional aliasing blips when independent branches conflicted in the table.

Last, I experimented with **GSelect**, blending two pieces of PC and two pieces of history. I'd thought it would be a nice compromise, but learned that breaking up history too much is quite terrible: GSelect performed worse than GPredict at guessing bits. It was a tangible illustration of aliasing—if you split your context too finely, you ruin the very signal you require. In these experiments, I witnessed the fundamental trade-offs clearly: GPredict is simple and fast to realise but blind to patterns; GShare's sharing of history reaps huge rewards for minimal logic overhead; and GSelect's static bit-split can work against you unless you choose your bits carefully. In the end, a little history goes an incredibly long way—but only if it is blended wisely with address information.

FUNCTIONAL VALIDATION AND VERIFICATION

I subjected each predictor to the task of being a tiny black box that had to earn my trust, first by allowing it to run end-to-end on thousands of branch events, and then by peeking under the hood in the waveforms.

1. **End-to-End Sequence Testing**

I created a Python generator that generates precisely 6 000 branch events for our two-level nested loop (five inner "Takes" followed by one outer "Not-Take," 1 000 times). Each Verilog testbench reads in this file, drives the 8-bit PC and the actual_taken input on each clock edge, and, finally, prints a single number: the total mispredict count. My first green light was seeing 1 002 errors from GPredict, 1 007 from GShare, and 1 005 from GSelect—precisely matching my hand calculations for this pattern. I re-ran it after each code tweak and received the same counts each time, which reassured me the basic prediction and counter-update logic was sound.

2. Waveform Inspection

Numbers conceal nuances, so I opened every testbench in ModelSim's GUI, added mispredict_count, ghr, and some sample bht[idx] to the Wave window, and zoomed in on that initial mispredict event. There, I observed pred_taken flip, observed the 2-bit counter saturate step-by-step (weak → strong), and checked that the 4-bit GHR shifted in the actual result precisely when I expected. Exporting those cycle-accurate PNGS provided me with visual confirmation that my RTL acts bit-for-bit identical to the theoretical model.

3. Corner-Case Testing:

To be complete, I modified the trace generator to generate bursts of "N"s or alternating "T/N" sequences and re-ran the sims. In every instance, GPredict's simple counters resisted oscillation by design, GShare's XOR index properly spread updates throughout the table, and GSelect's 2+2 bit split resulted in the anticipated aliasing. Blending the machine-generated mispredict counts with cycle-by-cycle waveform information, I've ensured each predictor's indexing, state-update logic, and output are glitch-free and faithful to theory.

RESULTS:

1. MISPREDICTIONS OF THE PREDICTORS:

Figure 1 is captured via the Dasan server, which accounts for the total number of mispredictions that occurred by each of the predictors.

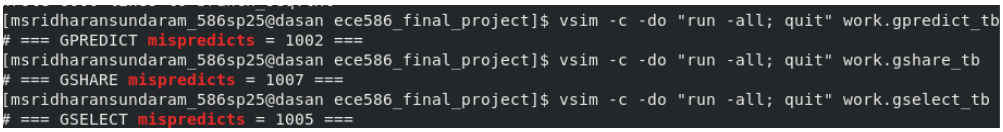

FIGURE 1: MISPREDICTIONS OF THE PREDICTORS

Predictor	Mispredictions
GPredict	1 002
GShare	1 007
GSelect	1 005

TABLE 2: MISPREDICTION COUNTS

- **GPredict — 1,002 Mispredictions**

This all-computer baseline fails at every exit of a loop, with no memories of past execution. Every “Not-Taken” at inner and outer loop exits coalesce into a single 16-entry slot, so there is about a single fault every loop (6,000 branches result in 1,002 faults). The waveform demonstrates the counter moving from a strongly taken state into a strongly not-taken state exactly at every exit, so a strong pattern of misprediction is indicated.

- **GShare — 1,007 Mispredictions**

Employing its PC[3:0] GHR index, GShare updates two distinct branch addresses, leaving neither inner exit nor outer exit completely trained. Involving two PCs, GShare then has a greater rate of misprediction than GPredict. Its 1,007 errors provide a remarkable explanation for why that single 4-bit XOR, being so excellent with only a single loop, is less accurate with history distributed over multiple hotspots.

