DESIGN AND STIMULATION OF LAND ROVER FIGO FSM

K. PREM CHAND

*Computer Science and Engineering (Data Science)*

*Malla Reddy College of Engineering & technology*

Hyderabad, India kotipallipremchand[@gmail.com](mailto:reddysandeep956@gmail.com)

GONGATI SANDEEP REDDY

*Computer Science and Engineering (Data Science)*

*Malla Reddy College of Engineering & technology*

Hyderabad, India [reddysandeep956@gmail.com](mailto:reddysandeep956@gmail.com)

Kunta Satya Bharadwaj *Computer Science and Engineering (Data Science)*

*Malla Reddy College of Engineering & Technology*

Hyderabad, India [satyabharadwaj2003@gmail.com](mailto:satyabharadwaj2003@gmail.com)

***Abstract*— The design and implementation of a Finite State Machine (FSM) for a Land Rover Figo vehicle is presented in this document.** **A FSM is a mathematical model that represents a system with a finite number of states and transitions between those states based on certain events or conditions. The Land Rover Figo FSM aims to simulate the vehicle's behavior by defining states such as "Initial," "start," "Drive," "Reverse," "motion," "parking," and "Emergency," and transitions triggered by events like ignition on,** **accelerator pedal press, brake pedal press, turn signal activation. The FSM is implemented using Verilog code, where each state transition is handled through a transition method. The implementation is tested by simulating various scenarios and events to verify the correct behavior of the FSM.**

**Keywords— Functional Simulation, FSM (Finite State Machine)**

1. INTRODUCTION

The implementation of the Land Rover Figo FSM aims to simulate the behavior of a Land Rover Figo vehicle using a Finite State Machine (FSM). A FSM is a mathematical model that represents a system with a finite number of states and transitions between those states based on certain events or conditions.

In this implementation, we begin by identifying the different states that the Land Rover Figo can be in, such as "Off," "Idle," "Accelerating," "Braking," and "Reverse." These states represent the various operational modes or behaviors of the vehicle.

Next, we define the transitions between these states. These transitions are triggered by specific events or conditions. For example, the transition from "Off" to "Idle" is triggered by the event "Ignition On." Similarly, pressing the accelerator pedal triggers the transition from "Idle" to "Accelerating," while pressing the brake pedal triggers the transition from "Idle" or "Accelerating" to "Braking."

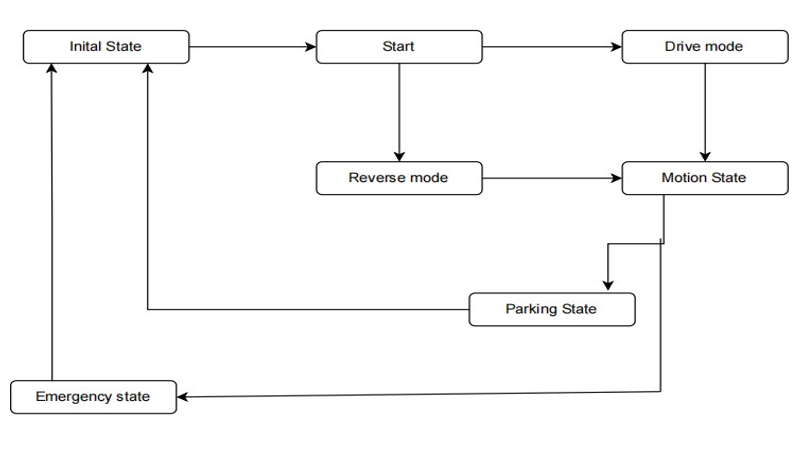
Based on the identified states and transitions, we create a state diagram, which visually represents the FSM. The state diagram consists of circles representing the states and arrows representing the transitions. The transitions are labeled with the events or conditions that trigger them.

To implement the FSM in code, we create a class called "Land Rover Figo FSM" that represents the vehicle's FSM. The class has methods to handle transitions and retrieve the current state. Inside the transition method, we check the current state of the FSM and perform appropriate state transitions based on the given event. For example, if the FSM is in the "Off" state and the event is "Ignition On," the state changes to "Idle."

