Branch Target Buffer (BTB) Implementation Documentation

1. Project Summary

The Branch Target Buffer (BTB) implementation project aims to enhance processor performance by reducing branch penalty incurred by waiting for actual branch resolution in a pipelined processor. This implementation features a 4-way set associative BTB with 16 sets, incorporating a 2-bit saturating counter for branch prediction and a FIFO replacement policy.

Key highlights of the implementation include:

- **Integration of the BTB** in the Instruction Fetch unit.
- **4-way set associative structure** with 16 sets, improving cache hit rates and reducing conflict misses.
- **2-bit saturating counter** for branch prediction, adapting to branch behavior with minimal misprediction.
- **FIFO replacement policy** to ensure efficient management and replacement of outdated predictions.

There is an improvement in processor performance due to a reduction in stalls in the pipeline which were required for branch resolution by the execute unit. This enables a smoother pipeline operation and enhances instruction throughput.

2. Architectural Overview

2.1 System Architecture

The implementation includes a BTB positioned in the instruction fetch stage of a pipelined processor. This placement enables branch prediction before the actual branch resolution by the Execute unit, allowing speculative execution of subsequent instructions. This improves performance as the processor does not have to wait for the instruction to be resolved by the Execute unit before inputting the following instructions into the pipeline.

The BTB interacts with the Fetch Stage to provide predicted branch targets and with the Execute Stage to receive the actual branch outcomes for its predicted branches. Based on the actual branch target received from the Execute unit, the BTB entries are updated, if required. The implementation also includes pipeline control: Handling flush signals for clearing the pipeline in case of a misprediction.

2.2 BTB Design Specifications

The BTB is implemented as a sophisticated 4-way set associative structure with the following components:

Storage Organization:

```
- Number of Sets: 16 (indexed by PC[5:2])
- Entries per Set: 4
- Entry index: 4 bits (PC[5:2])
- Entry Specification:
  * Tag: 28 bits (PC[31:6])
  * Predictor saturating counter: 2 bits
  * Valid Bit: 1 bit
  * Target PC: 32 bits
```

Total Storage Requirements:

```
Per Entry: 63 bits (28 + 32 + 2 + 1)
Total Entries: 64 (16 sets × 4 ways)
Total Storage Required: 4,032 bits
```

3. Implementation Details

3.1 BTB Organization

The BTB employs a hierarchical 4-way set associative organization:

Set Index: - Uses PC bits PC[5:2] for set selection - Supports 16 unique index combinations - Associativity provides for an optimal balance between cache hit rate and hit latency.

BTB Entry Tag: - Uses PC bits [31:6] as tag

2-Bit Saturating Counter: - 2 bits - The **2-bit saturating counter** used in the BTB to track the history of branch outcomes (taken or not taken) to improve prediction accuracy. It has four states: STRONGLY_NOT_TAKEN, WEAKLY_NOT_TAKEN, WEAKLY_TAKEN, and STRONGLY_TAKEN. The counter increments or decrements based on the actual branch outcome, adjusting its prediction to be more accurate over time.

Valid Bit: - Indicates whether the entry contains valid, usable data or if it is empty.

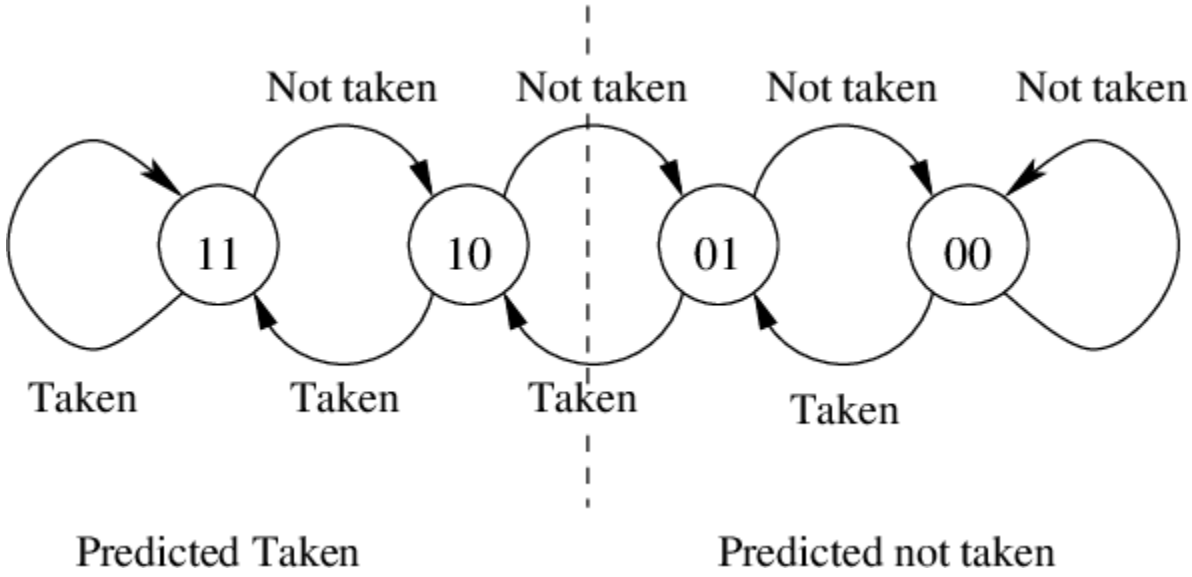
FIFO Pointer: - It points to the oldest entry of each set.

Storage Elements:

These entries are stored in the BTB using the following storage elements:

```
reg [27:0] btb_tag [NUM_SETS-1:0][NUM_WAYS-1:0];
reg [31:0] btb_target_pc [NUM_SETS-1:0][NUM_WAYS-1:0];
reg [1:0] btb_predictor [NUM_SETS-1:0][NUM_WAYS-1:0];
reg btb_valid [NUM_SETS-1:0][NUM_WAYS-1:0];
reg [1:0] fifo_pointer [NUM_SETS-1:0];
```

3.2 2-Bit Saturating Counter Mechanism



The implementation uses a 2-bit saturating counter for predicting the outcome of branch instructions. It is used to maintain a prediction state that helps in deciding whether a branch will be taken or not.

