

AutoChip Tutorial Assignment Report

LLM-Assisted Verilog Generation with Iterative Verification

Tejas Attarde

Course: LLM4ChipDesign

February 15, 2026

Contents

1	Introduction	2
2	AutoChip Framework and Methodology	3
3	Part I(a) Example 1 – Combinational Logic	3
3.1	Model Used	3
3.2	Problem Definition	3
3.3	Formal Boolean Analysis	4
3.4	RTL Implementation	4
3.5	Correctness Argument	4
4	Part I(a) Example 2 – Sequential FSM Design	5
4.1	Model Used	5
4.2	Problem Definition	5
4.3	FSM Formal Specification	5
4.4	State Transition Table	6

4.5	Binary State Encoding	6
4.6	RTL Implementation	6
4.7	Formal Correctness Argument	7
5	Part I(b) – Manual RTL Design	8
5.1	Design Objective	8
5.2	State Encoding	8
5.3	Manual RTL Implementation	8
5.4	Detailed RTL Explanation	9
5.5	Design Rationale	15
5.6	Verification	15
6	Comparative Evaluation	15
7	Timing and Synthesis Considerations	16
8	Lessons Learned	16
9	Conclusion	16

1 Introduction

This report documents the use of the AutoChip framework to generate synthesizable Verilog RTL from natural language specifications. The objective of this assignment was to evaluate the effectiveness of structured large language model (LLM) trajectory refinement in digital hardware design.

Two ChipChat examples were implemented:

- Example 1: Combinational logic module (GPT-4o-mini)
- Example 2: Sequential finite state machine (GPT-4o)

Both designs were validated using Icarus Verilog simulation. Additionally, one design was manually implemented for comparative analysis.

This report presents trajectory evolution, formal Boolean reasoning, FSM modeling, state encoding analysis, correctness arguments, synthesis considerations, and comparative evaluation.

2 AutoChip Framework and Methodology

AutoChip employs iterative refinement:

1. Natural language specification
2. RTL generation
3. Compilation
4. Simulation
5. Feedback incorporation

Let R_i represent the RTL generated at iteration i .

Refinement continues until:

$$\forall t \in \text{Testbench Cycles}, \quad R_i(t) = \text{Expected}(t)$$

This ensures behavioral correctness under provided verification constraints.

3 Part I(a) Example 1 – Combinational Logic

3.1 Model Used

GPT-4o-mini

3.2 Problem Definition

Design a synthesizable combinational module implementing:

$$y = (a \wedge b) \vee (\neg c)$$

3.3 Formal Boolean Analysis

Truth table:

a	b	c	y
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Minimized Boolean expression:

$$y = ab + \bar{c}$$

This matches the specification exactly.

3.4 RTL Implementation

```
assign y = (a & b) | (~c);
```

3.5 Correctness Argument

Since the RTL directly implements the minimized Boolean form and no sequential elements exist, correctness follows directly from Boolean equivalence.

Simulation confirmed full truth table coverage.

4 Part I(a) Example 2 – Sequential FSM Design

4.1 Model Used

GPT-4o

4.2 Problem Definition

Design a sequence detector for pattern:

1011

Output asserts high upon detection.

4.3 FSM Formal Specification

States:

- S_0 : No match
- S_1 : Matched ‘1’
- S_2 : Matched ‘10’
- S_3 : Matched ‘101’
- S_4 : Matched ‘1011’

4.4 State Transition Table

Current	Input	Next
S0	0	S0
S0	1	S1
S1	0	S2
S1	1	S1
S2	0	S0
S2	1	S3
S3	0	S2
S3	1	S4
S4	0	S2
S4	1	S1

Overlap handling ensures:

$$S_4 \xrightarrow{1} S_1$$

4.5 Binary State Encoding

$$S_0 = 000$$

$$S_1 = 001$$

$$S_2 = 010$$

$$S_3 = 011$$

$$S_4 = 100$$

Binary encoding minimizes flip-flop count while maintaining clarity.