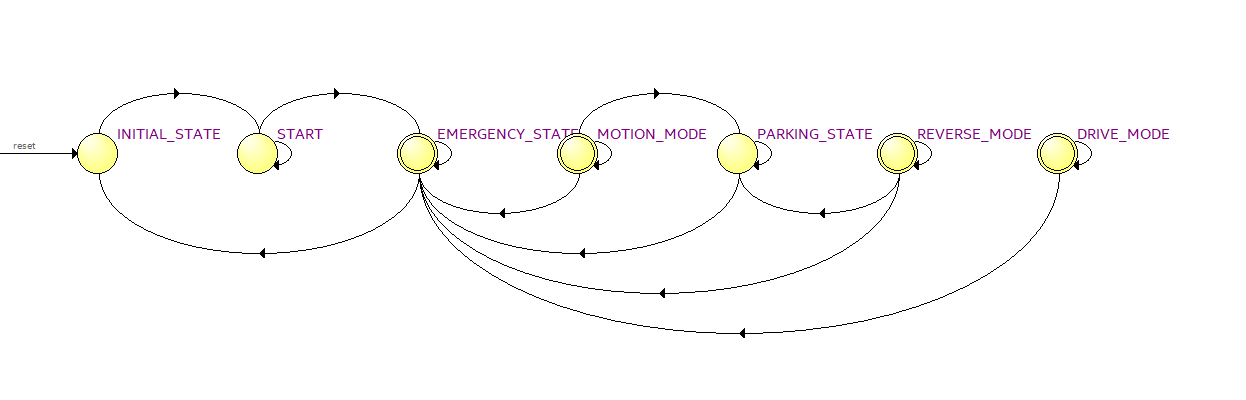
The FSM implementation can be tested by simulating various scenarios and events. By invoking the transition method with the corresponding events, we can observe how the FSM transitions between different states based on the given inputs.

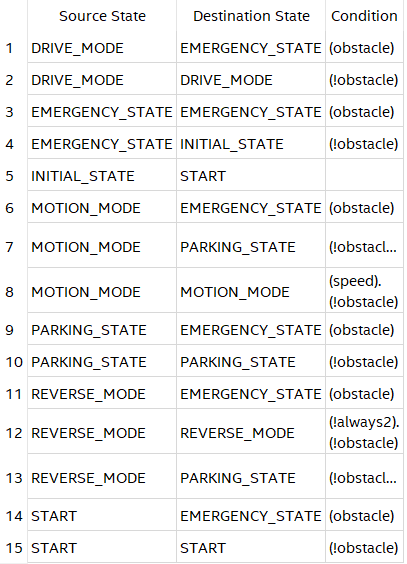
It's important to note that this implementation is a simplified example and may not cover all the possible states and transitions of a Land Rover Figo. The actual FSM implementation for a real vehicle would be more complex, considering factors such as engine status, gear shifts, vehicle speed, sensor inputs, and more. The provided example serves as a starting point, and additional states and transitions can be added based on the specific requirements and behaviors of the Land Rover Figo.

1. BLOCK DIAGRAM



III. STATE MODEL





IV. APPORACH

When writing the Verilog code for the Mealy FSM, I followed the following approach:

1- Identify the states: I identified the different states that the FSM would have based on the problem statement. In this case, the states are INITIAL\_STATE, START\_STATE, DRIVE\_MODE\_STATE,REVERSE\_MODE\_STATE, MOTION\_MODE\_STATE,PARKING\_STATE, and EMERGENCY\_STATE.

2- Define the state and output types: I defined two typedef enums to represent the state and output types. The state type, state\_t is a 3-bit logic vector, and the output type, park\_output\_t , is a 2-bit logic vector

3- Declare the state and output signals: I declared two registers, current\_state and next\_state, of type state\_t to store the current and next states, respectively. I also declared a register, park\_output, of type park\_output\_t to store the current park output.

4- Implement the FSM logic: I used an always\_ff block to describe the FSM logic. On the positive edge of the clock or reset signal, the current state is updated based on the next state. If reset is asserted, the FSM goes to the INITIAL\_STATE; otherwise, it transitions to the next state.

5- Implement the output logic: I used an always\_comb block to describe the output logic of the FSM. Depending on the current state, I used a case statement to determine the next state and set the park output accordingly.

6- Assign the park output: I used the assign statement to assign the value of the park\_output to the park output port of the module.

To create a test bench for the given Verilog code of the Mealy FSM, you can follow the following approach:

1- Create a new Verilog file: Start by creating a new Verilog file, let's call it "test\_bench.v", which will contain the test bench code.

