Abstract

Analysis of Data throughput on a Field Programmable Gate Array FPGA with special interest in searching data input streams for key patterns via automata / finite state machines.

FPGA's have great potential to speed up data throughput relative to a standard processor. Standard processors execute instructions sequentially while FPGA's execute operations in parallel. Depending on the algorithm, it is possible to see orders of magnitude higher data throughputs in FPGAs.

As time passes, data is becoming more unstructured and FPGA's are playing larger roles in the analysis of this unstructured data. Examples of unstructured data are:

- XML
- images
- videos
- emails

Regular expression matching is utilized in many systems, from simple text search in user applications to parsing network data for security applications. Popular intrusion systems such as Snort [3] and Bro [4] utilize regular expressions to perform deep packet inspection to identify security threats.

The key question to answer is, what types of automata / finite state machines can be implemented on an FPGA and how fast can these algorithms process data on an FPGA? Also of interest, is how circuit length / complexity affects output latency.

Background

String matching algorithms are utilized in many applications such as network security, web applications, bioinformatics, word processors, plagiarism detectors, and databases [8]. Many such applications utilize regular expressions for string matching due to the ease of representing complex string patterns.

Regular expressions are regular languages constructed with characters from a fixed alphabet. Each regular expression consists of regular characters as well as metacharacters. For example, in the regular expression “xy.”, “x” and “y” will only match the characters “x” and “y” while the metacharacter “.” will match any character. Therefore, “xyz” and “xyx” are both matches to the regular expression “xy.”.

A regular language offers three operators on character classes:

- Concatenation – represents a sequence of character classes (ex: abcde)
- Union – represents a choice between character classes (ex: ab | cd)
- Kleene closure – represents none or any number of a character class (ex: a* => “” or “aaaaa…”)

Other commonly used operators (+, ?, [ ], and { }) can be constructed utilizing proper arrangement of these basic operators. Because regular languages are accepted by finite state automata, a regular expression-matching engine can be implemented as a finite state machine (FSM) – either non-deterministic or deterministic. In this project, non-deterministic state machines are the focus.

As mentioned, FSM's are computing models that recognize and accept regular languages. FSM's are traditionally represented as a 5-tuple (Q, ∑, q, F, δ):

- Q – Set of states
- ∑ - Set of input symbols
- Q – Start state
- F – Final state
- δ – Transition function

In FSM's, transition from one state to another is determined by the transition function. Given an input character, the transition function will output the states that can be transitioned to based on that input character. In a non-deterministic FSM, one input character can produces multiple states as potential transitions.