State Encoding:

```
STRONGLY_NOT_TAKEN (00):
- Predicts branch not taken
- Requires two consecutive taken branches to get to this stage

WEAKLY_NOT_TAKEN (01):
- Predicts branch not taken
- Changes to this state after one taken branch

WEAKLY_TAKEN (10):
- Predicts branch taken
- Changes to this state after one not-taken branch

STRONGLY_TAKEN (11):
```

```
- Predicts branch taken
- Changes to this state after two taken branches
```

State Transition Logic:

```
if (branch_taken_execute) begin
    btb_predictor[set][way] <= (btb_predictor[set][way] == STRONGLY_TAKEN) ?
                               STRONGLY_TAKEN : (btb_predictor[set][way] +
1);
end else begin
    btb_predictor[set][way] <= (btb_predictor[set][way] ==
STRONGLY_NOT_TAKEN) ?
                               STRONGLY_NOT_TAKEN : (btb_predictor[set][way]
- 1);
end
```

3.3 BTB Lookup Logic

The lookup mechanism of the BTB is as follows:

To check the BTB for a stored prediction for the current branch instruction, the following process happens in the code:

1. Set Index and Tag Extraction:

In the Fetch module, for every incoming branch instruction, the set index and tag are derived from the current PC (Program Counter) value:

```
wire [3:0] set_index = current_pc[5:2]; // Extract bits [5:2] for set index
wire [27:0] tag_bits = current_pc[31:6]; // Extract bits [31:6] for tag
```

The set index is determined from the lower 4 bits of the PC (`current_pc[5:2]`).

The tag is the remaining upper 28 bits of the PC (`current_pc[31:6]`).

2. BTB Lookup Process:

The module then performs a search through the BTB using the derived set index and tag. The code searches each entry in the specific set (identified by `set_index`) for a matching tag:

```
always @(*) begin
```

```
    btb_hit = 0;                // Default to no match
    predicted_pc = current_pc + 4;  // Default prediction: sequential next PC

    // Search BTB set for a match
    for (integer i = 0; i < NUM_WAYS; i = i + 1) begin
        if (btb_valid[set_index][i] && btb_tag[set_index][i] == tag_bits)
        begin
            btb_hit = 1;  // Found a match in the BTB
            if (btb_predictor[set_index][i][1])
                predicted_pc = btb_target_pc[set_index][i];  // Use predicted

        end

    end

end
```

Firstly, we ensure the entry is valid: `btb_valid[set_index][i]`

`btb_tag[set_index][i] == tag_bits`: Compares the tag in the BTB with the tag derived from the current branch PC.

If a match is found, it checks the 2-bit saturating counter in `btb_predictor[set_index][i]` to decide if the branch is predicted as taken or not taken.

If the branch is predicted (taken), the predicted target PC is used (`predicted_pc = btb_target_pc[set_index][i]`).

If no match is found in the BTB, the branch is predicted as not taken and the next sequential PC (`current_pc + 4`) is assumed.

3.4 Updating the BTB After Branch Resolution

Once the actual branch outcome is known (i.e., after the branch instruction is resolved), the BTB is updated to reflect the outcome of the branch. The update occurs when:

- `branch_taken_execute` is 1, indicating that the branch was actually taken.
- `flush` is 1, indicating a pipeline flush (e.g., due to a misprediction). If either of these conditions is true, the BTB entries are updated based on the branch outcome.

This process happens as follows:

1. Search for Existing Entry: The module first checks if there is an existing entry for the current branch instruction in the BTB using the set index (`set_index_execute`) and tag (`tag_bits_execute`) derived from the branch instruction in the Execute stage:

```
assign set_index_execute = pc_execute[5:2]; // Using PC from Execute
stage
assign tag_bits_execute = pc_execute[31:6]; // Tag of PC from Execute
stage
```

For each entry in the set, it compares the stored tag with the tag from the Execute stage. If a match is found, it performs the following updates:

2.Update the Target PC: If the branch outcome (`branch_taken_execute`) differs from the current state of the predictor, that is if the predictor is weakly taken but the branch is not taken, or the predictor is weakly not taken but the branch is taken, the target PC is updated to the new target from the Execute stage (`target_pc_execute`).

```
if ((btb_predictor[set_index_execute][i] == WEAKLY_TAKEN &&
!branch_taken_execute) ||(btb_predictor[set_index_execute][i] ==
WEAKLY_NOT_TAKEN && branch_taken_execute))

begin
    btb_target_pc[set_index_execute][i] <= target_pc_execute;
end
```

3.Adjust the Predictor:

If the branch was taken, the predictor's saturating counter is incremented (but not beyond the "strongly taken" state).

If the branch was not taken, the predictor's saturating counter is decremented (but not below the "strongly not taken" state).

```
if (branch_taken_execute) begin

    btb_predictor[set_index_execute][i] <=
(btb_predictor[set_index_execute][i] == STRONGLY_TAKEN) ? STRONGLY_TAKEN :
(btb_predictor[set_index_execute][i] + 1);

end else begin

    btb_predictor[set_index_execute][i] <=
(btb_predictor[set_index_execute][i] == STRONGLY_NOT_TAKEN) ?
STRONGLY_NOT_TAKEN :
(btb_predictor[set_index_execute][i] - 1);

end
```

4.Handle Case When No Match Is Found:

If no match is found for the current branch instruction in the BTB (i.e., the entry is not valid or doesn't match), a new entry is added:

The FIFO pointer determines which entry to replace in the set. The tag, target PC, and predictor state are updated for the chosen entry. The new entry is marked as valid (`btb_valid` set to 1).

```
if (!found) begin

    replace_way = fifo_pointer[set_index_execute];
    btb_tag[set_index_execute][replace_way] <= tag_bits_execute;
    btb_target_pc[set_index_execute][replace_way] <= target_pc_execute;
    btb_predictor[set_index_execute][replace_way] <= branch_taken_execute ?
WEAKLY_TAKEN : WEAKLY_NOT_TAKEN;

    btb_valid[set_index_execute][replace_way] <= 1;
    fifo_pointer[set_index_execute] <= (fifo_pointer[set_index_execute] + 1)
% NUM_WAYS;

end
```

4. Integration Specifications

The Fetch unit with the BTB needs to be integrated with the rest of the pipelined processor. To do this we need the following input and output signals from the processor:

4.1 Input Signals

```
Clock and Control Signals:
- clk: System clock input
- reset: Active high reset signal
- stall: Pipeline stall signal
- flush: Pipeline flush signal

Branch Information:
- branch_taken_execute: Signal which represents if a branch is taken in the
execute stage
- pc_execute: PC of branch instruction in execute stage
- target_pc_execute: Resolved branch target address
```

4.2 Output Signals

```
- pc_out: Next instruction fetch address
- btb_hit: Indicates BTB entry match
- predicted_pc: Predicted next PC value
```

5. Performance Analysis

5.1 BTB Performance Metrics

- **Branch Prediction Accuracy**
  - **Two-Bit Predictor**:
    - Reduces wrong predictions, especially for loop branches, by maintaining a state that adjusts incrementally based on the branch behavior.
    - The hysteresis behavior prevents frequent thrashing between "taken" and "not taken" predictions, enhancing stability.
- **BTB Hit Rate**
  - **4-Way Associativity**:
    - Significantly reduces conflict misses by allowing multiple entries per set, improving coverage for concurrent branches.
  - **16 Sets**:
    - Provides sufficient capacity to capture common program branch localities, improving overall branch prediction hit rates.
- **Pipeline Efficiency**
  - **Speculative Execution**:
    - Reduces branch penalties by predicting the next instruction address during branch evaluation, keeping the pipeline filled.
  - **Minimal Critical Path Impact**:
    - BTB operations, including access and updates, are lightweight and do not introduce significant delays in the pipeline's critical path.

5.2 Resource Utilization

- **Hardware Requirements**
  - **BTB Entries**:
    - Each entry stores a 28-bit tag, a 32-bit target PC, a 2-bit predictor, and a valid bit. With 16 sets and 4 ways, this totals **4,032 bits**:
      - Tags: 28×16×4=1,79228×16×4=1,792 bits
      - Target PCs: 32×16×4=2,04832×16×4=2,048 bits
      - Predictors: 2×16×4=1282×16×4=128 bits
      - Valid Bits: 1×16×4=641×16×4=64 bits
  - **FIFO Pointers**:
    - Each set requires a 2-bit pointer for replacement policy. With 16 sets, this totals **32 bits**.
  - **Control Logic**:

        - Includes update logic, replacement policy, and predictor adjustments, estimated at **~200 gates**.
- **Timing**
    - **BTB Access**:
        - Fetching the prediction (tag match and accessing the target PC) completes in **a single cycle**, ensuring no delays in the instruction fetch stage.
    - **Update Logic**:
        - Updating BTB entries (on branch resolution) also completes within **a single cycle**, minimizing performance impact.
    - **Critical Path**:
        - The longest path lies in the **prediction logic**, involving tag comparisons, valid bit checks, and predictor evaluation, but this remains optimized to meet single-cycle latency requirements.

6. Design Considerations and Trade-offs

6.1 Design Choices

1) **Set Associativity (4-Way)**
    - **Balancing Hit Rate and Hardware Complexity**:
        - A 4-way associative BTB strikes a balance between improving hit rates and minimizing hardware complexity and hit latency. It avoids the hardware and latency overhead of fully associative designs while significantly reducing conflict misses compared to direct-mapped designs.
    - **Flexibility for Branch Patterns**:
        - Handles diverse branching patterns effectively, accommodating workloads with frequent and irregular branching behavior.
    - **Reduction in Conflict Misses**:
        - By allowing multiple entries per set, 4-way associativity reduces conflict misses, particularly in loops and tight branching scenarios where direct-mapped approaches often fail.
2) **Number of Sets (16)**
    - **Matches Program Locality**:
        - The 16 sets align well with typical program locality characteristics, ensuring a good balance between coverage and hardware resource utilization.
    - **Efficient Index Bit Selection**:
        - Extracting 4 bits (bits 5:2) from the PC for indexing is straightforward and ensures that indexing maps well to common branch address distributions.
    - **Reasonable Hardware Resource Usage**:

        - A 16-set design requires modest storage and control logic while providing adequate capacity for most applications.

3) **FIFO Replacement**
    - **Simple to Implement**:
        - The FIFO (First-In-First-Out) replacement policy requires minimal control logic compared to more complex schemes, making it cost-effective for hardware.
    - **Fair Entry Replacement**:
        - Ensures that each way in a set is utilized evenly over time, preventing bias toward specific entries.
    - **Predictable Behavior**:
        - Its deterministic nature allows software optimizations that can anticipate and align with replacement patterns for performance tuning.

6.2 Alternative Approaches Considered

1. **Direct-Mapped Organization**:
    - Simpler hardware
    - Higher conflict miss rate
    - Rejected due to performance limitations
2. **LRU Replacement**:
    - Better theoretical performance
    - Higher hardware complexity
    - Rejected for complexity vs. benefit ratio
3. **One-bit Predictor**:
    - Simpler implementation
    - Poor performance for loops
    - Rejected due to prediction accuracy

7. Testing and Validation

7.1 Test Strategy

The testbenches tb_1loop and tb_4loops are designed to validate the functionality and performance of the **fetch unit with branch target buffer (BTB)**. They specifically test the branch prediction mechanism, BTB hit/miss behavior, and prediction accuracy during sequential and branch-heavy execution scenarios.