2- Instantiate the module: Instantiate the MealyFSM module in the test bench code, providing the required input and output signals. Make sure to include the necessary module instance and port declarations.

3- Define test inputs: Declare and initialize the test input signals such as clk, reset, start, drive, reverse, motion, and emergency. These signals will be used to drive the inputs of the MealyFSM module during simulation.

4- Initialize the simulation: In the initial block of the test bench, initialize the input signals, and provide appropriate initial values for the inputs.

5- Create stimulus: Within an initial or always block, create a stimulus by applying different input combinations to the MealyFSM module. You can use procedural logic such as for loops or if statements to control the inputs and simulate various scenarios.

6- Capture and verify outputs: After applying the input stimuli, capture and verify the output signals from the MealyFSM module. You can use assertions or print statements to check the expected output values based on the provided input combinations.

7- Simulate the test bench: Add the necessary simulation directives to your test bench code. Use a simulation tool (e.g., ModelSim, Questa, or Vivado Simulator) to compile and simulate the test bench. Analyze the waveform to ensure the Mealy FSM behaves as expected.

8- Perform functional testing: Validate the behavior of the MealyFSM by checking the output values against the expected results for different input sequences

V. WORK FLOW

Processes and Tasks:

1. Project Creation:

- Create a new project in Intel Quartus software.

- Specify the project name, location, and target FPGA device.

2. Design Entry:

- Write the Verilog code for your FSM design.

- Define the states, inputs, outputs, and transitions according to your desired FSM behavior.

3. Design Compilation and Synthesis:

- Add the Verilog code file to the Quartus project.

- Run the compilation and synthesis process to generate the netlist for your design.

- This step converts your Verilog code into the hardware description language (HDL) that can be implemented on the FPGA.

4. Functional Simulation:

- Use the ModelSim simulator integrated with Quartus to perform functional simulation.

- Create a test bench to apply input stimuli and verify the expected outputs.

- Use waveform analysis to debug and validate the behavior of your FSM design.

5. Design Implementation:

- Configure the specific settings and constraints for your design.

- Define clock frequencies, pin assignments, I/O standards, and other necessary parameters.

- Apply optimizations and specify constraints based on your target FPGA device is Cyclone V SoC

6. Fitter Placement and Routing:

- Run the Fitter process in Quartus.

- The Fitter maps and places the synthesized design onto the physical resources of the target FPGA device.

- It optimizes the placement and routing to meet timing requirements and make efficient use of FPGA resources.

7. Timing Analysis and Optimization:

- Run timing analysis to verify that your design meets the required timing constraints.

- Perform optimizations such as pipelining, retiming, or register duplication to improve performance and meet timing requirements.

8. FPGA Programming:

- Program the FPGA with the generated programming files using the Quartus Programmer.

- Connect the FPGA board to your computer and select the appropriate programming mode (e.g., JTAG or AS mode).

- Configure the FPGA with the programming file to load your FSM design onto the device.

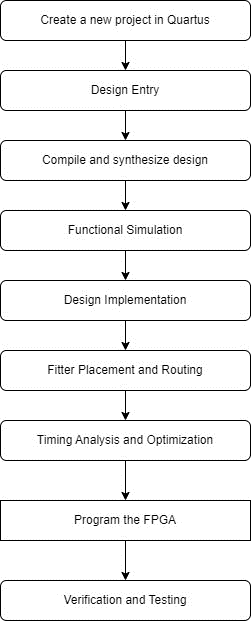
9. Verification and Testing:

- Test the functionality of your FSM design on the FPGA board.

- Verify that the implemented FSM behaves as expected and transitions between states correctly.

1

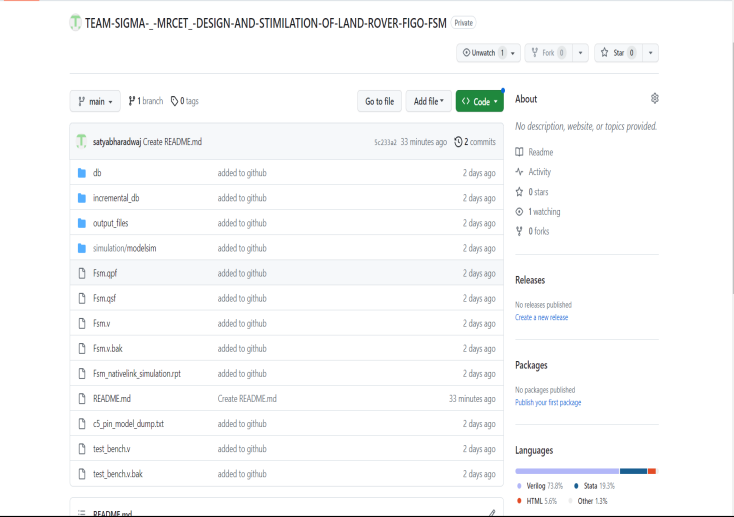
.F FLOW CHART



This flowchart outlines the sequential steps involved in the development process, from project creation to verification and refinement. Each step represents a key milestone or task that contributes to the successful implementation of the FSM design on the target FPGA device.