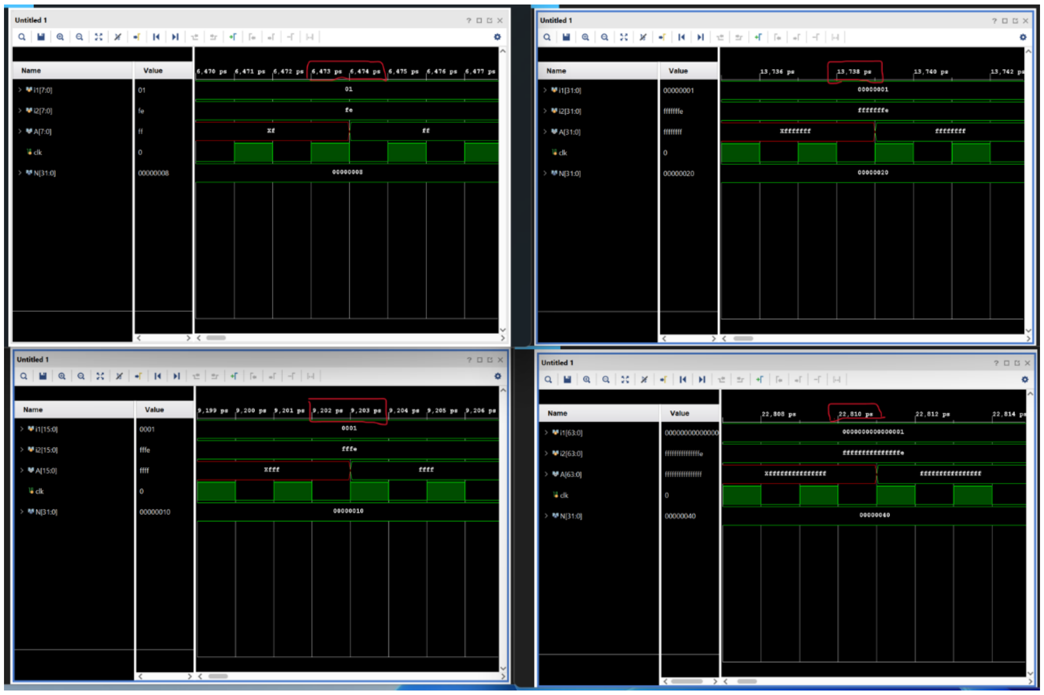
In a non-deterministic regular expression-matching engine (REME), each input character is sent to every state in the state machine and each input character can lead to multiple state transitions. As a result, a non-deterministic REME can have one or more active states at any given time. FPGA's offer high circuit parallelism which is highly compatible with non-deterministic state machine design due to the parallel state transitions [2]. Several challenges exist in working with REME's and hardware. One challenge is processing many patterns in parallel. State machines require logic, registers, and block memory to implement on hardware and more REME's require more on-chip resources. This makes design optimization very important. Another challenge is maximizing throughput. Maximizing throughput is dependent on efficient design, regular expression size, and maximizing the clock rate on the FPGA. Regular expression size increases the resources required to implement the corresponding state machine on hardware.

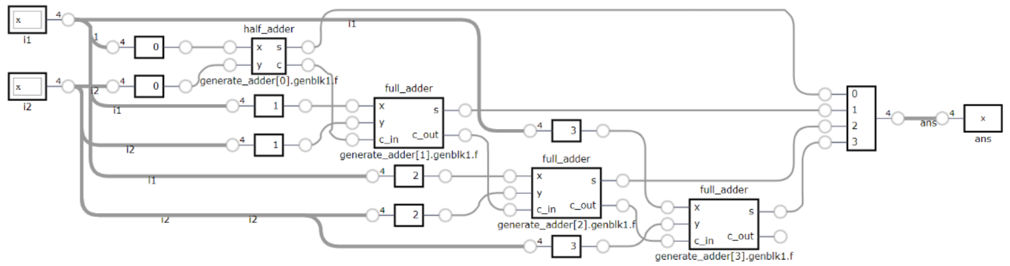
In this project, a method for converting a regular expression into an efficient non-deterministic state machine is presented. Thompson's construction is utilized to construct an NFA from a regular expression in linear time. Utilizing the methodology presented in Prasanna et al [2], NFA states are converted to a simple, highly modular structure, and block RAM is utilized for character class matching. This produces a final system design that makes efficient use of hardware resources and allows for high clock rates to increase data throughput.

<u>Methods and Results</u>

How does circuit complexity affect result latency?

In order to determine how circuit complexity affects result latency (time from initial input to result) a program to generate an N-bit adder was utilized. Given a parameter N, a linear N-bit adder is generated that utilizes one half-adder and N – 1 full-adders. As N increases the circuit length increases. As seen in the figure below, tests were run for 8, 16, 32, and 64 bit adders. As the bits added were doubled, the time to obtain a result was increased as well. The 8-bit adder time to result was 6473ps, the 16-bit adder time to result was 9202ps, the 32-bit time to result was 13738ps, and the 64-bit time to result was 22810ps. The tests were carried out using Xilinx Vivado [7]. Circuit length / complexity does affect latency as is indicated by the above results although it is worth noting that optimizations can be made via pipelining and compiler optimization to improve the above results.