4.6 RTL Implementation

```
localparam S0 = 3'b000;
```

```

localparam S1 = 3'b001;
localparam S2 = 3'b010;
localparam S3 = 3'b011;
localparam S4 = 3'b100;

always @(posedge clk) begin
    if (rst)
        state <= S0;
    else
        state <= next_state;
end

```

4.7 Formal Correctness Argument

Define detection condition:

$$x(t-3)x(t-2)x(t-1)x(t) = 1011$$

Invariant:

$$\text{State}(t) = S_k \Rightarrow \text{Longest prefix match length } k$$

By induction:

- Base: Reset initializes S_0
- Step: Transition preserves maximal prefix-suffix alignment

Thus:

$$\text{Detected}(t) = 1 \iff \text{State}(t) = S_4$$

Simulation confirmed:

$$Observed(t) = Expected(t)$$

5 Part I(b) – Manual RTL Design

5.1 Design Objective

Manually implement the same FSM using disciplined synchronous design principles.

5.2 State Encoding

Binary encoding selected for:

- Minimal hardware
- Reduced flip-flop count
- Clean state decoding

5.3 Manual RTL Implementation

```
module sequence_detector_manual (
    input wire clk,
    input wire rst,
    input wire in,
    output reg detected
);

localparam S0 = 3'b000;
localparam S1 = 3'b001;
localparam S2 = 3'b010;
localparam S3 = 3'b011;
localparam S4 = 3'b100;
```

```

reg [2:0] state, next_state;

always @(posedge clk) begin
    if (rst)
        state <= S0;
    else
        state <= next_state;
end

always @(*) begin
    case (state)
        S0: next_state = (in) ? S1 : S0;
        S1: next_state = (in) ? S1 : S2;
        S2: next_state = (in) ? S3 : S0;
        S3: next_state = (in) ? S4 : S2;
        S4: next_state = (in) ? S1 : S2;
        default: next_state = S0;
    endcase
end

always @(*) begin
    detected = (state == S4);
end

endmodule

```

5.4 Detailed RTL Explanation

This section provides a detailed register-transfer-level (RTL) interpretation of the manual FSM implementation. The design is partitioned into three logical components:

1. State register (sequential logic)

2. Next-state combinational logic

3. Output combinational logic

This separation reflects standard synchronous digital design methodology.

1. State Register (Sequential Logic)

```
always @ (posedge clk) begin
    if (rst)
        state <= S0;
    else
        state <= next_state;
end
```

This block describes the memory element of the FSM.

Clock Sensitivity The block is triggered only on the rising edge of `clk`. Therefore:

$$state(t+1) = next_state(t)$$

This ensures that state transitions occur synchronously and eliminates race conditions that would arise in asynchronous logic.

Synchronous Reset When `rst = 1`, the FSM is forced into S_0 .

Advantages of synchronous reset:

- Avoids metastability risks of asynchronous resets
- Simplifies timing analysis
- Aligns with FPGA synthesis best practices

From a hardware perspective, this block synthesizes into:

- Three D flip-flops (for binary encoding)
- Reset gating logic

2. Next-State Logic (Combinational Block)

```
always @(*) begin
    case (state)
        S0: next_state = (in) ? S1 : S0;
        S1: next_state = (in) ? S1 : S2;
        S2: next_state = (in) ? S3 : S0;
        S3: next_state = (in) ? S4 : S2;
        S4: next_state = (in) ? S1 : S2;
    default: next_state = S0;
    endcase
end
```

This block computes:

$$next_state = f(state, in)$$

It is purely combinational because:

- Sensitivity list uses `@(*)`
- No clock edge present
- No non-blocking assignments

Hardware Interpretation This block synthesizes into:

- Combinational logic gates implementing transition equations
- A multiplexer structure controlled by the current state

Each case entry corresponds to a row in the state transition table.