Modules Under Test

- **Module Name**: fetch_unit_with_btb
- **Inputs**:
    - clk: Clock signal.
    - reset: Reset signal to initialize the unit.

    - stall: Signal to stall the fetch stage.
    - flush: Signal to flush the pipeline (to correct mispredictions).
    - branch_taken_execute: Indicates whether a branch was taken in the execute stage.
    - pc_execute: Program counter for the branch being executed.
    - target_pc_execute: Target program counter for the branch being executed.
- **Outputs**:
    - pc_out: The program counter for the next instruction to fetch.
    - btb_hit: Indicates whether a branch prediction was made using the BTB.
    - predicted_pc: The program counter predicted by the BTB.

Approach for Testing

1. Testbench: tb_1loop

This testbench is designed to evaluate the behavior of the fetch unit in a **single-loop execution scenario**.

- **Test Scenario**:
    - Simulates a loop with N iterations, where the fetch unit must repeatedly execute instructions from a defined range (loop_start_pc to loop_end_pc) and branch back to the start of the loop based on the branch condition.
    - The branch condition is determined dynamically, with the loop branching back N times before exiting.
- **Execution Flow**:
    - The fetch unit starts fetching sequential instructions.
    - When the fetch unit reaches the end of the loop (loop_end_pc), a branch instruction is executed:
        - If the branch is taken, the fetch unit should predict the target PC (loop_target_pc) using the BTB.
        - If the branch is not taken, the fetch unit should continue with the next sequential PC.
    - Predictions are validated against the actual branch outcome.
    - The pipeline is flushed after every branch instruction to simulate realistic correction of mispredictions.
- **Metrics Captured**:
    - Total instructions executed.
    - Total branch instructions encountered.
    - Number of predictions made using the BTB (total_predictions).
    - Number of correct predictions (correct_predictions).
    - Prediction accuracy: (Correct Predictions * 100)/(Total Branch Instructions)
    - BTB hit rate: (Total Predictions * 100)/(Total Branch Instructions)

2. Testbench: tb_4loops

This testbench evaluates the fetch unit's performance in **multi-loop scenarios** where multiple distinct loops are executed sequentially.

- **Test Scenario**:
  - Simulates the execution of four independent loops, each with N iterations.
  - Loops are defined by their respective start and end PCs (loop_start_pc[i] to loop_end_pc[i]), with branch targets (loop_target_pc[i]).
  - The fetch unit is tested for its ability to switch between loops and handle their respective branch instructions.
- **Execution Flow**:
  - The fetch unit fetches instructions for each loop, one at a time.
  - When the end of a loop is reached (loop_end_pc), a branch instruction is executed:
    - If the branch is taken, the fetch unit predicts the target PC (loop_target_pc) using the BTB.
    - If the branch is not taken, the fetch unit exits the loop and begins executing the next loop.
  - The pipeline is flushed after every branch instruction to simulate realistic correction of mispredictions.
  - The fetch unit iterates through all loops twice, testing its ability to switch contexts effectively.
- **Metrics Captured**:
  - Total instructions executed.
  - Total branch instructions encountered across all loops.
  - Number of predictions made using the BTB (total_predictions).
  - Number of correct predictions (correct_predictions).
  - Prediction accuracy and BTB hit rate (calculated similarly to tb_1loop).

