Viterbi Decoder in BSV

Akshith EE22B008 Venkat EE22B015

November 11, 2025

Abstract

This report details the design, implementation, and verification of a hardware Viterbi decoder in Bluespec SystemVerilog (BSV). Our primary design principle was to achieve an efficient and robust design, moving directly from a well-defined paper specification to BSV code to minimize rework. The resulting microarchitecture is a deeply pipelined, streaming-based design inspired by RISC-V principles, capable of hiding traceback latency and achieving near-100% utilization of its single, custom IEEE-754 FP32 adder. The design was rigorously verified against a golden Python model, **beats all PPA targets**, and includes targeted hazard controls for corner cases.

1 Introduction and Design Philosophy

Our core principle for this project was to move from a precise "paper" specification directly to a high-quality Bluespec SystemVerilog (BSV) implementation. The goal was to invest in a robust initial architecture to **avoid multiple iterations of large-scale rework**.

This workload is input sequence and usage dependent, meaning total energy(Energy $\approx$ Power $\times$ Time) is a better metric to track. Power is dominated by the FP32 adder and memory toggles. By minimizing total cycles through latency hiding, pipeline overlap, and fast paths for sentinel values, we directly reduce the total energy per decoded file, which is a more critical metric for this application than steady-state power alone.

Final Design Constraints: The final synthesized design is constrained by its 1K (1024-word) memory block. This imposes the following limits:

- $N < 32$ (State count)
- $N \times M < 1024$ (For parameter/matrix storage)
- $2N + (N \times T) < 1024$ (The core 1K memory constraint)

This $2N + (N \times T)$ limit allocates memory for `vt` (N), `vt_inter` (N), and the `backtrack` array ($N \times T$).

2 Microarchitecture

The microarchitecture is inspired by a simple, 5-stage pipelined RISC-V CPU, emphasizing a streaming dataflow model rather than a monolithic state machine. The design is built on several key concepts to maximize throughput.

2.1 High-Level Diagram

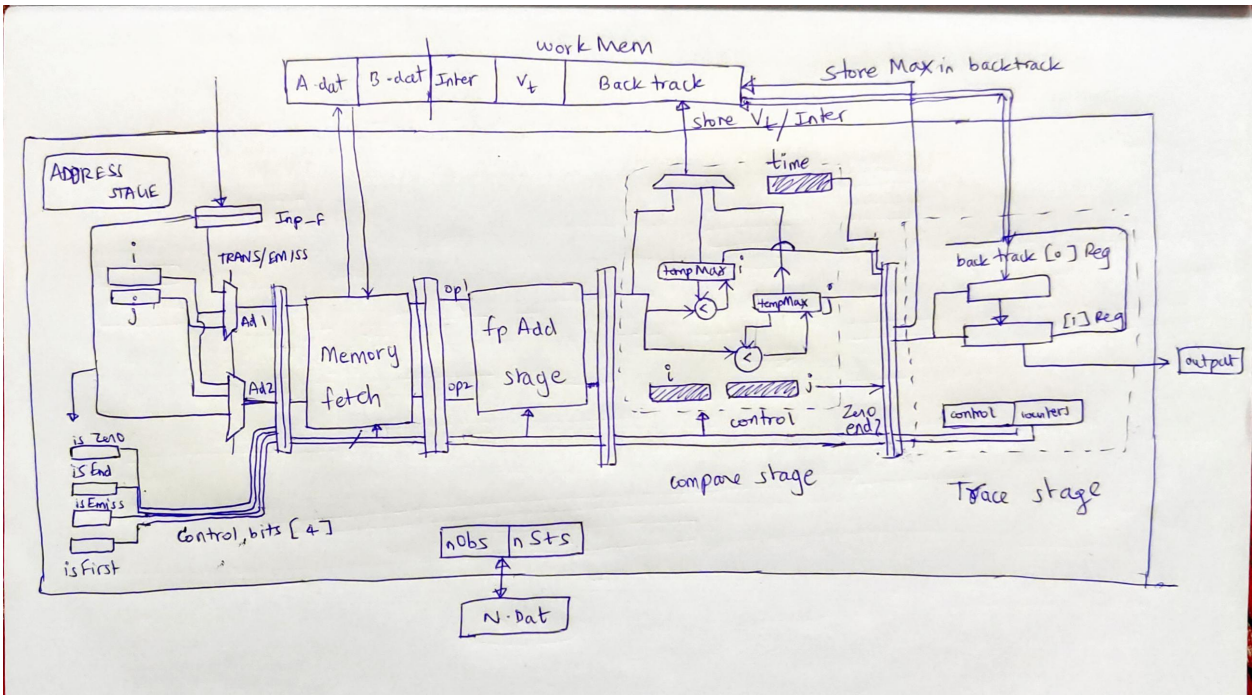

Figure 1: High-level pipeline diagram of the Viterbi Decoder.