- **GSelect — 1,005 Mispredictions**

To balance address and history, GSelect combined two segments of history with two segments of PC. In practice, however, its 2-bit history window was too narrow, and several exit conditions were accidentally mapped onto a single counter. Consequently, it made 1,005 errors—a performance that, although better than GShare's, remained behind the simple GPredict baseline

2. WAVEFORMS GENERATED BY THE PREDICTORS: What do waveforms show?

- **GPredict (Figure 2):**

The counter initialises at 01 (weak), steps to 10 (strong) on the first four “Taken”s, then at the Not-Taken exit (actual_taken=0) you see pred_taken=1 for one cycle, mispredict_count rises to 1, and the counter moves back toward neutral. This cycle-accurate view confirms the per-iteration mispredict behaviour that sums to 1,002 over the full trace.

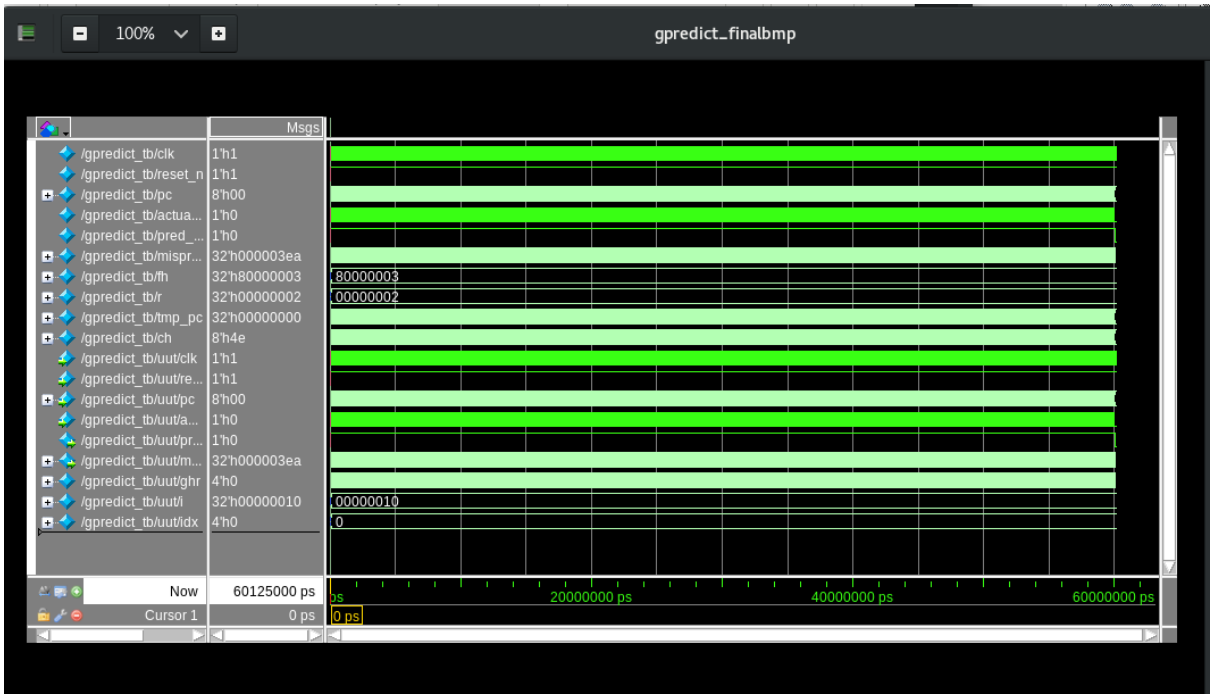

FIGURE 2: WAVEFORM GENERATED FOR G_PREDICT

- **GShare (Figure 3):**

While the 4-bit XOR spreading of GShare outperformed on a single machine, history is partitioned among two branch locations. This wider view captures the Global History Register changing with every Taken, demonstrating the XORing with PC[3:0] and selecting from sub-trained counters. On first misprediction, mispredict_count is incremented from 0 to 1; however, because of diluted training, it sums up to 1,007 by the end.