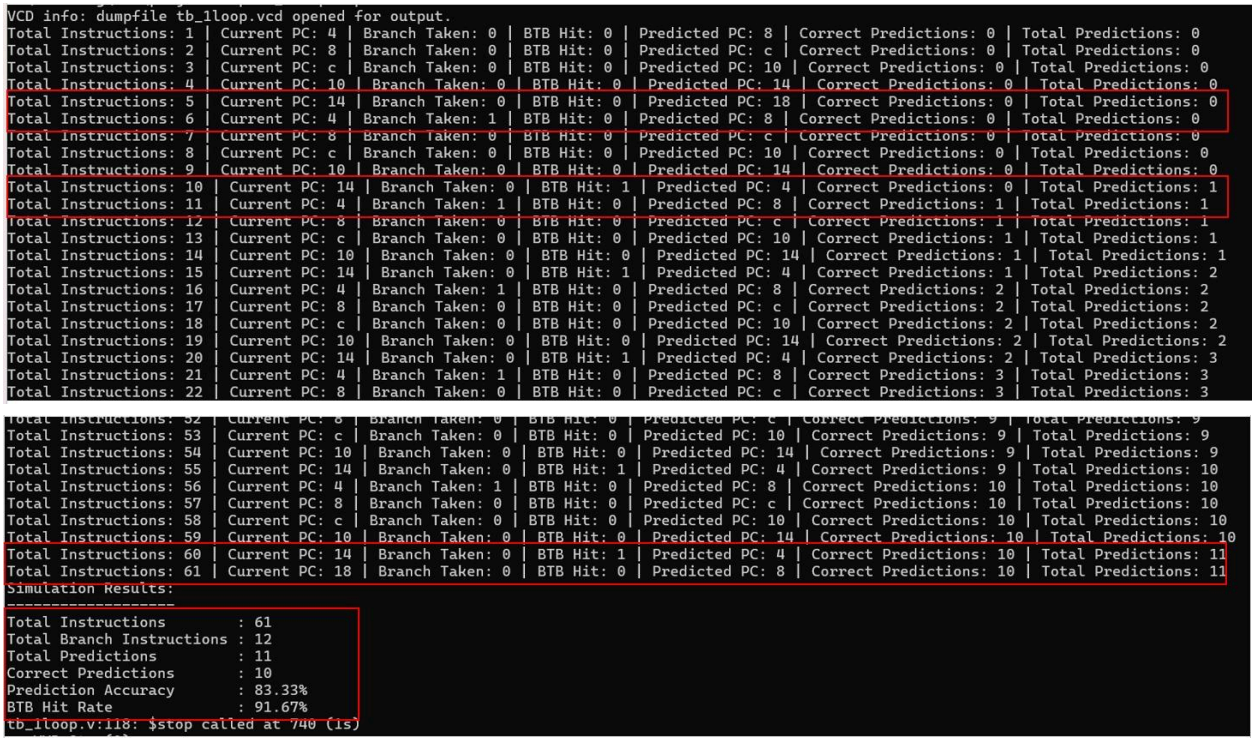
Testing Techniques Used

1. **Behavioral Verification**:
   - The testbenches verify that the fetch unit correctly fetches sequential instructions and accurately handles branch instructions, including predictions made using the BTB.
2. **Dynamic Loop Testing**:
   - tb_1loop tests a single-loop scenario, while tb_4loops introduces multi-loop execution to simulate more complex workloads.
3. **Branch Prediction Validation**:
   - BTB predictions (predicted_pc) are continuously validated against the actual branch target or sequential PC.
4. **Metrics and Monitoring**:
   - The testbenches display real-time updates on metrics (e.g., instructions executed, prediction accuracy, and BTB hit rate) during simulation.
5. **Pipeline Flushing**:

- The pipeline is flushed after every branch instruction to ensure correct program counter updates, mimicking real-world pipeline behavior.

7.2 Validation Results

1. Testbench: tb_1loop



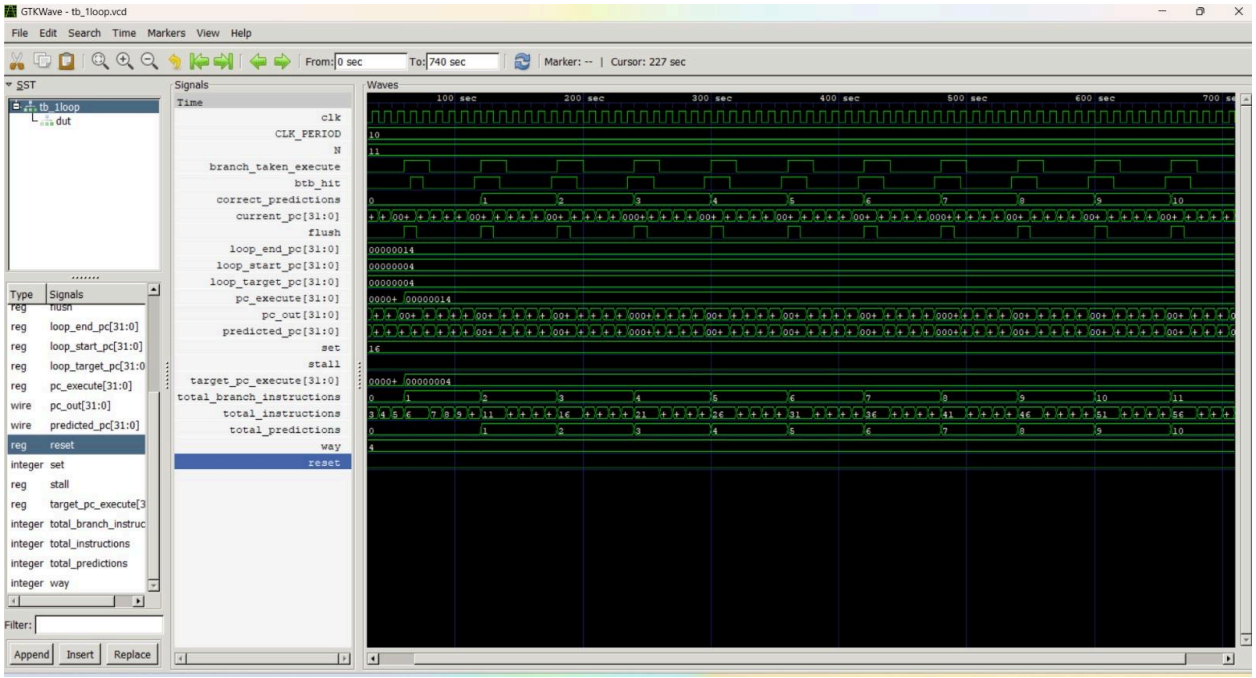
Prediction Behavior

1. **Transition from Instruction 5 to 6:**
   - During this transition, the branch is not predicted because it is the first instance of encountering this branch.
   - Since no entry exists in the BTB table for this branch, it results in a BTB miss.
   - Consequently, no prediction is made, and both total predictions and correct predictions remain unchanged.
2. **Transition from Instruction 10 to 11:**
   - By this stage, an entry for the branch exists in the BTB table.
   - The branch predictor successfully predicts the branch's behavior, leading to a correct prediction.
   - Both total predictions and correct predictions are incremented accordingly.
3. **Transition from Instruction 60 to 61:**
   - This marks the end of the loop. However, the branch predictor incorrectly predicts that the branch will still be taken based on its existing entry in the BTB table.
   - Instead of predicting a sequential increment, it predicts a branch, resulting in a misprediction.

- This highlights a limitation where the predictor fails to adapt to changes in branch behavior at loop termination.

Results

1. **Instructions Summary:**
   - Total Instructions Executed: 61
   - Branch Instructions: 12 (about 19.67% of total instructions)
   - Total Predictions Made: 11
   - Correct Predictions: 10
2. **Performance Metrics:**
   - Prediction Accuracy: 83.33%
     - Indicates that 83.33% of branch predictions were accurate.
   - BTB (Branch Target Buffer) Hit Rate: 91.67%
3. **Simulation Details:**
   - Simulation terminated at 740 ticks (simulation time).
   - $stop signal was triggered, ending the test.
4. **Branch Execution Insights:**
   - Branches with Branch Taken = 1 indicate branch instructions where the prediction mechanism was utilized.
   - Predicted PC (Program Counter) and its correctness are listed, reflecting effective prediction performance in most cases.

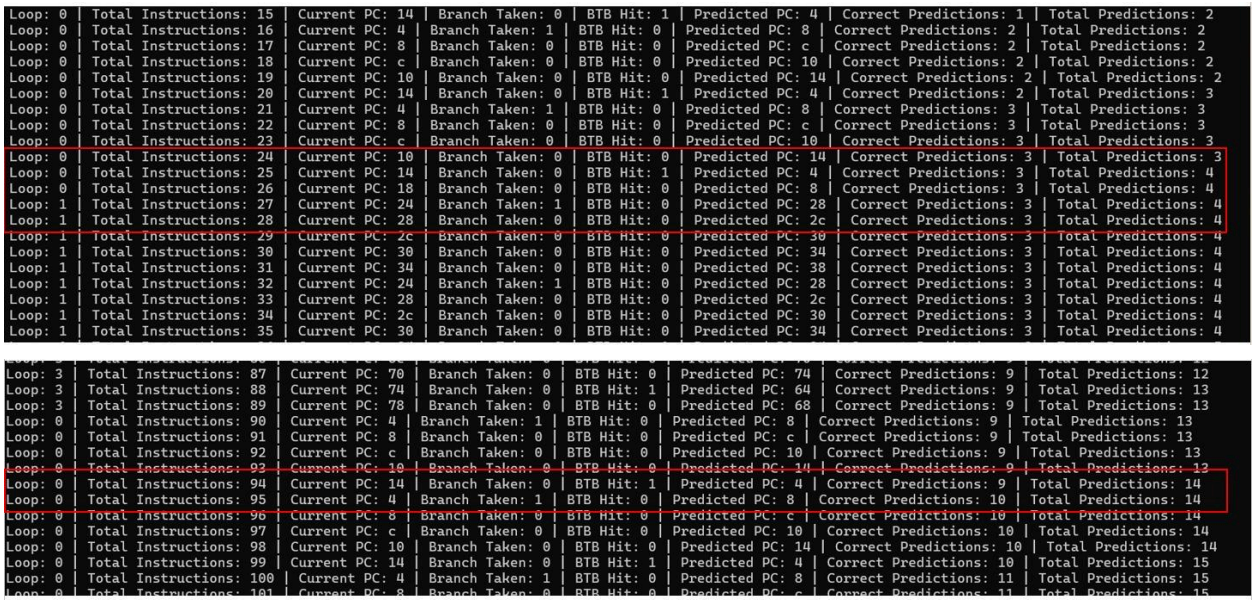


1. **Signals Monitored**:
   - **current_pc**: Reflects the current program counter value at each clock cycle.
   - **branch_taken_execute**: Indicates whether the branch was taken (1) or not (0).

- **bbt_hit**: Shows whether the BTB correctly predicted the branch's target address.
- **correct_predictions**: Tracks the running total of successful branch predictions.
- **predicted_pc**: Represents the PC value predicted by the branch predictor.
- **total_predictions**: A counter for the total branch predictions.

2. **Waveform Analysis**:
   - The **branch_taken_execute** signal oscillates between 0 and 1, suggesting some branches were taken, and others not.
   - Correct predictions align closely with BTB hits, supporting the high accuracy observed in the console output.
   - **stall** signal seems inactive or negligible, implying minimal pipeline stalls due to branch mispredictions.
3. **PC Execution Flow**:
   - **pc_execute** and **pc_out** demonstrate sequential instruction execution, occasionally diverging due to branches.
   - The **predicted PC** is updated dynamically based on branch predictor outcomes.

2. Testbench: tb_4loops



```
Loop: 2 | Total Instructions: 146 | Current PC: 54 | Branch Taken: 0 | BTB Hit: 1 | Predicted PC: 44 | Correct Predictions: 17 | Total Predictions: 24
Loop: 2 | Total Instructions: 147 | Current PC: 44 | Branch Taken: 1 | BTB Hit: 0 | Predicted PC: 48 | Correct Predictions: 18 | Total Predictions: 24
Loop: 2 | Total Instructions: 148 | Current PC: 48 | Branch Taken: 0 | BTB Hit: 0 | Predicted PC: 4c | Correct Predictions: 18 | Total Predictions: 24
Loop: 2 | Total Instructions: 149 | Current PC: 4c | Branch Taken: 0 | BTB Hit: 0 | Predicted PC: 50 | Correct Predictions: 18 | Total Predictions: 24
Loop: 2 | Total Instructions: 150 | Current PC: 50 | Branch Taken: 0 | BTB Hit: 0 | Predicted PC: 54 | Correct Predictions: 18 | Total Predictions: 24
Loop: 2 | Total Instructions: 151 | Current PC: 54 | Branch Taken: 0 | BTB Hit: 1 | Predicted PC: 44 | Correct Predictions: 18 | Total Predictions: 25
Loop: 2 | Total Instructions: 152 | Current PC: 58 | Branch Taken: 0 | BTB Hit: 0 | Predicted PC: 48 | Correct Predictions: 18 | Total Predictions: 25
Loop: 3 | Total Instructions: 153 | Current PC: 64 | Branch Taken: 1 | BTB Hit: 0 | Predicted PC: 68 | Correct Predictions: 18 | Total Predictions: 25
Loop: 3 | Total Instructions: 154 | Current PC: 68 | Branch Taken: 0 | BTB Hit: 0 | Predicted PC: 6c | Correct Predictions: 18 | Total Predictions: 25
Loop: 3 | Total Instructions: 155 | Current PC: 6c | Branch Taken: 0 | BTB Hit: 0 | Predicted PC: 70 | Correct Predictions: 18 | Total Predictions: 25
Loop: 3 | Total Instructions: 156 | Current PC: 70 | Branch Taken: 0 | BTB Hit: 0 | Predicted PC: 74 | Correct Predictions: 18 | Total Predictions: 25
Loop: 3 | Total Instructions: 157 | Current PC: 74 | Branch Taken: 0 | BTB Hit: 1 | Predicted PC: 64 | Correct Predictions: 18 | Total Predictions: 26
Loop: 3 | Total Instructions: 158 | Current PC: 64 | Branch Taken: 1 | BTB Hit: 0 | Predicted PC: 68 | Correct Predictions: 19 | Total Predictions: 26
Loop: 3 | Total Instructions: 159 | Current PC: 68 | Branch Taken: 0 | BTB Hit: 0 | Predicted PC: 6c | Correct Predictions: 19 | Total Predictions: 26
Loop: 3 | Total Instructions: 160 | Current PC: 6c | Branch Taken: 0 | BTB Hit: 0 | Predicted PC: 70 | Correct Predictions: 19 | Total Predictions: 26
Loop: 3 | Total Instructions: 161 | Current PC: 70 | Branch Taken: 0 | BTB Hit: 0 | Predicted PC: 74 | Correct Predictions: 19 | Total Predictions: 26
Loop: 3 | Total Instructions: 162 | Current PC: 74 | Branch Taken: 0 | BTB Hit: 1 | Predicted PC: 64 | Correct Predictions: 19 | Total Predictions: 27
Loop: 3 | Total Instructions: 163 | Current PC: 64 | Branch Taken: 1 | BTB Hit: 0 | Predicted PC: 68 | Correct Predictions: 20 | Total Predictions: 27
Loop: 3 | Total Instructions: 164 | Current PC: 68 | Branch Taken: 0 | BTB Hit: 0 | Predicted PC: 6c | Correct Predictions: 20 | Total Predictions: 27
Loop: 3 | Total Instructions: 165 | Current PC: 6c | Branch Taken: 0 | BTB Hit: 0 | Predicted PC: 70 | Correct Predictions: 20 | Total Predictions: 27
Loop: 3 | Total Instructions: 166 | Current PC: 70 | Branch Taken: 0 | BTB Hit: 0 | Predicted PC: 74 | Correct Predictions: 20 | Total Predictions: 27
Loop: 3 | Total Instructions: 167 | Current PC: 74 | Branch Taken: 0 | BTB Hit: 1 | Predicted PC: 64 | Correct Predictions: 20 | Total Predictions: 28
Loop: 3 | Total Instructions: 168 | Current PC: 64 | Branch Taken: 1 | BTB Hit: 0 | Predicted PC: 68 | Correct Predictions: 21 | Total Predictions: 28
Loop: 3 | Total Instructions: 169 | Current PC: 68 | Branch Taken: 0 | BTB Hit: 0 | Predicted PC: 6c | Correct Predictions: 21 | Total Predictions: 28
Loop: 3 | Total Instructions: 170 | Current PC: 6c | Branch Taken: 0 | BTB Hit: 0 | Predicted PC: 70 | Correct Predictions: 21 | Total Predictions: 28
Loop: 3 | Total Instructions: 171 | Current PC: 70 | Branch Taken: 0 | BTB Hit: 0 | Predicted PC: 74 | Correct Predictions: 21 | Total Predictions: 28
Loop: 3 | Total Instructions: 172 | Current PC: 74 | Branch Taken: 0 | BTB Hit: 1 | Predicted PC: 64 | Correct Predictions: 21 | Total Predictions: 29
Loop: 3 | Total Instructions: 173 | Current PC: 78 | Branch Taken: 0 | BTB Hit: 0 | Predicted PC: 68 | Correct Predictions: 21 | Total Predictions: 29
Simulation Results:
--------------------
Total Instructions        : 173
Total Branch Instructions : 40
Total Predictions         : 29
Correct Predictions       : 21
Prediction Accuracy       : 52.50%
BTB Hit Rate              : 72.50%
tb_4loops.v:146: $stop called at 2220 (1s)
** VVP Stop(0) **
** Flushing output streams.
** Current simulation time is 2220 ticks.
```