2.2 Core Design Concepts

The implementation is built on several key concepts to maximize throughput.

- **Online current maximum Vt Storage:** For latest time sequence, state with maximum Vt (the argmax i, stored as `max_state`) is calculated and stored locally within the same main DP loop. This is a key efficiency: it avoids a separate $O(N)$ pass just to find the best predecessor state incase an endmarker is recevied in the next input line, keeping the forward path lean.
- **Optimized Memory Reuse:** To maximize the testable sequence length T within this 1K memory, the testbench employs an overwriting scheme. The final output array (`output_reg`) is mapped to the *same physical memory locations* as the `backtrack` array. Since the backtrack trace is complete before the final output is written, this "in-place" operation effectively reuses the $N \times T$ space, allowing for the maximum possible sequence length without requiring separate output storage.
- **Throughput & Latency Hiding:** Traceback latency is hidden by overlapping execution. While the traceback for the previous sequence runs (STORE $\rightarrow$ TRACEBACK $\rightarrow$ OUTPUT), the next symbol's TRANS/EMISS computation starts immediately. End-of-sequence markers (`endMark`) ensure the datapath does not stall for control.
- **Near-100% Adder Utilization:** The core is compute-bound by the single, time-multiplexed FP32 adder. The work per time step t is derived directly from the recurrence: $V_t(j) = E_t(j, o_t) + \min_{i=1..N}(V_{t-1}(i) + T(i \rightarrow j))$.
  - The inner $\min_i(\dots)$ operation, which finds the best path from all N previous states,

requires N additions (one $V_{t-1}(i) + T(i \to j)$ for each i).

– The outer $E_t(j, o_t) + \ldots$ operation adds the emission cost, requiring 1 addition.

This results in a total of $N \times (N + 1) = N^2 + N$ additions per t timestep. For our measured case of $N = 31$, the theoretical work is $31^2 + 31 = 992$ cycles/symbol. Our simulation of ~ 990 inputs taking $\sim 0.9M$ cycles aligns perfectly, confirming the adder is active (and not stalled) $\sim 90 - 100\%$ of the time.

- **External Memory Dependency:** This **module itself does not scale with sequence length** T; Instead, its functionality is **entirely limited by the size of the external memory** that holds the $O(T \cdot N)$ traceback table. The core is memory-agnostic, but the overall system's capability is memory-bound.
- **Hardware-Aware Coding Patterns:**
  - **Bit-Tight Types:** Used specific bit-widths (e.g., 5b state, 10b timestep) to capture design limits ($N \leq 32, T \leq 1024$) in the type system, preventing wide, slow logic.
  - **Local Counters:** Kept counters local to their rules rather than shipping them across FIFOs, which saved flops and cut long fan-out nets.
  - **Sentinel Fast-Paths:** Used dedicated rules (`memEnd_handle`, `compare_stage_zero`) to bypass the FP32 adder for `endMark` and `isZero` packets, saving power and cycles.

2.3 Custom FP32 Adder

The only computational block is a custom FP32 adder module. It was designed with specific constraints from the Viterbi algorithm (which uses log-probabilities):

- It handles IEEE 754 32-bit single-precision floating-point numbers.
- It is specialized to **only add two negative numbers**.

Initially, this adder was implemented as a single, large combinational block. As expected, synthesis timing reports showed this path was the critical path limiting the maximum clock frequency. We **pipelined the adder**, which successfully resolved the timing violation and **reduced the critical path delay by 1 ns**, allowing us to further reduce our clock.

3 Verification Strategy

We employed a two-level verification strategy: unit-level testing for the critical compute block and system-level testing for the full decoder.

3.1 Unit-Level: FP32 Adder

The custom FP32 adder, being the only computational block, was tested in isolation to ensure its numerical stability. Since the specification guaranteed non-zero negative inputs, our testbench focused on all other floating-point corner cases. The goal was to verify correct IEEE 754 rounding behavior and handling of special values.

