Logic Design Lab 3: Pattern Matching Report

110062219

May 30, 2022

1 Design of Finite State Machine

We are to carry out a patter matching module for a certain sequence of binary sequences in the Lab 3. The pattern is described like a regular expression: `010(101)+11`, which is equivalent to `010101(101)*11`.

On top of the regex, we could derive the **Nondeterministic Finite Automata (NFA)** and **Deterministic Finite Automata (DFA)**. By reducing redundant states, we could obtain the state diagram (Figure 1).

The forward edges could be constructed by the definition. Since we accept none or more `101` in the middle, we have a loop from the last `1` to the first `1`. The backward edges are far more difficult. When we fail to match, we should go back to the LPS, i.e. the longest prefix which is also a suffix of current state. I'll explain more details later.

State	0	1	Output	LPS when unmatched
A0	B1	A0	0	`""`
B1	B1	C2	0	`"0"`
C2	D3	A0	0	`""`
D3	B1	E4	0	`"0"`
E4	F5	A0	0	`""`
F5	B1	G6	0	`"0"`
G6	F5	H7	0	`"01010"`
H7	J9	I8	0	N/A
I8	B1	A0	1	`"0"`
J9	B1	K10	0	`"0"`
K10	D3	H7	0	`"010"`

Table 1: State Table[1]

[1]The states are named by a letter and a digit so that we could refer them by either.

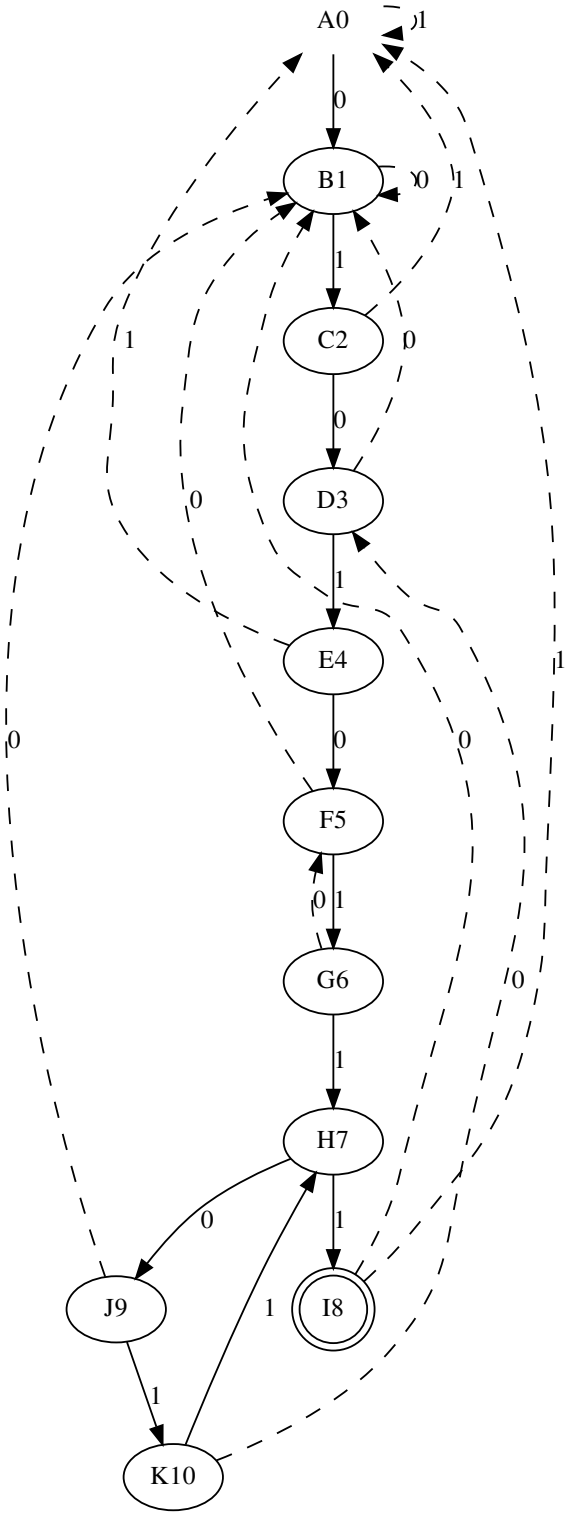

Figure 1: State Diagram

2 Implementation in `verilog`

The 11 states are declared as `parameter`s. We need two 4-bit registers to record current state and next state.

```
parameter A = 4'd0, B = 4'd1, C = 4'd2, D = 4'd3,
    E = 4'd4, F = 4'd5, G = 4'd6, H = 4'd7, I = 4'
    d8, J = 4'd9, K = 4'd10;
reg[3 : 0] state, next;
```

Our **Moore Machine** consists of 3 parts: the next state logic (combinational, synchronous circuit), the output logic and the current state register (sequential, asynchronous circuit).