i1
i2
half_adder
generate_adder[0].genblk1.f
full_adder
generate_adder[1].genblk1.f
full_adder
generate_adder[2].genblk1.f
full_adder
generate_adder[3].genblk1.f
ans

Thompson Construction

In order to convert a regular expression into a modular circuit, the first step was to build a NFA from a string representation of a regular expression. Thompson's construction was utilized for this purpose. The construction was done via a linear scan of the regular expression string. Two stacks were used; one to manage operator precedence and another to build the NFA. To expand the NFA, a fragment structure was used that contains a pointer to the starting state, and a list of all outgoing unconnected transitions. For each operator, the fragment is expanded. This process is carried out until the end of the regular expression string is reached. Overall, the process takes linear time, dependent on the length of the input string.

Operations

- **Character =>** Create new state; Store transition value in State (label in the figure below); Convert to fragment; push to stack.

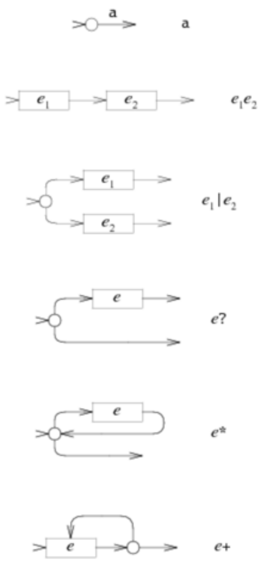
- **Concatenation =>** Pop two fragments from the stack; Connect out of first fragment to start of second fragment; Push new connected fragment onto stack.

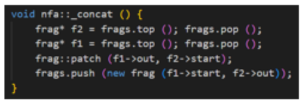
- **Union =>** Pop two fragments; Create new state with “split” label (indicates union transition), Connect out1 to first fragment start; Connect out2 to second fragment start; Add fragment transitions to unconnected transition list; Push new fragment onto stack.

- **Option =>** Pop one fragment; Create new state with “split” label, Connect out1 to start of fragment, Add fragment out and out2 to list of unconnected transitions, Push new fragment onto stack.

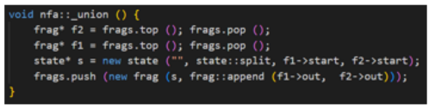
- **Star =>** Pop one fragment; Create new state with “split” label, Connect out1 to start of fragment; Connect fragment out transitions back to new state; Add out2 to list of unconnected transitions; Push new fragment onto stack.

- **Plus =>** Pop one fragment; Create new state with “split” label; Connect out1 to start of fragment; Connect fragment out to new state; Add out2 to the list of unconnected transitions; Push new fragment onto stack.

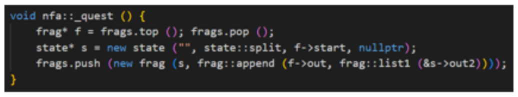




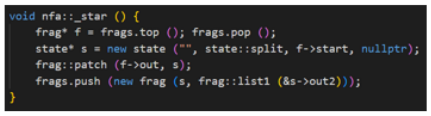
```
void nfa::_concat () {
    frag* f2 = frags.top (); frags.pop ();
    frag* f1 = frags.top (); frags.pop ();
    frag::patch (f1->out, f2->start);
    frags.push (new frag (f1->start, f2->out));
}
```



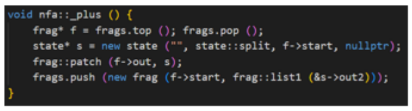
```
void nfa::_union () {
    frag* f2 = frags.top (); frags.pop ();
    frag* f1 = frags.top (); frags.pop ();
    state* s = new state ("", state::split, f1->start, f2->start);
    frags.push (new frag (s, frag::append (f1->out,  f2->out)));
}
```