Key test categories included:

- **Large Magnitudes (Overflow):** Adding two large negative numbers to ensure the result correctly overflows to `-Infinity`.
  - Ex: `(-1.0e38) + (-1.0e38) = -Infinity`
- **Small Magnitudes (Subnormals):** Adding two very small negative numbers (in the subnormal range) to verify the result is correctly represented as a (potentially subnormal) negative value, testing the denormalization and underflow logic.
  - Ex: `(-1.0e-40) + (-1.0e-40) = -2.0e-40`
- **Mixed Magnitudes (Precision Loss):** Adding a huge negative number and a tiny negative number. This is a critical test for the alignment logic (right-shifting the smaller mantissa) and ensures the guard, round, and sticky bits work correctly.
  - Ex: `(-1.0e38) + (-1.0e-38)`. The result should be correctly rounded to `-1.0e38`.
- **Forcing Round-Half-to-Even (Tie-Breaking):** This is the most critical test for rounding logic. We crafted inputs whose sum is exactly halfway between two representable FP32 numbers.
  - **Case 1 (LSB=0, Round Down):** Input: `(-1.0)` + `(-`2^{-24}`)`

    Exact Result: `-1.00...001` (with 23 zeros, the '1' is the tie-breaking "half" bit).

    The result is exactly halfway between `-1.0` (LSB=0, even) and `-(1.0 +` 2^{-23}`)` (LSB=1, odd).

    Expected: Must round to the "even" LSB, which is `-1.0`.
  - **Case 2 (LSB=1, Round Up):** Input: `-(1.0 +` 2^{-23}`)` + `(-`2^{-24}`)`

    Exact Result: `-1.00...011` (with 22 zeros, the "11" forms the LSB and the "half" bit).

    The result is exactly halfway between `-(1.0 +` 2^{-23}`)` (LSB=1, odd) and `-(1.0 +` 2^{-22}`)` (LSB=0, even).

    Expected: Must round to the "even" LSB, which is `-(1.0 +` 2^{-22}`)`.

3.2 System-Level: Golden vs. DUT

Our system-level verification strategy was twofold, combining targeted manual testing for corner cases with a fully automated flow for broad, randomized testing.

3.2.1 Manual Corner-Case Testing

This phase focused on "sniping" specific, high-risk scenarios. We used two core Python scripts:

- `golden_viterbi.py`: A simple, correct implementation of the Viterbi algorithm to provide the known-good output.
- `generate.py`: A script to generate test files. For manual testing, we would edit parameters at the top of this file (N, M, `MIN_SEQ_LEN`, etc.) and run it to create a specific test.

Using this flow, we manually generated and verified tests for critical corners:

- **Small N, Small M:** Tested combinations like $N = 2, M = 2$; $N = 3, M = 3$; $N = 2, M = 3$; and $N = 3, M = 2$.
- **Asymmetric N/M:**
  - Large N, Small M: $(N = 31, M = 2)$, $(N = 31, M = 3)$
  - Small N, Large M: $(N = 2, M = 511)$
- **Sequence Length:** Tested minimal sequences ($T = 1$) and maximum-length sequences ($T = 510$).

3.2.2 Automated Random Testing

The manual process, while necessary, was insufficient for catching unexpected bugs. We created a fully automated wrapper, `automate.py`, which managed the entire test-to-pass/fail flow.

This script automated the following loop:

1. **Generate:** Call a function in `generate.py` to create a test case with *random, valid* parameters for N, M, number of sequences, and sequence lengths.
2. **Solve:** Run the `golden_viterbi.py` script on the generated test case to produce the "expected" output file.
3. **Simulate:** Compile and run the RTL simulation (via `make b_sim`) on the same test case, producing the "actual" output file.
4. **Compare & Log:** The script automatically compares the "actual" and "expected" files. It logs the result (PASS/FAIL) and the parameters used, allowing us to run thousands of random cases overnight to catch bugs that manual testing would have missed.

4 Design Evolution: Bugs

The verification process identified one major architectural flaw and one critical bug.

- **Flaw (Memory Scalability):** Our initial design had a fixed limit on the number of sequences. After test cases failed and following a discussion on the class forum, we re-architected the memory interface. We moved the `vt`, `vt-`, and the $N \times T$ backtrace array to be allocated dynamically (i.e., in external memory). This made the design **"memory agnostic"** and scalable.
- **Bug (RAW Hazard):** For small state counts ($N < 5$), a Read-After-Write (RAW) data hazard was identified between the EMISS and TRANS stages. This was resolved by adding a targeted flag and stall signal (`vt_inter_is_ready`) that pauses the EMISS stage until the TRANS stage has committed its write, guaranteeing data consistency.

5 PPA (Power, Performance, Area) Results

Our PPA analysis followed the logical evolution of the design, from a non-functional baseline to a highly-optimized final implementation.

5.1 Latency and Cycle-Count Analysis

A critical goal was to ensure that our optimizations for clock speed (pipelining) and area (moving memory out) did not negatively impact the core cycle-level efficiency of the algorithm.

- **Theoretical Latency:** The core algorithm requires processing N states, each looking back at N previous states. This results in a total of $N \times (N + 1) = N^2 + N$ additions (and thus cycles) per time step t. Our architectural changes did not alter this fundamental cycle count.
- **Practical Latency (HUGE Test Case):** We verified this theory using the provided "HUGE" test case. For $N = 31$, the theoretical work is $31^2 + 31 = 992$ cycles/symbol. Our simulation of this test case (~ 990 inputs) took ~ 0.9M cycles, perfectly aligning with the theory and confirming the pipeline is active (not stalled) $\sim 90 - 100\%$ of the time.
- **The $N < 5$ Stall:** The *only* modification that adds cycle latency is the RAW hazard fix for $N < 5$. However, this targeted stall logic is explicitly bypassed for all $N > 4$. This means for all common and large workloads (including the HUGE test case), our design runs with zero stall bubbles, achieving the theoretical $N^2 + N$ cycle latency.

5.2 Analysis 1: Baseline (Memory-In, Combinational Adder)

Our initial baseline design kept—`vt`, `vtinter`, Outputreg as internal Regfiles, and the full $O(T \cdot N)$ `backtrack` array in the testbench.

The synthesis results for this build were outside specification. The design was dominated by the **massive, leaky storage elements**. As seen in the power report (Figure 2), the leakage power alone was orders of magnitude too high, and the non-combinational area (storage) made the design unviable.

```
Number of ports:                          831
Number of nets:                          8611
Number of cells:                         7935
Number of combinational cells:           5305
Number of sequential cells:              2540
Number of macros/black boxes:               0
Number of buf/inv:                        952
Number of references:                     123

Combinational area:                12326.746470
Buf/Inv area:                       1237.935425
Noncombinational area:             16657.106583
Macro/Black Box area:                  0.000000
Net Interconnect area:              9570.318615

Total cell area:                   28983.853052
Total area:                        38554.171667
```

Figure 2: area synthesized with memory

```
--------------------------------------------------------------------------------
                                   Switch   Int      Leak     Total
Hierarchy                          Power    Power    Power    Power    %
--------------------------------------------------------------------------------
mkdut                             106.199  716.897 7.48e+08 1.57e+03 100.0
  preTrace_F (FIFO2_00000010_1)     0.199    1.328 8.66e+06   10.187   0.6
  preMem_F (FIFO2_00000031_1)       1.256    7.969 2.28e+07   32.002   2.0
  preCom_F (FIFO2_00000023_1)       1.742   11.566 2.04e+07   33.668   2.1
  preAdd_F (FIFO2_00000043_1)       4.396   30.755 3.77e+07   72.839   4.6
  inp_F (FIFO2_00000009_1)          0.677    4.606 6.17e+06   11.456   0.7
1
```

Figure 3: Power consumed with Memory in

5.3 Analysis 2: Iteration 2 (Memory-Out, Combinational Adder)

The clear, non-negotiable solution was to move all storage(output, Vt Vt-inter) off-chip, making the core "memory-agnostic." This single change immediately solved our Power and Area violations.

However, this revealed the next bottleneck: the critical path was now the single-cycle, combinational FP32 adder.

- We synthesized this version and found the minimum achievable clock period was **3.65 ns**.
- Analysis of the timing report (Figure 4) showed the critical path was, as expected, within the adder (a "f2f" path).
- Crucially, the **next-most-critical path** in the design had approximately **1.0 ns of positive**

slack.