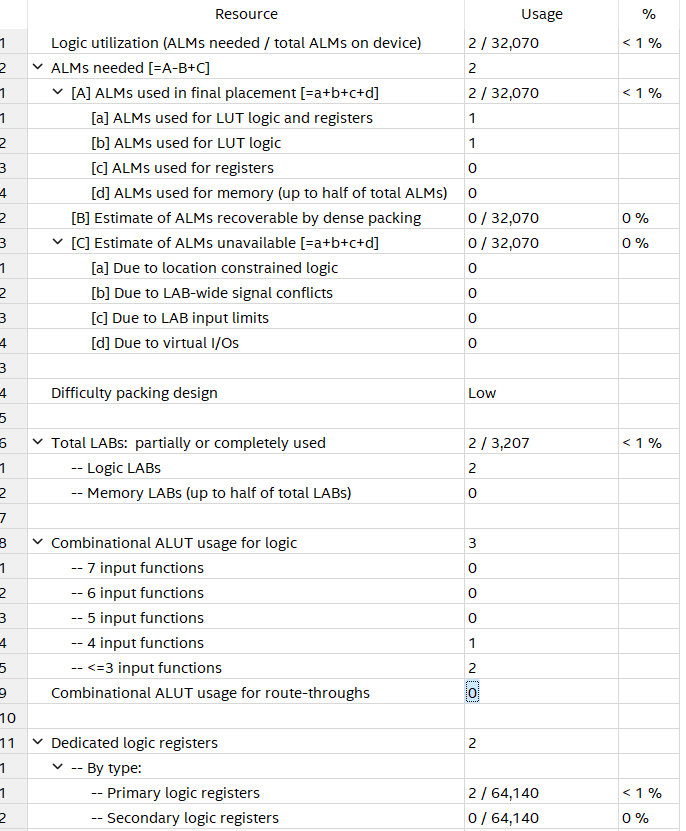
VI. GITHUB

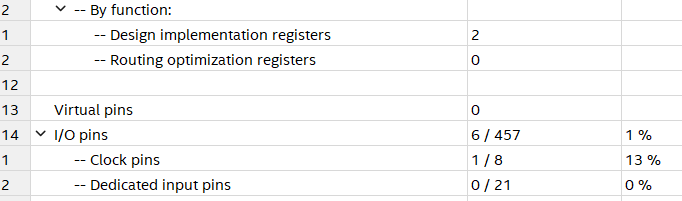
Link :https://github.com/satyabharadwaj/TEAM-SIGMA-\_-MRCET\_-DESIGN-AND-STIMILATION-OF-LAND- ROVER-FIGO-FSM.git