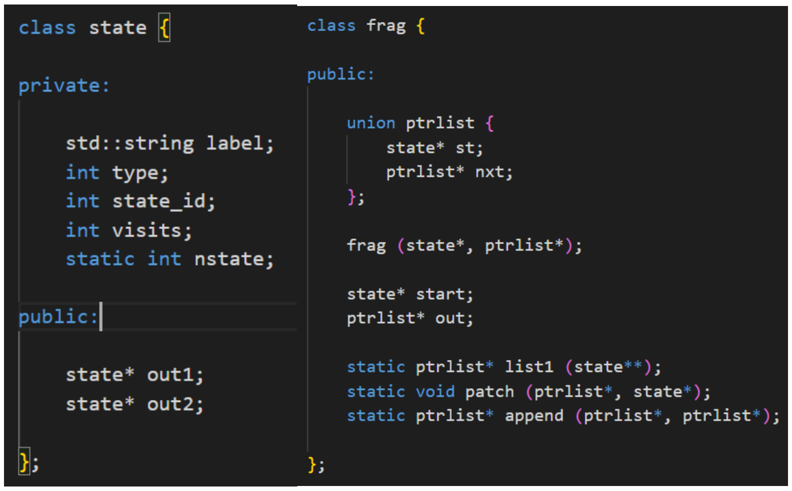
```
void nfa::_quest () {
    frag* f = frags.top (); frags.pop ();
    state* s = new state ("", state::split, f->start, nullptr);
    frags.push (new frag (s, frag::append (f->out, frag::list1 (&s->out2))));
}
```



```
void nfa::_star () {
    frag* f = frags.top (); frags.pop ();
    state* s = new state ("", state::split, f->start, nullptr);
    frag::patch (f->out, s);
    frags.push (new frag (s, frag::list1 (&s->out2)));
}
```



```
void nfa::_plus () {
    frag* f = frags.top (); frags.pop ();
    state* s = new state ("", state::split, f->start, nullptr);
    frag::patch (f->out, s);
    frags.push (new frag (f->start, frag::list1 (&s->out2)));
}
```



```
class state {

private:

    std::string label;
    int type;
    int state_id;
    int visits;
    static int nstate;

public:

    state* out1;
    state* out2;

};
```

```
class frag {

public:

    union ptrlist {
        state* st;
        ptrlist* nxt;
    };

    frag (state*, ptrlist*);

    state* start;
    ptrlist* out;

    static ptrlist* list1 (state**);
    static void patch (ptrlist*, state*);
    static ptrlist* append (ptrlist*, ptrlist*);

};
```

Module Design

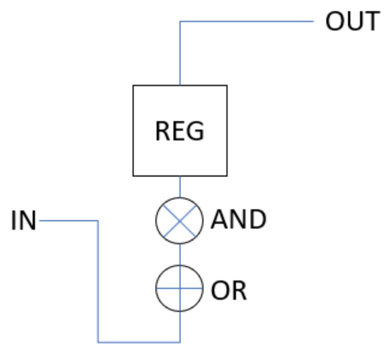
Module

In the construction of the HDL-based state machine, using the implementation of Prasanna et al [2] as a starting point, the module in the figure below represents a single state. These modules are pieced together to create a state-machine representing a given regular expression. The contents of each module are as follows:

- **OR-Gate:** The values associated with all incoming states (1=active, 0-inactive) are routed to the OR gate. If the module is the starting state, the input source is set to 1 to ensure continuous processing of the input.

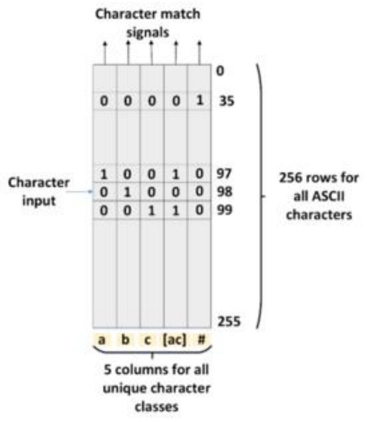
- **AND-Gate:** The AND gate will receive the output of the OR gate and the block ram (BRAM, discussed in next section). The BRAM provides the value to determine if the current input character matches the transition needed to activate a particular state.

