IIT Madras

Department of Computer Science and Engineering

CS6230: July–Nov 2025

Project : Viterbi Decoder

Due: Nov 09 2025 11:59 PM

1 Ground rules

1. Students must work in **groups of 2** for this project.
2. Sharing of code between student groups and/or copying from public git repositories will be considered **cheating**. The TAs will scan source code through various tools available to us for detecting cheating. Source code that is flagged by these tools will receive the following actions:
   - **Zero credits for the project**.
   - **Final grade will be two grades lower than the actual grade obtained by the student.**
3. Students are encouraged to engage in white board discussions, which is an essential aspect of the project.
4. Students must design the module using Bluespec System Verilog (BSV), and subsequently, compile, verify and synthesize the same using the Shakti Docker Image provided.

2 Project description

The Viterbi algorithm is a dynamic programming technique commonly used with Hidden Markov Models(HMM). Given a sequence of observations, it computes the most likely sequence of hidden states that could have generated these observations.

The objective is to design a Viterbi Decoder that implements the Viterbi algorithm to solve a generic Hidden Mark Model (HMM) problem. The decoder takes in the sequences of observations and uses predefined transmission and emission probabilities to output the most probable sequence of hidden states, along with its probability score (in natural logarithmic form).

References: `https://en.wikipedia.org/wiki/Viterbi_algorithm`
`https://web.stanford.edu/~jurafsky/slp3/A.pdf`

2.1 I/O File Format Specifications

2.1.1 N.dat

This file contains two values: N and M.

- N is the number of states, stored as a 32-bit integer. The state set is defined as $Q = q_1, q_2, \ldots, q_N$, with two additional special states: the start state q_0 and the end state q_{end}. Each state is represented by a 32-bit integer.
- M is the number of possible observations. The observation set is given by $O = o_1, o_2, \ldots, o_M$, with each observation represented as a 32-bit integer.

2.1.2 A.dat

This file contains the transition probability matrix

$$A = \{a_{01}, a_{02}, \ldots, a_{N1}, \ldots, a_{NN}\}.$$

It consists of $(N + 1) \times N$ entries, capturing state transitions both from the start state q_0 and between states. All entries are stored as natural logarithms of the probabilities, using IEEE single-precision floating-point format.

2.1.3 B.dat

This file stores the emission probabilities

$$B = b_i(o_t),$$

arranged in the order $b_1(o_1), b_1(o_2), \ldots, b_1(o_M), b_2(o_1), \ldots, b_N(o_M)$.
Each probability value is stored in natural logarithmic form, using IEEE single-precision floating-point format.

2.1.4 Input.dat

This file contains multiple sequences of input observations. Each observation is a 32-bit integer. The marker `FFFFFFFF` indicates the end of one sequence. The final sequence is terminated with `32'hFFFFFFFF` followed by `32'h0`.

2.1.5 Output.dat

This file stores the results produced by the hardware simulation. For each input sequence, the following are written:

- The most probable sequence of states.
- The probability of this sequence, given in natural logarithmic form.
- The marker `32'hFFFFFFFF`.

The very last result must end with `32'h0`.

Note: The maximum number of entries in any probability matrix (transitions or emissions) is limited to 1024.
All states and observations are considered 1-indexed for the input and output sequences.

Sample Input

N_xxxx.dat

```
2
4
```

A_xxxx.dat — hex values (ln values in brackets are for understanding only)

```
BF317218  (= ln 0.5)
BF317218  (= ln 0.5)
BEB69E1A  (= ln 0.7)
BF9A1BC8  (= ln 0.3)
BF02C577  (= ln 0.6)
BF6A9207  (= ln 0.4)
```

B_xxxx.dat

```
C0135D8E  (= ln 0.1)
BFCE0210  (= ln 0.2)
BF6A9207  (= ln 0.4)
BF9A1BC8  (= ln 0.3)
BF9A1BC8  (= ln 0.3)
BF9A1BC8  (= ln 0.3)
BFCE0210  (= ln 0.2)
BFCE0210  (= ln 0.2)
```

input_xxxx.dat

```
1
3
4
FFFFFFFF
2
1
2
FFFFFFFF
0
```

Output

output.dat

```
2
1
1
C09C50F8
FFFFFFFF
2
2
1
C04679B
FFFFFFFF
0
```

3 Specification

Design a baseline Viterbi decoder and improvise the same to gain a measurable improvement in **power, performance, or logic area** . The improvised decoder design must exhibit functional correctness as that of the baseline and meet timing requirements, while showing a clear boost in the chosen metric through synthesis reports.