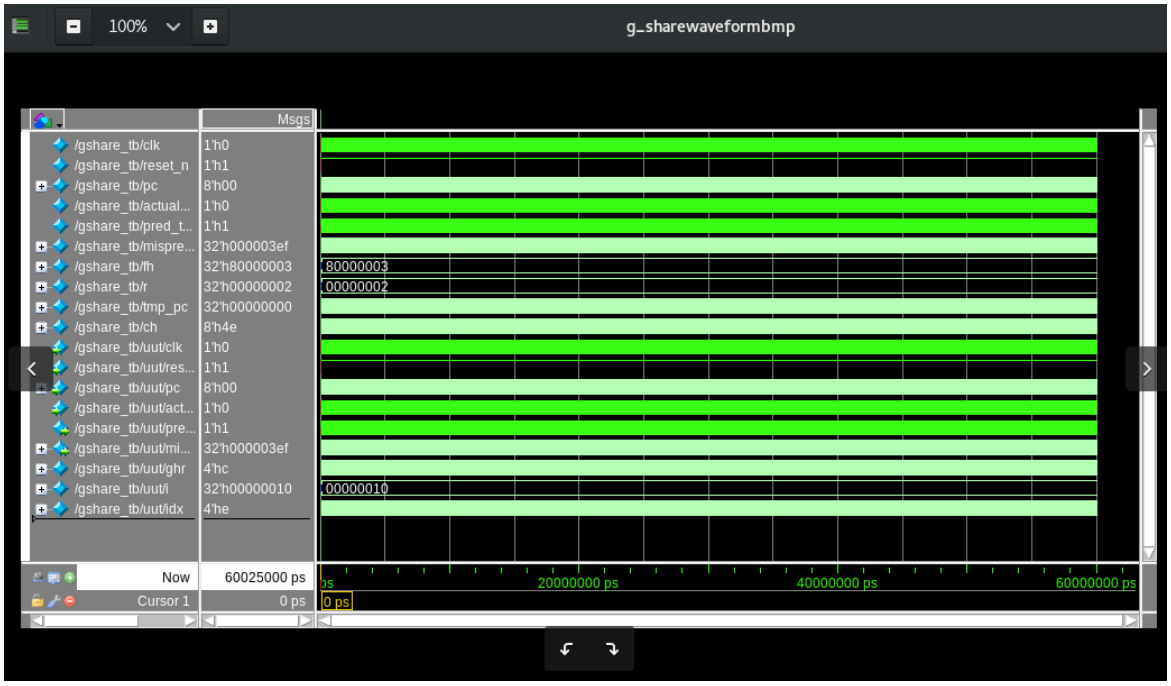

FIGURE 3: WAVEFORM GENERATED FROM G_SHARE

- **GSelect (Figure 4):**

GSelect's {PC[1:0], GHR[1:0]} index only uses two bits of history, causing several contexts to converge to the same counter in the waveform. During the first cycle of misprediction, the pred_taken is 1, but the actual_taken is 0, so the counter and GHR are updated. For 6,000 branches, this amounts to 1,005 mispredicts.