- **REGISTER:** The register receives the output of the AND gate. The value produced by the register will be one if an incoming state transition is one and the current input character allows transition to the state. This indicates the incoming BRAM value is one. The register is implemented as a flip-flop which allows each state to have access to the previous clock cycles state values.



BRAM

The BRAM implementation was done in a similar fashion to Prasanna et al [2] as well. The BRAM is implemented as a 2x2 matrix where each column represents a character class or state, and each row represents the decimal value of each ASCII character (0 – 255). Each character class column will store a 1 in the row of the ASCII char that allows transition to that state. Each column signal is routed to the corresponding module of the character class, as seen in the figure below. This BRAM implementation allows for both conservation of hardware resources and efficient character matching.

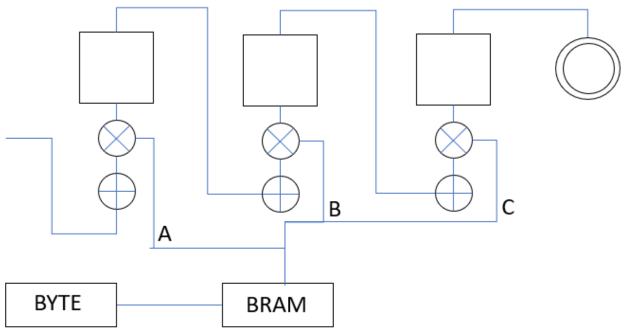


REGEX Example

In the figure below, an example regular expression ("ABC") is provided utilizing the modular structure outlined above. In the case that matching string "ABC" is the input:

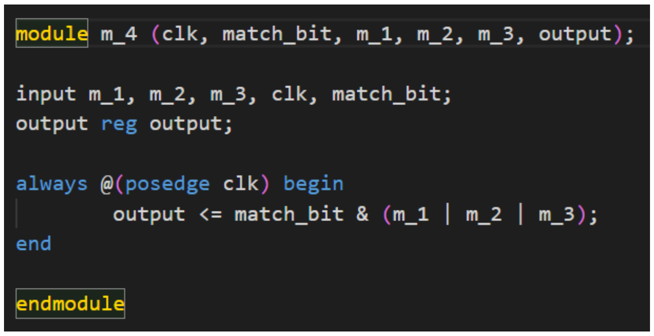
- The initial input coming into the first module is set to one.
- On the first clock cycle A is read into the BRAM. From left to right, the first module will receive signal 1, and the next two modules will receive signal 0 (100).

- Module 1 will now be active due to input and BRAM signals both being 1.
- On the next clock cycle B is read into the BRAM. From left to right, the signals sent from the BRAM will be 010. The register in module 1 will be sending a signal of 1 to the middle module (from the previous clock cycle match).
- Module 2 will now be active due to input and BRAM signals both being 1.
- On the next clock cycle C is read into the BRAM. From left to right, the signals sent from the BRAM will be 001. The register in module 2 will be sending a signal of 1 to the right module (from the previous clock cycle match).
- Module 3 will now be active due to input and BRAM signals both being 1. Module 3 will send a signal of 1 to indicate the string "ABC" matches.