Core Recursion Formula

For each time step $t = 1, 2, \ldots, T - 1$ and each state j:

$$V_t(j) = \max_{1 \le i \le N} \Big(V_{t-1}(i) + \log P(s_j \mid s_i) \Big) + \log P(o_t \mid s_j)$$

where:

- $V_t(j)$ = maximum log-probability of a path ending in state j at time t,
- $P(s_j \mid s_i)$ = transition probability from state i to j,
- $P(o_t \mid s_j)$ = emission probability of observation o_t in state j.

Constraints

- The design implementation using Bluspec System Verilog must not use direct addition ('+') or multiplication ('*') operators.
- Alternative approaches must be used to realize the above operations.

Traceback

The most likely state sequence is obtained by storing the argument of the maximum (predecessor state) at each step and performing a traceback from the final state.

4 Design requirement

Implement the Viterbi Decoder using Bluespec System Verilog (BSV), as mentioned in the beginning of this specification.

5 Verification Requirement

A basic BSV testbench that instantiates the Viterbi Decoder module must be used to verify the functional correctness of the design. Initially, the testbench should ensure that the module is operational. It must then be refined to validate the design against the provided reference input-output values. The testbench must perform the following tasks:

1. **Input Loading:** The input sequences must be loaded into the module using the `mkRegFileLoad` utility inside the testbench. A maximum of two ports can be used to read data, and one port can be used to write data. In other words, in a single clock cycle, you can read a maximum of two addresses and write to one address.

2. **Output Capture:** The output sequence from the Viterbi Decoder must be captured inside the testbench using the `$fwrite` utility.

3. **Input/Output Formatting:** Example code demonstrating how to read from and write to files has been provided in the **demo GitHub repository** . All input and output must strictly follow the specified file format.

4. **Reference Model:** Inputs must also be given to a reference model developed in any programming language of preference. The outputs of the reference model should match the design output format for ease of comparison. This comparison can be automated.

Finally, the generated output file must be compared against `Output_XXXX.dat` - the reference file having the expected output for each set of inputs to verify the correctness of the Viterbi Decoder implementation.

The project directory contains two sample test cases in the `test-cases` subdirectory: 'small' and 'huge'. Students may use these test cases to validate the initial functional correctness of their design.

5.1 Evaluation Points

- Test cases developed or generated randomly should be able to reasonably cover the verification space. The justification for choosing the tests and the number of tests should be clear.
- Reference model has to be developed for checking the design output
- Extra points for automating the whole setup

6 Submission Requirements

Students are requested to submit a single ZIP file (one per team) to the assignment posted on Moodle. ZIP file must contain the following:

- The complete **source code**.
- A **detailed report** describing the **microarchitecture** and the **design/verification methodologies** used.
- The report must also include a discussion of the **synthesis reports**, clearly stating:
  - The maximum clock frequency achieved by the design.
  - The improvements observed from the initial design to the final design in terms of **power**, **performance**, and **area (PPA)** and the maximum clock frequency achieved in the design.
  - A brief explanation of how these improvements were achieved (e.g., pipeline optimizations, resource sharing, architectural changes).
- The **README** file must provide:

- An overview of the design.
- A summary of **individual contributions** by each team member.
- Clear **compilation and execution instructions**.
- The maximum clock frequency achieved.

Both team members must be able to **explain their individual contributions appropriately**. Students may refer to **blog** for frequently used Linux commands and the **demo GitHub repository** for Makefile and design reference.

7 Evaluation Criteria

This design project will account for **30%** of your final grade.