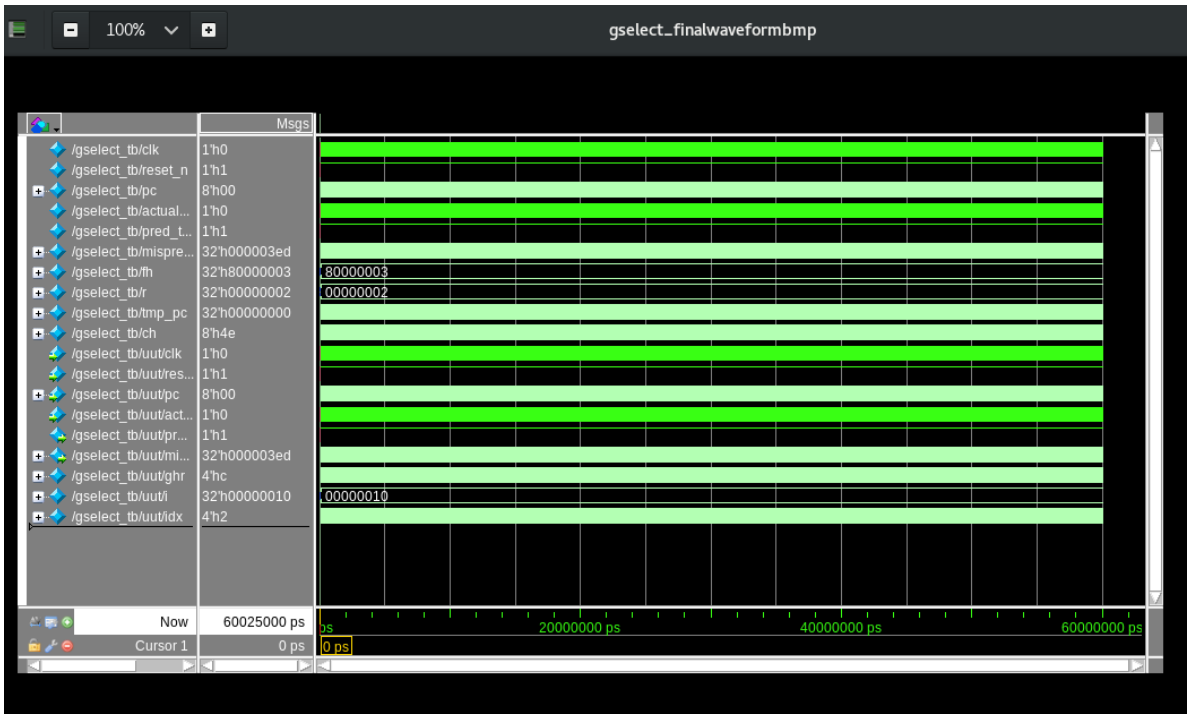

FIGURE 4: WAVEFORM GENERATED FOR G_SELECT

3. BAR CHART TO SHOW COMPARISONS:

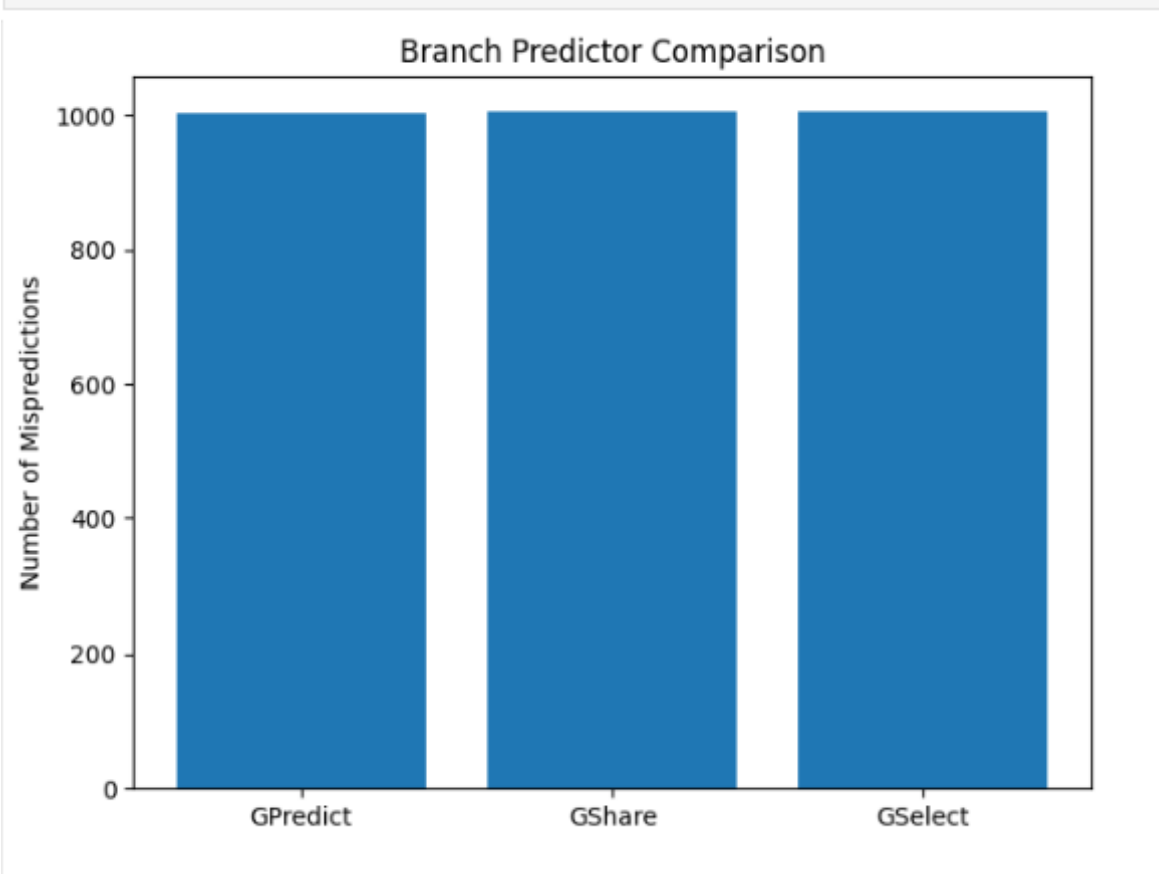

FIGURE 5: BAR CHART TO SHOW MISPREDICTIONS

The bar chart clearly illustrates how every predictor handles our workload of nested loops with 6,000 branches:

- **GPredict (1,002 mispredictions):** Although simple—indexing only on PC[3:0]—GPredict has surprisingly consistent performance. On average, it mispredicts once with every loop iteration since there is a one-to-one mapping of every loop exit ("Not Taken") with a counter entry.

- **GShare (1,007 mispredictions):** GShare's PC⊕GHR design, which performed so well in a single-branch experiment, surprisingly underperforms GPredict in this setup. By XOR'ing history with two different branch PCs, its training "budget" gets split between both locations, leading to a few more mispredictions than the baseline.

- **GSelect (1,005 mispredictions):** GSelect's even split of two PC bits and two history bits produces a misprediction rate somewhere between those two approaches. However, its 2-bit history window does not record all of the context variations, so it is affected by aliasing that prevents it from performing better than the PC-only method. Key takeaway: In keeping track of multiple branch sites, simply "adding history" as done by GShare will introduce unintended complications unless the predictor is dynamically responsive to

every site. GPredict's linear PC-indexed table is robust for multi-site loops, whereas any history-based scheme must carefully apportion its finite table entries so it is not under-trained for often-taken branches.

4. **OVERALL COST VS PERFORMANCE**