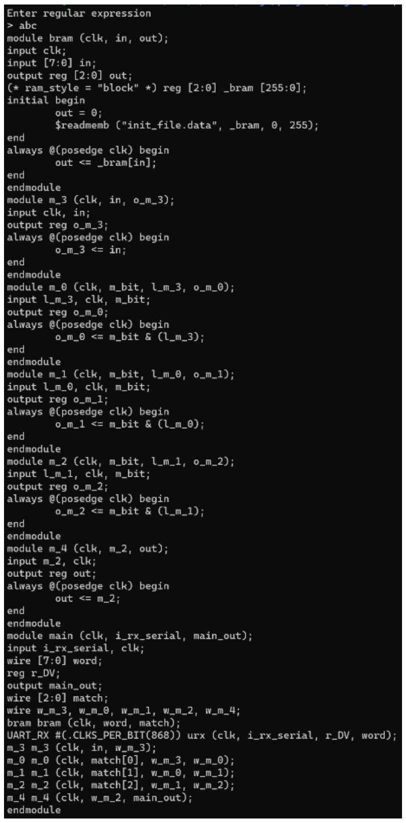
Hardware Implementation

Once built, a program to convert each state in the NFA into a module (described above) was utilized. The program both creates the modules and makes the necessary connections between the modules that are required to make the circuit functional. As seen in the example HDL below, each HDL module contains an or-gate along with an and-gate. By utilizing the system clock, the output register acts as a flip-flop that stores the previous clock cycle's state value.

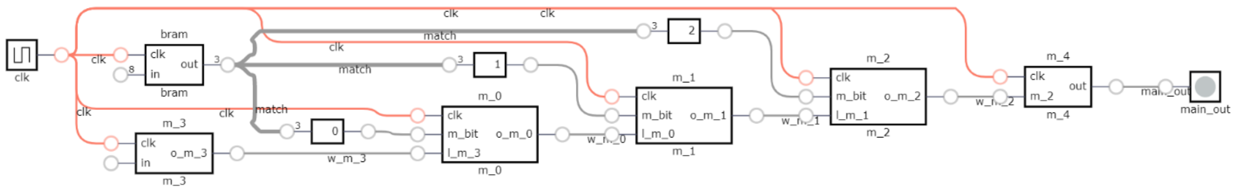


```
module m_4 (clk, match_bit, m_1, m_2, m_3, output);

input m_1, m_2, m_3, clk, match_bit;
output reg output;

always @(posedge clk) begin
        output <= match_bit & (m_1 | m_2 | m_3);
end

endmodule
```

Below is an example of the program converting the regular expression “abc” to an HDL representation of the modular state machine described above. The top figure provides the execution result of the program, and the bottom image provides the circuit created as a result of the HDL.



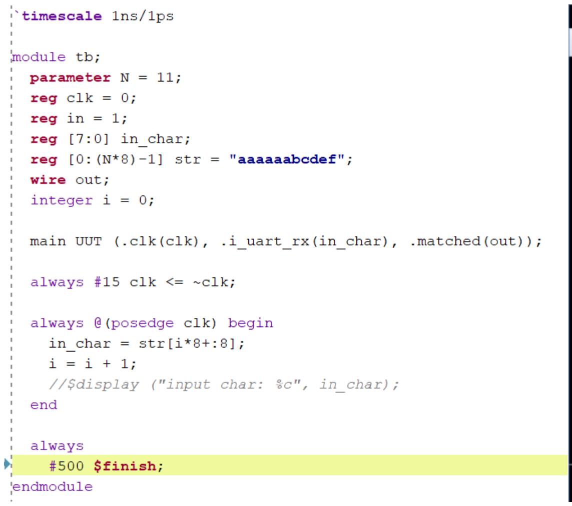
```
Enter regular expression
> abc
module bram (clk, in, out);
input clk;
input [7:0] in;
output reg [2:0] out;
(* ram_style = "block" *) reg [2:0] _bram [255:0];
initial begin
        out = 0;
        $readmemb ("init_file.data", _bram, 0, 255);
end
always @(posedge clk) begin
        out <= _bram[in];
end
endmodule
module m_3 (clk, in, o_m_3);
input clk, in;
output reg o_m_3;
always @(posedge clk) begin
        o_m_3 <= in;
end
endmodule
module m_0 (clk, m_bit, l_m_3, o_m_0);
input l_m_3, clk, m_bit;
output reg o_m_0;
always @(posedge clk) begin
        o_m_0 <= m_bit & (l_m_3);
end
endmodule
module m_1 (clk, m_bit, l_m_0, o_m_1);
input l_m_0, clk, m_bit;
output reg o_m_1;
always @(posedge clk) begin
        o_m_1 <= m_bit & (l_m_0);
end
endmodule
module m_2 (clk, m_bit, l_m_1, o_m_2);
input l_m_1, clk, m_bit;
output reg o_m_2;
always @(posedge clk) begin
        o_m_2 <= m_bit & (l_m_1);
end
endmodule
module m_4 (clk, m_2, out);
input m_2, clk;
output reg out;
always @(posedge clk) begin
        out <= m_2;
end
endmodule
module main (clk, i_rx_serial, main_out);
input i_rx_serial, clk;
wire [7:0] word;
reg r_DV;
output main_out;
wire [2:0] match;
wire w_m_3, w_m_0, w_m_1, w_m_2, w_m_4;
bram bram (clk, word, match);
UART_RX #(.CLKS_PER_BIT(868)) urx (clk, i_rx_serial, r_DV, word);
m_3 m_3 (clk, in, w_m_3);
m_0 m_0 (clk, match[0], w_m_3, w_m_0);
m_1 m_1 (clk, match[1], w_m_0, w_m_1);
m_2 m_2 (clk, match[2], w_m_1, w_m_2);
m_4 m_4 (clk, w_m_2, main_out);
endmodule
```



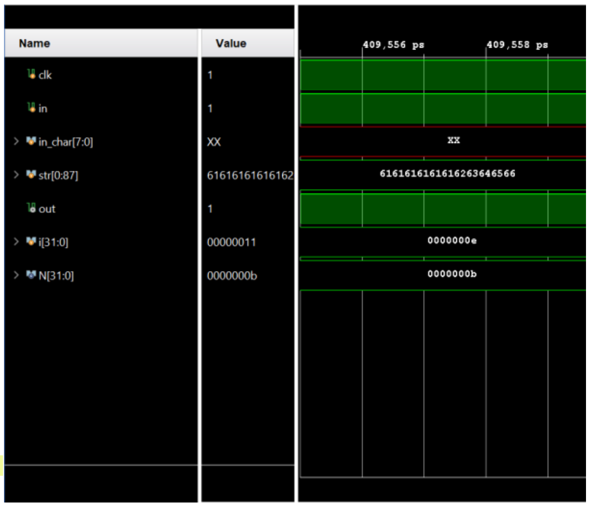
<u>Future Work</u>

Current State

The program produces correct results in simulation and can successfully be converted to a bit stream and loaded onto hardware. Below is a successful test bench for a string matching the regular expression ab(c|d)*ef. The out signal high (1) indicates that the state machine has found a match in the string "aaaaaabcdef".

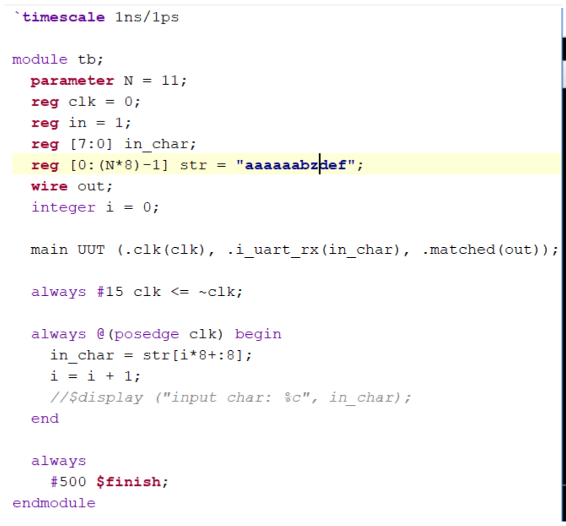


```
`timescale 1ns/1ps

module tb;
  parameter N = 11;
  reg clk = 0;
  reg in = 1;
  reg [7:0] in_char;
  reg [0:(N*8)-1] str = "aaaaaabcdef";
  wire out;
  integer i = 0;

  main UUT (.clk(clk), .i_uart_rx(in_char), .matched(out));

  always #15 clk <= ~clk;

  always @(posedge clk) begin
    in_char = str[i*8+:8];
    i = i + 1;
    //$display ("input char: %c", in_char);
  end

  always
    #500 $finish;
endmodule
```

