Final Report of Autonoma Processor (AP) Project

Juan Garza

December 11, 2023

1 Abstract

This project uses custom RTL logic on FPGA to accelerate Autonoma processing which matches regular expression matches in text data. A flow using python scripting was used to create to create RTL logic based on regular expressions desired for matching. This flow allows for simulation based results from RTL logic and an easy way to add into FPGA to test on hardware and get results to PC. Raw text data is stored as bytes into BRAM on FPGA and processed into a custom AP block. The Verilog RTL has no dependencies on which FPGA it can be implemented on making this solution easily interchangeable and accessible. The end goal being to show proof of concept of Autonoma processing logic and its speeddup.

2 Introduction

The goal of this project is to create a hardware accelerator for fast pattern detection on a stream of data using autonoma processing. This design will be implemented on an FPGA and the goal is to have low latency, high throughput of data which will be compared to current designs on mainline processors with same compute resources as FPGA.

The skills used in this project involve scripting and automation of the design process with RTL logic, verification and making code very general to apply to different processes. This project will simulate the chip design process. Covering design of RTL, automation of design process when applicable, verification, and implementation on using real word data.

Coding the logic of the autonoma processor was relatively simple compared to complexities of data handling and configuration of the FPGA fabric. Most design effort for this design was getting data handling to work in hardware and make hardware results match simulation.

3 Methods

Background and Experiment Design

Regular expressions are ways to represent character classes or strings that meet a certain set of criteria. DNA sequence patterns are a prime example case. A few other examples:

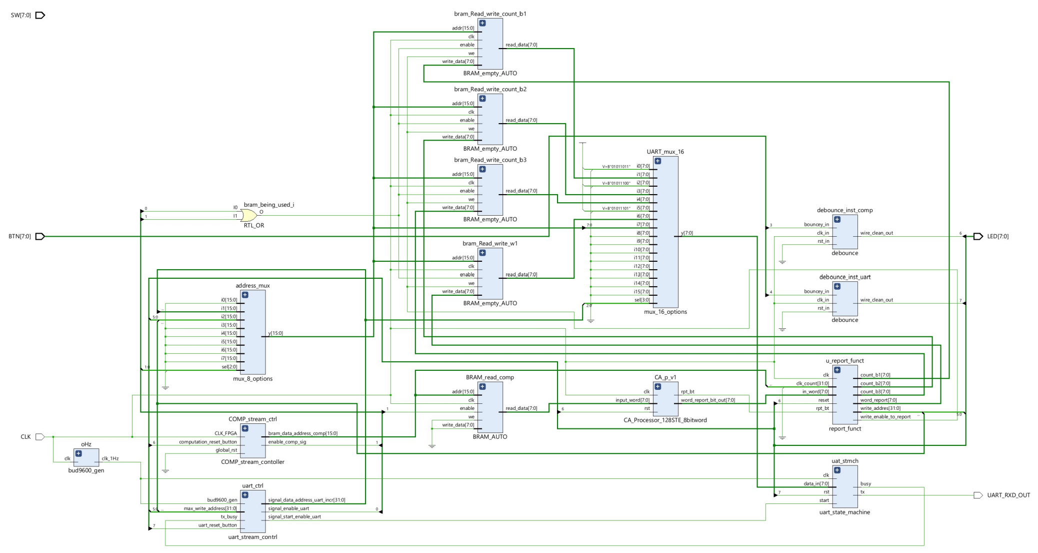
a.*b → characters that have an a followed by a final b at some point, ex: "aXXXXXXXXbXXb"
A|c → string with A or c → "AXXXX" OR "XcXXXAXXXXc" Regex that accepts: has "abd" or "ac" → abd|ac

This can then be presented in a graph when you iterate through the string with state transition elements (STEs). Where the STE is a node with one or more vertices which can be thought of where the node with an active and not active state with activation character(s) where if the STE is "active" and has "activation character" sense from word input, then the STEs it is pointing to in its vertices will be "active" in the next word character and this process continues.

FPGA Implementation

All RTL and processing is done on Nexys video and Nexys DDR4 FPGAs which are programmed in Vivado tool suite. The final inputs into FPGA are the constraint file (provided by Nexys), and the Verilog RTL files which are automatically generated through scripts. This makes the design general and easy to employ on other FPGAs. A template project is placed on the git-hub where the Verilog RTL are directly from the scripts. Each Verilog file generated by scripts correspond to Verilog file in Vivado. This project used Vivado Version 2022.1 however other versions are easy to use.

Figure 1: RTL on FPGA



Design Processor Logic

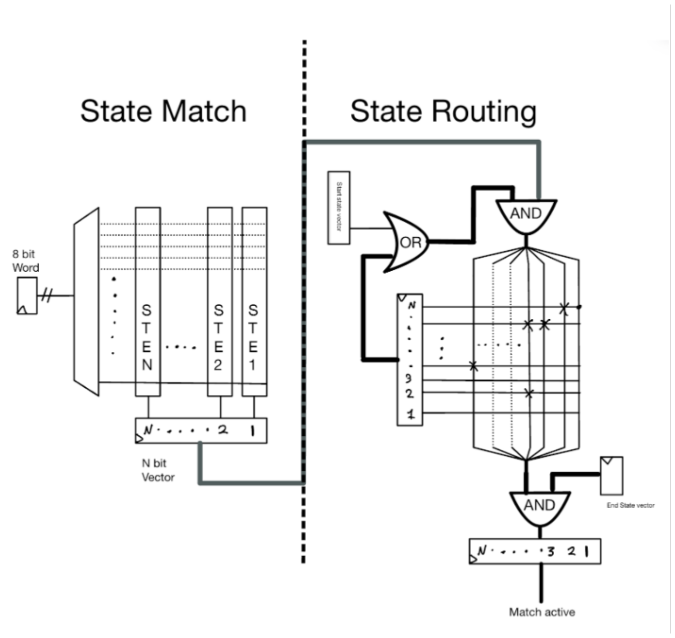
The entire design has three main modules design components the Autonoma processor logic, the memory / data handling logic, and match report logic. The memory / data handling is responsible for string data stored in BRAM of FPGA fabric and how data is written and read memory in FPGA. The Autonoma Processor (AP) design refers to the graph NFA implemented on hardware which actually processes the data from memory and reports when matches occur. The match report logic refers to how the FPGA captures information when a match occurs recording clock cycle, word activated and how that report data is streamed back to PC by UART.

Autonoma Processor logic

The AP RTL Logic is generated from a python script taking a regular expression as input. This code first generates the graph for the regular expression as an ANML NFA graph. This gives the N amount of STE elements the AP creates which is equal to the $\lceil log_2(STE_n) \rceil$. The AP logic is pipelined into two main stages, Match stage and route stage. This architecture has been modeled after the *Cache Automaton*[1] paper listed below. This paper gives more background and fundamental understanding for this project.

The match state is checking for which STE activation characters are at current word, and the route stage then checks which STE needs to activate and route to next STE active bit for next cycle. There are $2 + 2 * STE_{needed}$ main parameter types: one start STE vector, one end STE vector,

Figure 2: AP logic



and for each STE (N total STEs): N STE activation characters, and N parameters denoting which STEs the corresponding STE maps to.

Memory Handling

This project uses Verilog describing the memory instead of manually loading the BRAM with a file. The Vivado tool, during the synthesis and implementation will configure the BRAM block to match the bahvior of logic. The memory RTL is created by python script given the text data which will encode the each address as one byte describing each character.

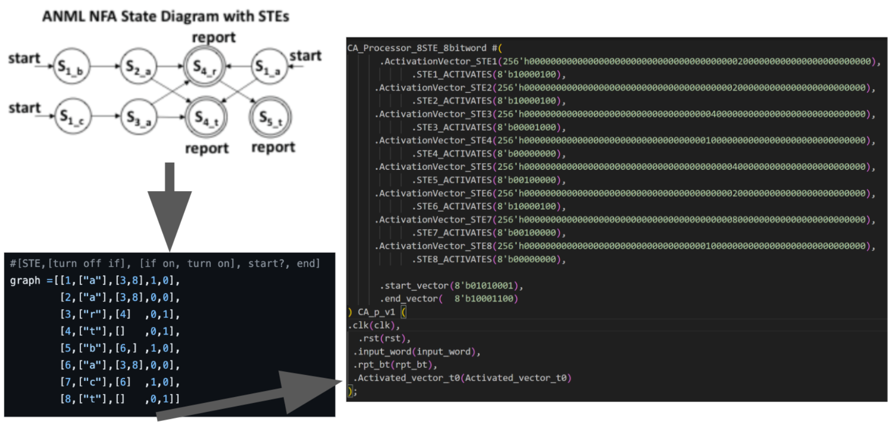
Match Report

This logic of AP controls what data to record into memory and how to take bytes from memory to PC through UART. This logic also includes button inputs which tell FPGA when to process or stream data. When streaming data back from FPGA, only 1 byte can be sent back at a time at a slower frequency than AP logic frequency. On PC, a python script runs which captures data sent through UART and stores as text file in PC.

Scripting

This project is very dependant on python scripts to create and validate design, all scripts are accessible on GitHub. One script can be called which will create all RTL logic where the inputs are text file representing the regex implementing the regular expression and a text file representing the data wanting to process. First the RTL logic is created, the AP logic is created by the regular expression text, the memory RTL is created by text data. This RTL is put into folder (given by user) with a copy of all input text and regular expression text (in case needed for future debugging), additionally verilog simulation log file in dumped into folder as well . An expected output log file is dumped into folder (software implementation of Autonoma Processor provided by the SIMS lab) to compare results. Lastly a copy of UART communication python script is placed into folder for testing FPGA implementation after the bit stream is created in vivado. This script is used when FPGA is programmed, and when running, will document all UART communication from FPGA memory.

Figure 3: Script Flow



```
#[STE,[turn off if], [if on, turn on], start?, end]
graph =[[1,["a"],[3,8],1,0],
        [2,["a"],[3,8],0,0],
        [3,["r"],[4]  ,0,1],
        [4,["t"],[]   ,0,1],
        [5,["b"],[6,] ,1,0],
        [6,["a"],[3,8],0,0],
        [7,["c"],[6]  ,1,0],
        [8,["t"],[]   ,0,1]]
```

```
CA_Processor_8STE_8bitword #(
      .ActivationVector_STE1(256'h0000000000000000000000000000000000000002000000000000000000000000),
            .STE1_ACTIVATES(8'b10000100),
     .ActivationVector_STE2(256'h0000000000000000000000000000000000000002000000000000000000000000),
            .STE2_ACTIVATES(8'b10000100),
     .ActivationVector_STE3(256'h0000000000000000000000000000000000040000000000000000000000000000),
            .STE3_ACTIVATES(8'b00001000),
     .ActivationVector_STE4(256'h0000000000000000000000000000000000100000000000000000000000000000),
            .STE4_ACTIVATES(8'b00000000),
     .ActivationVector_STE5(256'h0000000000000000000000000000000000000004000000000000000000000000),
            .STE5_ACTIVATES(8'b00100000),
     .ActivationVector_STE6(256'h0000000000000000000000000000000000000002000000000000000000000000),
            .STE6_ACTIVATES(8'b10000100),
     .ActivationVector_STE7(256'h0000000000000000000000000000000000000008000000000000000000000000),
            .STE7_ACTIVATES(8'b00100000),
     .ActivationVector_STE8(256'h0000000000000000000000000000000000100000000000000000000000000000),
            .STE8_ACTIVATES(8'b00000000),

      .start_vector(8'b01010001),
      .end_vector(  8'b10001100)
) CA_p_v1 (
.clk(clk),
  .rst(rst),
 .input_word(input_word),
 .rpt_bt(rpt_bt),
 .Activated_vector_t0(Activated_vector_t0)
);
```

Verification

Validation of logic in all levels was very crucial for this design and took must of the time to create design. Python scripts were used to simulate test benches for AP logic, UART communication, and data handling. These scripts were converted into the scripts creating automated design. The vivado tool suite has an integrated logic analyzer which aided in debugging signals in FPGAs. When full design was implemented on FPGA, results from hardware were compared from software results and was a metric for success for this project. Additionally data and real world regular expressions were provided by the SIMS lab to which were used to validate design.

4 Results

This project was able to get the FPGA hardware implementation of the RTL logic to work. Results coming from UART communication, which is AP processed data, were converted into a log file. The results from hardware match simulation results of RTL Verilog. The design processed each character byte with no stalling. Depending on the FPGA used, max frequency determined the throughput of the design. At 100 Mhz, there was considerable slack in design allowing for higher frequency which has not been tested thoroughly (completely) which is the future plans of this project. This means the entire latency of the design from start of the text to the end entirely depends on memory size, which currently is verified for 2^{16} byte. This data size is processed in 0.65540 m seconds (There is latency from initial cycles of AP) . The UART Communication was slower than processing logic (9600 baud rate). The most utilized components for this design were used by BRAM (>50% of

Total BRAM allocation used), this has to do with the text data placed on FPGA. The logic of Autonoma Processor is mostly made up of OR and AND gates with two long registers for STE vectors. The AP logic gets implemented as LUT, registers and mostly route dominated requiring less resources. Power of this implementation is mostly from Memory and routing as together it accounts for around 77% of total power.

Figure 4: 630STE FPGA Utilization

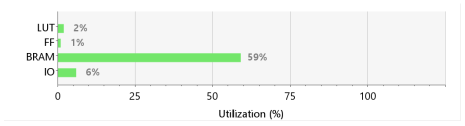
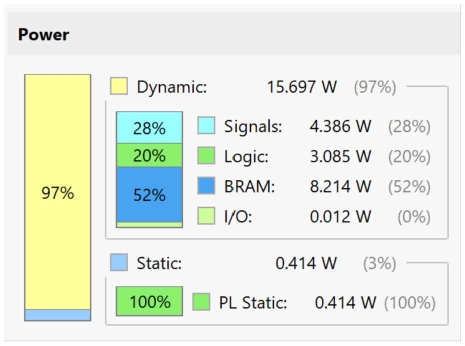


Figure 5: 630STE FPGA Power Report



Dynamic:	15.697 W	(97%)	
	Signals:	4.386 W	(28%)
	Logic:	3.085 W	(20%)
	BRAM:	8.214 W	(52%)
	I/O:	0.012 W	(0%)
Static:	0.414 W	(3%)	
	PL Static:	0.414 W	(100%)

Simulation and Hardware Implementation

The full design flow has been tested and verified for 630 STE elements, meaning that this was implementable in FPGA. In simulation, testing for AP has been at max 1024 but not yet tested fully in hardware. Further testing with larger data-sets is needed. My design is able to get same amount of matches as the provided SIMS labs software implementation of AP using the same data but is slightly differing by an offset in index value of match (this is due to how hardware reports the matches).

5 Discussion / Conclusion

This current implementation functions as intended to test AP logic feasibility as software (from SIMS lab) and hardware implementations of AP results match. The main benefit from this implementation would be the decrease latency there are no stalls in design when data is streamed in cycle by cycle. Additionally my solution allows an easy way to test and use design since scripts automate most of the process and FPGA implementation is accessible. an FPGA running this RTL only needs two GPIO button input, clock input and outputting only the UART pin for data handling, which are standard on most FPGAs.

Many design alternatives were considered, mainly for the data handling and method of match reporting such as using AXI stream for data, using a micro-controller for data transfers, using different FPGAs, streaming data into AP, from PC, or manually programming BRAM (instead of letting tool to optimize). The current implementation using custom RTL instead of imported IP block as is typical for FPGA projects was because these approaches had memory latency drawbacks. Additionally the Vivado tools optimize physical implementation to match behavior of RTL code. This came with trade off of increased design complexities as RTL had to be created manually.

The future of this project will explore improvements of architecture of AP logic and scale-ability of design. AP logic in this FPGA implementation will be optimized for that one regular expression (as AP logic uses static parameters for STEs), however true AP logic use in practice will involve AP logic that is programmed post FPGA implementation. This project seeked to show proof of concept for hardware implementation of AP logic and has accomplished this goal. Additionally the AP logic has been designed to be easily interchangeable with other accelerator logic which requires readout from FPGA memory.

6 Acknowledgements

The SIMS labs has greatly contributed to the foundations of this work. Their software implementation of AP logic and data for testing was crucial for this project. Special thanks to Ziyuan Wen and Alexis Le Glaunec, two graduate students guiding me on software, and professors Dr. Kaiyuan Yang and Dr. Konstantinos Mamouras for advising the direction of this project.

7 References

[1] Arun Subramaniyan, Jingcheng Wang, Ezhil R. M. Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. 2017. Cache automaton. In Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17). Association for Computing Machinery, New York, NY, USA, 259–272. https://doi.org/10.1145/3123939.3123986

[2] https://github.com/JuanLikesRice/Automated$_{parameter_C}A$/