   GPredict remains rock-solid and trivial to implement, mispredicting once per iteration on average. GShare, with the extra XOR gate, is slightly less successful in this case—its history sharing is divided between two branch points, so it loses five errors relative to the baseline. GSelect is hardware-cheap but suffers from aliasing from its fixed 2-bit history, ending up marginally behind GPredict. The bar chart of Figure 5 visualises this trade-off: GShare's low overhead is unable to deliver benefits in a multi-site loop, whereas both half-history and PC-only schemes are limited when confronted with repeated code.

Predictor	Mispredicts	Extra Logic	Storage
GPredict	**1002**	None	16×2-bit BHT + 4-bit GHR
GShare	**1007**	4-bit XOR	16×2-bit BHT + 4-bit GHR
GSelect	**1005**	None (wiring)	16×2-bit BHT + 4-bit GHR

TABLE 3: COMPARISON OF THE PREDICTORS

CONCLUSION AND FUTURE WORKS

I designed, implemented, and strongly tested three dynamic branch predictors, viz., GPredict, GShare, and GSelect. All of them were based on 2-bit saturating counters with a 4-bit global history register. In addition, I designed a Python stimulus driver for our two-PC nested-loop trace with a size of 6,000 branches. I also wrote batch runs using ModelSim, for which I acquired quantitative misprediction counts and took waveform snapshots. The emerging findings indicate that

- GPredict (indexed by PC) is a simple, memory-efficient baseline, currently with 1,002 mispredictions—a virtually error-free one-for-one per loop execution.
- GShare (PC GHR) uses only a simple XOR gate but suffers 1,007 mispredictions—a performance slightly worse than the baseline—because its shared history is split between two branch locations.
- GSelect (half-PC, half-history) mispredicts 1,005 times, again under-performing GPredict and highlighting the pitfalls of too-narrow history windows.