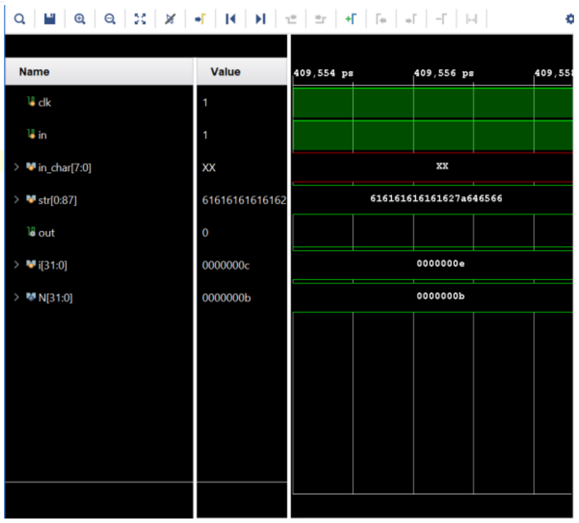


Name	Value
clk	1
in	1
in_char[7:0]	XX
str[0:87]	61616161616162
out	1
i[31:0]	00000011
N[31:0]	0000000b

Below is an unsuccessful test bench for a string not matching the regular expression ab(c|d)*ef. The out signal low (0) indicates that the state machine has not found a match in the string "aaaaaabzdef".



```
`timescale 1ns/1ps

module tb;
  parameter N = 11;
  reg clk = 0;
  reg in = 1;
  reg [7:0] in_char;
  reg [0:(N*8)-1] str = "aaaaaabzdef";
  wire out;
  integer i = 0;

  main UUT (.clk(clk), .i_uart_rx(in_char), .matched(out));

  always #15 clk <= ~clk;

  always @(posedge clk) begin
    in_char = str[i*8+:8];
    i = i + 1;
    //$display ("input char: %c", in_char);
  end

  always
    #500 $finish;
endmodule
```



Name	Value
clk	1
in	1
in_char[7:0]	XX
str[0:87]	61616161616162
out	0
i[31:0]	0000000c
N[31:0]	0000000b

Next Steps

Overall, the methodology is in place and the algorithm produces correct results. The HDL produced by this program can successfully be loaded onto an FPGA. The next steps are to implement an interface capable of streaming data into the FPGA. Currently, a simple UART interface has been developed and works correctly in simulation. The UART interface is in the process of being tested on hardware. Although, the best option for streaming data into the FPGA is ethernet. The most important next step will be to implement an ethernet interface to the state machine described in this project. Xilinx offers proprietary IP that implements an ethernet interface although the IP is bulky and contains features that are not necessary for the scope of this project. The best option will be to implement a custom ethernet interface. Once sufficient background knowledge is gathered, and a specific feature set is narrowed down, this interface will be developed.

References

1. Rachana S et al (2020). Design and Delay Analysis of 256 bit adders. Design and Delay Analysis of Various 256-Bit Adders using Verilog (ijert.org)
2. Prasanna et al (2008). Compact Architecture for High-Throughput Regular Expression Matching on FPGA. https://www.eecg.utoronto.ca/~jzhu/csc467/readings/NFA-hardware.pdf
3. Snort.org - http://www.snort.org/
4. Bro Intrusion Detection System - http://bro-ids.org/
5. Prasanna et al (2001). Fast Regular Expression Matching Using FPGAs. https://www.researchgate.net/publication/4139271_Fast_Regular_Expression_Matching_Using_FPGAs
6. Vivado - https://www.xilinx.com/products/design-tools/vivado.html
7. Sert et al (2018). NFA-Based Regular Expression Matching on FPGAs