2.1 Next State Logic

I made use of `case`-block and ternary conditional operator so that my codes are quite concise and make sense.

```
always @(*)
    if (reset)
        next = A;
    else
        case (state)
            A: next = data ? A : B;
            B: next = data ? C : B;
            C: next = data ? A : D;
            D: next = data ? E : B;
            E: next = data ? A : F;
            F: next = data ? G : B;
            G: next = data ? H : F;
            H: next = data ? I : J;
            I: next = data ? A : B;
            J: next = data ? K : B;
            K: next = data ? H : D;
        endcase
```

2.2 Output Logic

There's no need to declare the `flag` as a register and utilize an `always`-block. It's enough to assign it as a wire with simple logic expression.

```
assign flag = !reset && state == I;
```

2.3 Current State Register

The transition of the states takes place when the positive clock edge occurs. If `reset`, the state goes back to A.

```
always @(posedge clk)
    if (reset)
        state <= A;
    else
        state <= next;
```

3 Problems

This time, I also encountered several problems without exception to previous labs.

3.1 Accept Input Prematurely

Initially, I designed the **FSM** as **Mealy Machine** and didn't have the state I. Instead, I directly connected state H to state A with a 1/1 edge and to state B with a 0/0 edge. The output `flag` was assigned to `!reset && state == H && data` then.

Nonetheless, my module failed to the test case 0 and mismatched the sequence like `0101011`. I took a look at the wave shown in Figure 2, realizing that the problem is due to time difference.

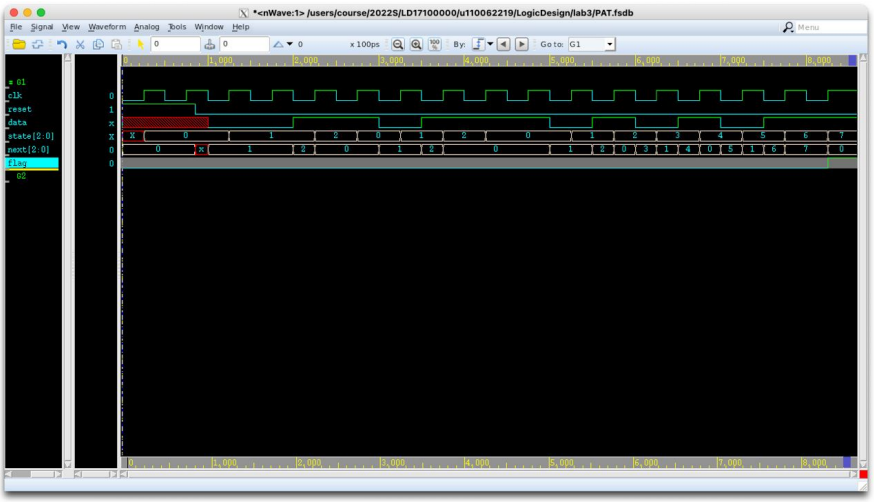

Figure 2: Wrong Wave

When the current state was H, the output is true if the next input is 1. But in my original design, the output would be true right at state H since it catch the current input.

As a consequence, I added the state I. Thus, the output depends on current state only and never triggers too early again. That is, my **FSM** became **Moore Machine**.

3.2 Wrong Backward Edges & State Reduction

I didn't come up with the whole state diagram at a time. In fact, I even just found some mistakes while I was writing this report, one of which is I connected state E to state C when unmatched. Interesting enough, these didn't affect that I passed all 3 test cases of both simulation and synthesis.

The longest time I struggled on was the backward edge of state G. At first I connect that to state B but failed at line 32 of test case 1. I checked again and connect it to state D. However, I failed again at line 75 of test case 1. So I pondered for a while and connect it to state F. Subsequently, I passed the remaining test cases of both simulation and synthesis.

Though, I'm really afraid that I still have some mistake. I discussed with my friend and found another fatal flaw in my design. I used to interpret the regex directly and connected state H back to state F if the input is false to form a loop. Nevertheless, there were a problem hidden. When we were at the `0` in the `101` (state G) and failed to match, the LPS was different depending on whether it was our first time to visit. That's why I used to connect it to state D in the previous paragraph. Thus, I reinterpreted the regex as `010101(101)*11`, adding two more states to separate the first and further visits. It seemed that I reduced these two state in the very beginning, regarding J, K as equivalent to F, G respectively.

4 Reflections & Questions

This lab is quite intriguing for me. I wasn't very concentrated on the courses until Prof. Mak lectured on the example 5 of sequential circuit, which was pattern recognizer. It raised my interest immediately since I learnt some algorithm about string matching such as **Knuth-Morris-Pratt (KMP)** and **Gusfield (Z)** before.

The crucial key to pattern matching is to construct those backward edges when unmatched. This has a lot to do with Failure Function in **KMP**. So how to find it? We need to find the LPS, i.e. the longest proper prefix, which is also a suffix of current state, which is what **Z** does.

I also have some questions to ask TAs:

- Could this lab be done in **Mealy Machine**? If so, how to avoid the situation I ran into in **Section 3.1**?
- In what circumstance one module would pass the simulation but fail the synthesis? I don't know the meaning to use the Digital Vision.
- It seems that `verilog` is such a high-level **HDL** that we needn't deal with those bothersome latches or flip-flops at all. Are they, along with muxs, decoders and so forth, really important?
- Could you release more hard test cases? I passed all of them while having two flaws.
- Could we have our second midterm paper back? Will the final exam be taken physically?
- Will the professor kindly "curve" our grades? I didn't do so well in the two midterms though having such fun time doing all three labs. Nonetheless, I'm quite anxious about my grade.