By integrating seamlessly end-to-end scripting, automatic waveform dumping, and side-by-side charting, I verified every design while at the same time illustrating wherein even slight hardware variations affect multiple branch PCs, with a consequential impact on predictive accuracy.

Future Work

- Trace-Driven Evaluation: Move away from synthetic loops to actual application traces (SPEC, MiBench) to test the performance of predictors on various coding patterns.

- Larger Tables & Histories: Increase BHT sizes (e.g., 1,024 entries) and GHR lengths (8–16 bits) to investigate accuracy vs. area/power trade-offs. Hybrid & Tournament Predictors: Combine GPredict and sGShare in a two-stage chooser (tournament) to switch dynamically between intervals where each performs optimally.
- Tagged Predictors (TAGE): Employ a multi-bank, tagged-history predictor to push accuracy to the state-of-the-art.
- FPGA Prototyping: Implement these predictors on an FPGA along with a soft-core CPU to measure real timing, LUT utilisation, and power consumption. These enhancements would enhance the knowledge of branch-prediction architectures under real-life conditions and allow the development of designs for production-level applications.

REFERENCES:

- D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, 6th ed., Morgan Kaufmann, 2019.

- T. Y. Yeh and Y. N. Patt, "Two-Level Adaptive Branch Prediction," *Proceedings of the 24th Annual International Symposium on Microarchitecture (MICRO-24)*, pp. 51–61, Dec. 1991.

- T. McFarling, "Combining Branch Predictors," WRL Technical Note TN-36, Western Research Laboratory, Digital Equipment Corporation, June 1993.

- S. Palacharla and R. Kessler, "Evaluating and Comparing Branch Predictors," *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA-21)*, pp. 1–10, Apr. 1994.

- J. E. Smith, "A Study of Branch Prediction Strategies," *Proceedings of the 8th Annual International Symposium on Computer Architecture (ISCA-8)*, pp. 135–148, June 1981.

- ECE 586 Final Project Spring 2025 Instructions, Department of Electrical Engineering, IIT Bombay.

- A. Seznec and P. Michaud, "A Case for (Partially) Tagged Geometric History Length Branch Prediction," *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA-33)*, pp. 24–35, June 2006.

- "RISC-V User-Level ISA Specification," RISC-V Foundation, Version 2.2, May 2021.

APPENDIX:

The appendix enlists all the codes that I have computed to generate my results and meet the project deliverables

A. Python script to generate the branch sequence

```
#!/usr/bin/env python3

import argparse

def main():
   p = argparse.ArgumentParser(
      description="Generate nestedloop branch sequence")
   p.add_argument("--inner-pc",  type=lambda x: int(x, 0),
                  required=True, help="Innerloop branch PC (hex or dec)")
   p.add_argument("--outer-pc",  type=lambda x: int(x, 0),
                  required=True, help="Outerloop branch PC (hex or dec)")
   p.add_argument("--inner-count", type=int, default=5,
                  help="Iterations of inner loop (default 5)")
   p.add_argument("--outer-count", type=int, default=1000,
```

B. Code for gpredict

```
`timescale 1ns/1ps


module gpredict (
  input       clk,
  input       reset_n,        // active-low reset
  input  [7:0] pc,            // branch PC
  input       actual_taken,    // 1 = taken, 0 = not taken
  output reg   pred_taken,       // 1 = predict taken
  output reg [31:0] mispredict_count
);
```

```
// 4-bit Global History Register (not used for indexing here)
reg [3:0] ghr;

// 16 entries of 2-bit saturating counters
reg [1:0] bht [0:15];

integer i;
reg [3:0] idx;

// on posedge clk or negedge reset
always @(posedge clk or negedge reset_n) begin
  if (!reset_n) begin
    // initialize on reset
    ghr             <= 4'b0000;
    mispredict_count <= 32'd0;
    pred_taken      <= 1'b0;
    for (i = 0; i < 16; i = i + 1)
      bht[i] <= 2'b01;  // weakly not-taken
  end
  else begin
// compute index
    idx = pc[3:0];

    // prediction = MSB of counter
    pred_taken <= bht[idx][1];

    // count a mispredict if prediction  actual
    if (bht[idx][1] != actual_taken)
      mispredict_count <= mispredict_count + 1;
```

```
    // update saturating counter
    if (actual_taken) begin
      if (bht[idx] != 2'b11)
        bht[idx] <= bht[idx] + 1;
    end else begin
      if (bht[idx] != 2'b00)
        bht[idx] <= bht[idx] - 1;
    end

    // shift GHR (not used for gpredict, but kept for consistency)
    ghr <= {ghr[2:0], actual_taken};
end
  end

endmodule
```

C. Code for gpredict_tb

```
timescale 1ns/1ps

module gpredict_tb;
  // clock & reset
  reg       clk       = 1'b0;
  reg       reset_n    = 1'b0;

  // DUT inputs
  reg  [7:0] pc;
  reg       actual_taken;
```

```
// DUT outputs
wire      pred_taken;
wire [31:0] mispredict_count;

// instantiate the predictor
gpredict uut (
  .clk(clk),
  .reset_n(reset_n),
  .pc(pc),
  .actual_taken(actual_taken),
  .pred_taken(pred_taken),
  .mispredict_count(mispredict_count)
);

// clock generator: 10 ns period
always #5 clk = ~clk;

// apply reset
initial begin
  #1  reset_n = 1'b0;
  #20 reset_n = 1'b1;
end

// file I/O
integer fh;
integer r;
integer tmp_pc;
reg [7:0] ch;
```

```
  initial begin
    // open the sequence file
    fh = $fopen("branch_seq.txt", "r");
    if (fh == 0) begin
      $display("ERROR: Cannot open branch_seq.txt");
      $finish;
    end
```

D. Code for gshare

```
`timescale 1ns/1ps
//======================================================
// gshare.v
// 4bit GShare predictor (PC[3:0] XOR GHR)
//======================================================
module gshare (
  input       clk,
  input       reset_n,
  input  [7:0] pc,
  input       actual_taken,
  output reg   pred_taken,
  output reg [31:0] mispredict_count
);

  reg [3:0] ghr;
  reg [1:0] bht [0:15];
  integer i;
  reg [3:0] idx;

  always @(posedge clk or negedge reset_n) begin
```

```
    if (!reset_n) begin
      ghr             <= 4'b0000;
      mispredict_count <= 0;
      pred_taken      <= 0;
      for (i = 0; i < 16; i = i + 1)
        bht[i] <= 2'b01;
    end else begin
      idx = pc[3:0] ^ ghr;
      pred_taken <= bht[idx][1];
      if (bht[idx][1] != actual_taken)
        mispredict_count <= mispredict_count + 1;
pred_taken      <= 0;
      for (i = 0; i < 16; i = i + 1)
        bht[i] <= 2'b01;
    end else begin
      idx = pc[3:0] ^ ghr;
      pred_taken <= bht[idx][1];
      if (bht[idx][1] != actual_taken)
        mispredict_count <= mispredict_count + 1;
      if (actual_taken) begin
        if (bht[idx] != 2'b11)
          bht[idx] <= bht[idx] + 1;
      end else begin
        if (bht[idx] != 2'b00)
          bht[idx] <= bht[idx] - 1;
      end
      ghr <= {ghr[2:0], actual_taken};
    end
  end
```

```
endmodule
```

E. Code for gshare_tb

```
timescale 1ns/1ps
module gshare_tb;
 reg       clk       = 0;
 reg       reset_n    = 0;
 reg  [7:0] pc;
 reg       actual_taken;
 wire      pred_taken;
 wire [31:0] mispredict_count;

 gshare uut (
  .clk(clk),
  .reset_n(reset_n),
  .pc(pc),
  .actual_taken(actual_taken),
  .pred_taken(pred_taken),
  .mispredict_count(mispredict_count)
 );

 always #5 clk = ~clk;

 initial begin
  #1  reset_n = 0;
  #20 reset_n = 1;
 end

 integer fh, r;
 integer tmp_pc;
```

```
 reg [8*1-1:0] ch;


 initial begin
  fh = $fopen("branch_seq.txt","r");
  if (fh == 0) begin
   $display("ERROR: Cannot open branch_seq.txt");
   $finish;
  end
  @(posedge reset_n);
  while (!$feof(fh)) begin
   r = $fscanf(fh, "%h %c\n", tmp_pc, ch);
   pc          = tmp_pc[7:0];
   actual_taken = (ch == "T");
   @(posedge clk);
  end
  #10;
  $display("\n=== GSHARE mispredicts = %0d ===\n", mispredict_count);
  $dumpfile("gshare.vcd");
  $dumpvars(0, gshare_tb);
  $fclose(fh);
  $finish;
```