For example:

$$S_3 : next_state = \begin{cases} S_4 & \text{if } in = 1 \\ S_2 & \text{if } in = 0 \end{cases}$$

This encodes the partial match extension property of the pattern detector.

Overlap Handling The critical transition:

$$S_4 \xrightarrow{1} S_1$$

ensures that if the input stream continues with a ‘1’ after detection, the FSM reuses that bit as a prefix for a new match.

Without this transition, overlapping patterns would be missed.

Latch Avoidance Because:

- Every state is explicitly handled
- A default case is provided

The synthesis tool will not infer unintended latches.

3. Output Logic (Moore Architecture)

```
always @(*) begin
    detected = (state == S4);
end
```

The output depends solely on the current state:

$$detected(t) = 1 \iff state(t) = S_4$$

This confirms the design is a Moore machine.

Why Moore? Moore machines provide:

- Glitch-free outputs
- Output stability throughout clock cycle

- Clear separation of state and transition logic

In hardware, this block synthesizes into:

- A comparator between the state register and constant S_4

4. Register-Transfer Semantics

The complete system obeys:

$$state(t+1) = f(state(t), in(t))$$

$$detected(t) = g(state(t))$$

This aligns exactly with classical FSM formalism.

5. Hardware Resource Mapping

Binary encoding uses:

- 3 flip-flops for state storage
- Combinational gates for transition logic
- Comparator logic for detection

Total hardware cost is modest and suitable for FPGA or ASIC implementation.

6. Timing Considerations

Critical path:

$$T_{clk} \geq T_{CQ} + T_{logic} + T_{setup}$$

Where:

- T_{CQ} = Clock-to-Q delay of flip-flop

- T_{logic} = Delay through next-state combinational logic
- T_{setup} = Setup time of state register

Because transition logic is shallow (single case structure), T_{logic} remains small.

7. Formal Correctness Interpretation

Define pattern:

$$P = 1011$$

At time t , the FSM state encodes:

k = length of longest prefix of P that matches suffix of input up to t

Thus:

$$state(t) = S_k$$

When $k = 4$, full match occurs and output asserts.

This proves correctness by construction.

8. Why This RTL is Synthesizable

The design satisfies synthesis constraints:

- No blocking assignments in sequential logic
- No incomplete case statements
- No combinational feedback loops
- Single clock domain
- Deterministic reset state

Therefore, it maps cleanly to hardware without ambiguity.

5.5 Design Rationale

The design uses:

- Moore architecture
- Synchronous reset
- Non-blocking sequential assignments
- Complete state coverage

The design is fully synthesizable.

5.6 Verification

```
iverilog -o sequence_detector_tb sequence_detector_manual.v tb.v  
vvp sequence_detector_tb
```

All test cases passed.

6 Comparative Evaluation

GPT-4o-mini converged quickly for combinational logic.

GPT-4o required additional refinement for sequential logic due to:

- Reset handling
- Overlap correctness
- State transition completeness

Manual implementation required deeper reasoning but achieved correctness deterministically.

7 Timing and Synthesis Considerations

Sequential timing constraint:

$$T_{clk} \geq T_{CQ} + T_{logic} + T_{setup}$$

Combinational path delay:

$$T_{comb} = T_{AND} + T_{OR}$$

FSM logic depth remains bounded and synthesizable.

8 Lessons Learned

- Iterative refinement is critical for correctness.
- Larger models handle sequential abstraction better.
- Simulation-driven verification prevents silent logic errors.
- Manual design reinforces architectural discipline.

9 Conclusion

AutoChip demonstrates effective integration of LLM-assisted RTL generation with simulation-based verification.

Combinational logic tasks are efficiently handled by lightweight models, while FSM-based designs benefit from higher-capacity reasoning models.

The structured refinement workflow significantly improves convergence reliability and bridges natural language specifications to correct, synthesizable RTL.