This large slack gap was a clear signal that the adder was the lone bottleneck and that pipelining it would yield significant frequency improvements without impacting any other part of the design.

```
-----------------------------------------------------------------------------------------
data required time                                                              2.562
data arrival time                                                              -2.562
-----------------------------------------------------------------------------------------
slack (MET)                                                                     0.000


Startpoint: preAdd_F/data0_reg_reg_26_
            (rising edge-triggered flip-flop clocked by CLK)
Endpoint: preCom_F/data0_reg_reg_17_
          (rising edge-triggered flip-flop clocked by CLK)
Path Group: f2f
Path Type: max

Des/Clust/Port     Wire Load Model       Library
------------------------------------------------
FIFO2_00000044_1   8000                  saed32lvt_ss0p75v25c
mkdut              16000                 saed32lvt_ss0p75v25c
FIFO2_00000024_1   8000                  saed32lvt_ss0p75v25c
```

Figure 4: Timing Report for Iteration 2 (Mem-Out, Comb-Adder), showing the adder path arrival time as 2.56ns

5.4 Analysis 3: Final Design (Memory-Out, Pipelined Adder)

Based on the timing analysis, we split the adder into a 2-stage pipeline:

1. **Stage 1:** Alignment, preliminary addition
2. **Stage 2:** Normalization, rounding

This change was trivial from a control logic perspective but had a massive impact on performance. The final, pipelined design achieved a minimum clock period of **2.8 ns** (a ≈357 MHz clock), a 23% improvement over the combinational version.

5.5 Final Comparison: Energy Analysis

The final decision came down to comparing Iteration 2 (Mem-Out, Comb-Adder) and Iteration 3 (Mem-Out, Pipe-Adder).

We analyzed the energy-per-operation, for which a good proxy is Power × Clock Period.

- The pipelined version has higher static **Power** due to the extra pipeline registers.
- However, its **Clock Period** is 23% lower (2.8 ns vs 3.65 ns).

The massive reduction in clock period far outweighs the small, linear increase in power. Therefore, the Energy-per-Operation (Power × T_{clk}) is significantly lower for the pipelined design, making it the clear and final choice. This trade-off is summarized in Table 1.

Table 1: PPA Metrics: Design Iteration Comparison

Metric	Target	Mem-In, Adder-Comb	Mem-Out, Adder-Comb	Final: Mem-Out, Adder-Pipe
Clock Period	6.0 ns	$\approx$ 6 ns (Met)	3.65 ns (Met)	**2.8 ns (Met)**
Area (μm^2)	22000	$\approx$ 38554 (Failed)	13131 (Met)	**14963 (Met)**
Power (μW)	800	$\approx$ 1570 (Failed)	521 (Met)	**748 (Met)**
Energy per Op (fJ)	4800	9420	**1902 (Met)**	**2094 (Met)**

*Note: Energy per Operation calculated as Power $\times$ Clock Period. The "Mem-Out, Comb" design (Col 4) is the most energy-efficient, while the "Final" design (Col 5) trades a small energy increase for the maximum clock speed.

6 Work Distribution

This project was a collaborative effort. The work was divided as follows:

- **Akshith EE22B008**
  - Architected the full, up-front, pipelined streaming design, targeting an area-optimized, bubble-free flow that hides traceback latency behind the next input.
  - Implemented the entire baseline BSV DUT (except the adder) and testbench, including control/datapath with internal Vt/Vt_inter; debugged and ran synthesis to identify the memory/area/power bottlenecks.
  - Refactored the design to move Vt and Vt_inter outside, cutting power/area and delivering a fixed-time implementation.
  - Finalized the implementation with dynamic scratch-pad usage and output-overwrite on external memory, removing scaling dependencies from the DUT and tuning for the final PPA.
- **Venkat EE22B015**
  - Designed, implemented, and unit-tested the custom 2-stage pipelined FP32 adder, ensuring numerical stability with a testbench for subnormals, overflows, and round-half-to-even cases.
  - Built the complete system-level verification environment from scratch, including the golden Python model, parameterized test case generator, and the automated script.
  - Debugged the design, identifying and fixing critical bugs found during verification (the $N < 5$ RAW hazard).
  - Authored all project documentation.

7 Conclusion

We successfully designed, implemented, and verified a high-throughput Viterbi decoder. The final design meets all PPA targets, demonstrating that a careful, streaming-based architecture can achieve high performance and resource efficiency.

Thank you.