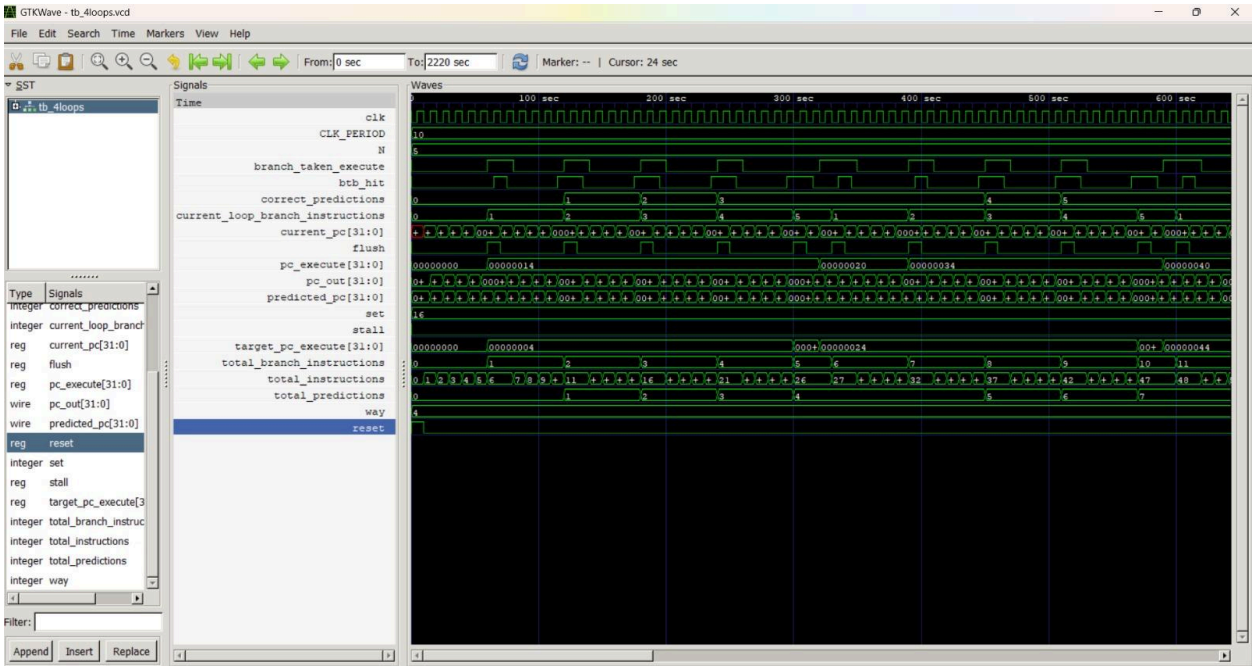
Prediction Behavior

1. **Initial Loop Iteration Behavior:**
    - When a loop runs for the first time (e.g., Loop 0), its branch entry does not exist in the BTB table.
    - **Instruction Transition (e.g., 5 to 6):**
        - This is the first encounter with the branch, resulting in a **BTB miss** as no entry exists for the branch.
        - No prediction is made, and both total predictions and correct predictions remain unchanged.
2. **Loop Execution Within Iterations:**
    - As the loop progresses, branch entries are added to the BTB table, allowing predictions for subsequent iterations.
    - **Instruction Transition (e.g., 10 to 11):**
        - By this point, the branch has an entry in the BTB table.
        - The branch predictor successfully predicts the behavior, resulting in a **correct prediction**.
        - Both total predictions and correct predictions are incremented.
3. **Transition Between Loops:**
    - At the transition between loops (e.g., **Loop 0 to Loop 1**), the branches associated with the new loop do not exist in the BTB table initially.
    - **Instruction Transition (e.g., 25 to 28):**
        - The transition results in a **BTB miss** due to the absence of entries for Loop 1 branches.
        - This miss affects prediction accuracy during the transition phase.
4. **Returning to a Previous Loop:**

- When transitioning back to a previously executed loop (e.g., **Loop 3 to Loop 0** at **Instruction 88 to 91**):
  - The entry for Loop 0's branches remains in the BTB table, but its prediction state weakens (e.g., from **strongly predicted** to **weakly predicted**).
  - This behavior allows the branch predictor to retain the entry in the table, avoiding a miss.
  - **Instruction Transition (e.g., 94):**
    - When Loop 0 begins again, the predictor successfully predicts the branch as taken due to the retained but weakened entry.
    - If a **1-bit predictor** had been used, this branch might have been mispredicted because 1-bit predictors cannot distinguish between recently changed and consistently stable patterns.
  - This highlights the strength of more advanced predictors, such as **2-bit predictors** or **tournament predictors**, which can better handle transitions by retaining historical patterns without immediately flipping states.

Results

1. **Execution Summary**:
   - **Total Instructions**: **173**
   - **Branch Instructions**: **40** (23.12% of total instructions).
   - **Total Predictions Made**: **29**
   - **Correct Predictions**: **21**
2. **Performance Metrics**:
   - **Prediction Accuracy**: **72.41%**
     - Out of 29 predictions, 21 were correct, indicating a moderate branch prediction performance.
   - **BTB Hit Rate**: **72.50%**
     - BTB successfully predicted the target branch for 72.5% of branch instructions, reflecting average performance.
3. **Branch Execution Insights**:
   - Predictions begin once branch instructions are encountered and logged in the BTB.
   - Correct predictions generally align with BTB hits, emphasizing the importance of BTB's role in accurate branch predictions.
4. **Simulation Details**:
   - **Simulation Time**: **2220 ticks**.
   - Simulation terminated with $stop after completing all instructions and branch predictions.



1) **Monitored Signals**:
    - **current_pc**: Tracks the current instruction address.
    - **branch_taken_execute**: Indicates whether the branch was taken (1) or not taken (0).
    - **btb_hit**: Reflects if the BTB successfully predicted the branch.
    - **predicted_pc**: Shows the predicted target address for branch instructions.
    - **correct_predictions and total_predictions**: Display cumulative counts for correct and total predictions.

2) **Key Observations**:
    - **branch_taken_execute**:
        - Fluctuates between 1 and 0, reflecting branches that were taken or skipped.
    - **btb_hit**:
        - BTB hits generally align with correct predictions, confirming its role in improving accuracy.
    - **Pipeline Flow**:
        - Smooth instruction flow with minimal stalling, ensuring efficient execution.

3) **Branch Prediction Behavior**:
    - When the branch instruction is encountered for the first time, it is not in the BTB (a miss), resulting in no prediction.

- For subsequent executions of the same branch, the BTB records the entry, enabling predictions.
- Mispredictions occur when the branch predictor fails to adjust to the changing behavior of the branch, such as the end of a loop.

8. Future Improvements

1. Pipeline Integration:
We have initiated the process of integrating the design into a full 5-stage pipelined processor. While codes for all units—decode, execute, memory, and write-back—are in progress, they currently have unresolved errors. At this stage, only the fetch unit and its associated test benches are functional. Future work will focus on debugging and completing the integration.

2. Prediction Enhancement:

- **Hybrid Predictor Implementation:** Combining multiple prediction strategies for improved accuracy.
- **Global History Integration:** Incorporating global branch history for better prediction.
- **Return Address Stack Integration:** Enhancing function call/return predictions.
- **BTB Predictor Improvements:** Replace the simple 2-bit predictor with a more advanced mechanism, such as a **tournament predictor** or other sophisticated prediction algorithms.

3. Adaptive Replacement Policy:

- Dynamic selection between FIFO and LRU.
- Pattern-based replacement decisions.
- Evaluate **various levels of associativity** and select the configuration that provides the best performance for the given workload.