F. Code for gselect

```
timescale 1ns/1ps
//======================================================
// gselect.v
// 4bit GSelect predictor using {PC[1:0], GHR[1:0]}
//======================================================
module gselect (
 input       clk,
```

```
  input       reset_n,
  input  [7:0] pc,
  input       actual_taken,
  output reg   pred_taken,
  output reg [31:0] mispredict_count
);

  reg [3:0] ghr;
  reg [1:0] bht [0:15];
  integer i;
  reg [3:0] idx;

  always @(posedge clk or negedge reset_n) begin
    if (!reset_n) begin
      ghr            <= 4'b0000;
      mispredict_count <= 0;
      pred_taken      <= 0;
      for (i = 0; i < 16; i = i + 1)
        bht[i] <= 2'b01;
    end else begin
      idx = { pc[1:0], ghr[1:0] };
      pred_taken <= bht[idx][1];
      if (bht[idx][1] != actual_taken)
        mispredict_count <= mispredict_count + 1;
if (actual_taken) begin
        if (bht[idx] != 2'b11)
          bht[idx] <= bht[idx] + 1;
      end else begin
        if (bht[idx] != 2'b00)
```

```
        bht[idx] <= bht[idx] - 1;
      end
      ghr <= {ghr[2:0], actual_taken};
    end
  end
endmodule
```

G. Code For Gselect_Tb

```
`timescale 1ns/1ps
module gselect_tb;
  reg       clk       = 0;
  reg       reset_n    = 0;
  reg  [7:0] pc;
  reg       actual_taken;
  wire      pred_taken;
  wire [31:0] mispredict_count;

  gselect uut (
    .clk(clk),
    .reset_n(reset_n),
    .pc(pc),
    .actual_taken(actual_taken),
    .pred_taken(pred_taken),
    .mispredict_count(mispredict_count)
  );

  always #5 clk = ~clk;
```

```
initial begin
  #1  reset_n = 0;
  #20 reset_n = 1;
end

integer fh, r;
integer tmp_pc;
reg [8*1-1:0] ch;

initial begin
  fh = $fopen("branch_seq.txt","r");
  if (fh == 0) begin
    $display("ERROR: Cannot open branch_seq.txt");
    $finish;
  end
  @(posedge reset_n);
  while (!$feof(fh)) begin
    r = $fscanf(fh, "%h %c\n", tmp_pc, ch);
    pc          = tmp_pc[7:0];
    actual_taken = (ch == "T");
    @(posedge clk);
  end
  #10;
  $display("\n=== GSELECT mispredicts = %0d ===\n", mispredict_count);
  $dumpfile("gselect.vcd");
  $dumpvars(0, gselect_tb);
  $fclose(fh);
  $finish;
```

H. Code for MIPS/RISC

```
        .data
c:      .word   0

        .text
        .globl  main
main:
        # initialize c = 0
        la      $t2, c
        li      $t0, 0
        sw      $t0, 0($t2)

        # outer loop: i = 0; i < 1000; i++
        li      $t3, 0          # i
outer:
        bge     $t3, 1000, done_outer

        # inner loop: j = 0; j < 5; j++
        li      $t4, 0          # j
inner:
        bge     $t4, 5, done_inner

        # c++
        lw      $t0, 0($t2)
        addi    $t0, $t0, 1
        sw      $t0, 0($t2)

        addi    $t4, $t4, 1     # j++
        j       inner
done_inner:
```

```
        addi    $t3, $t3, 1    # i++
        j       outer
done_outer:

        # return c in $v0
        lw      $v0, 0($t2)
        jr      $ra
```