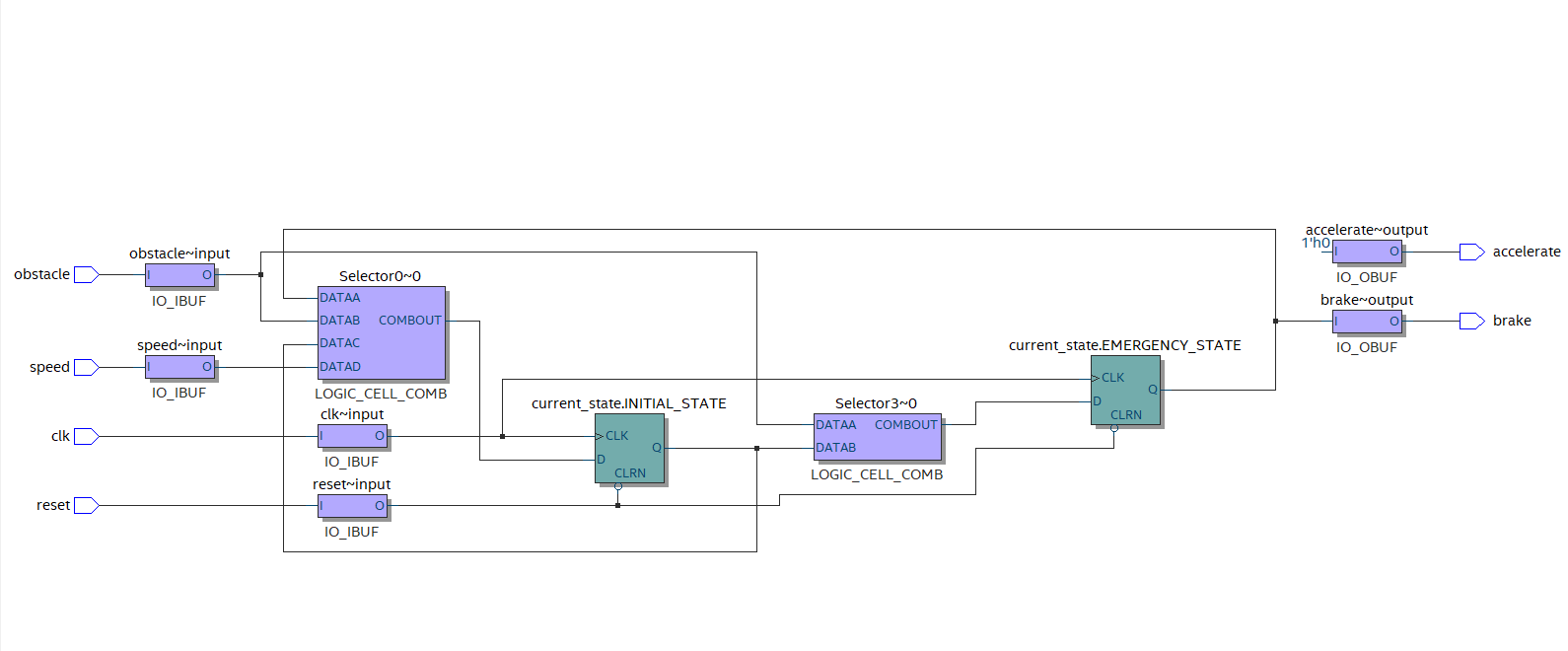
VII. OUTPUT SCREENS

Resource Usage Summary:

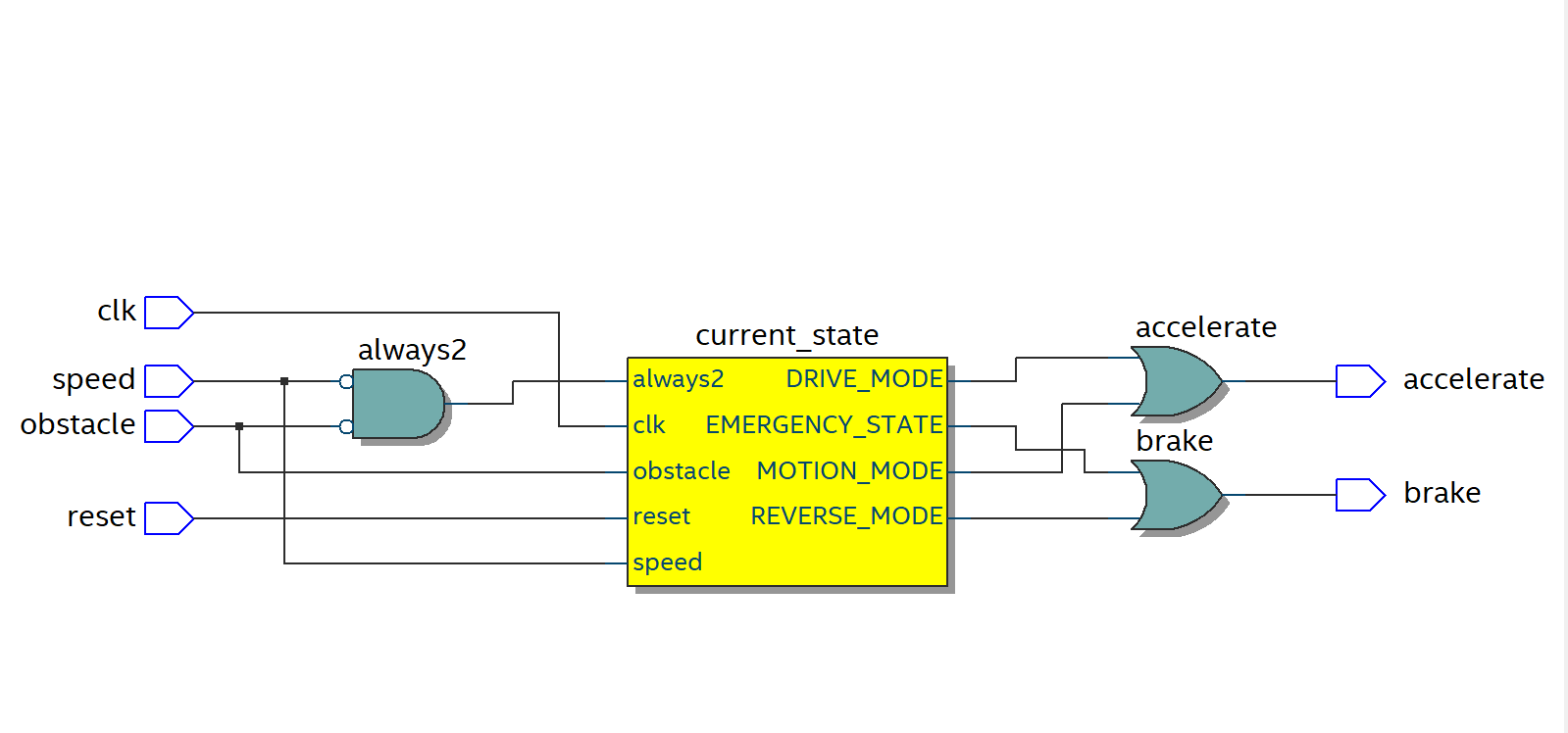




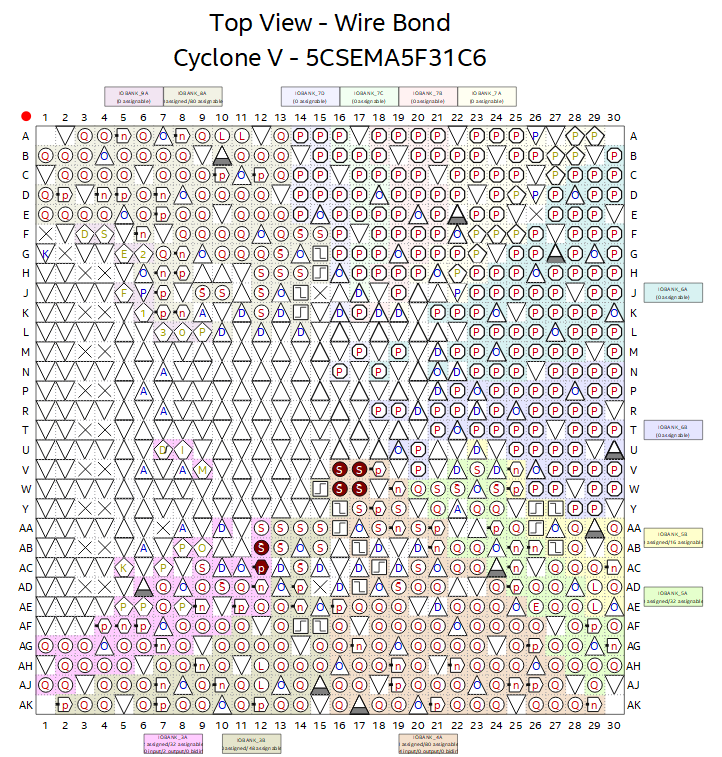
Technology Map Viewer Post Mapping:

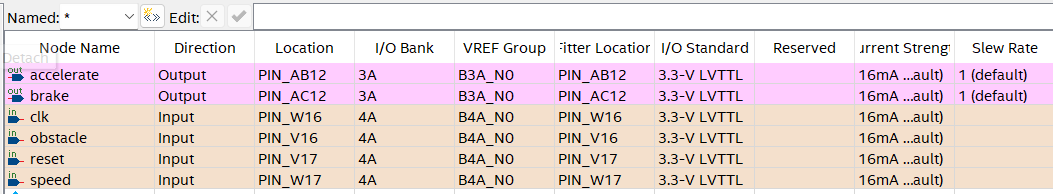


RTL Viewer :



PIN PLANNER:





CHIP PLANNER:

