Large Scale Pattern Matching Peripheral

EE2003:Course Project Report

Abishek S	Puvi Arasu N.P	Abhishek Sekar
EE18B001	EE18B027	EE18B067

December 2020

1 Problem Introduction

Pattern matching is used almost everywhere,from searching files in the computer to matching DNA patterns in genomics. Large scale pattern matching is one of the most important problems in data stream processing, which finds application in Network Intrusion Detection Systems (NIDS) and Natural Language Processing (NLP) to name a few fields, where a number of complex patterns are required to be matched in high speed data streams. There are also use cases where text from different sources need to match specific complex patterns.
These are CPU-intensive tasks and relying on software algorithms to handle these efficiently might not be a good idea. Hence there is a necessity for a dedicated peripheral / accelerator for solving this problem.

In this project, we implement pattern matching hardware as a peripheral that allows dynamic loading of regular expressions at once as well as efficient pattern matching for high speed data streams.

2 Relation to the Course

- **Designing and implementing a memory mapped peripheral and corresponding interface module**
- **Using the CPU built in course to drive the peripheral**
- **Designing the architecture and instruction set for the peripheral**
- **Examining and Modifying a Linker Script and writing an original C runtime**
- **Implementing an Efficient Data Path and Control Logic for the peripheral**
- **Memory Design of Pattern Matching Module**
- **Post Synthesis**

3 Solution to the Problem

Pattern Matching Peripheral
An accelerator is a device used to enhance the overall performance of the computer / system by bringing in multiprocessing capabilities. We propose to design and build a dedicated peripheral which acts as an accelerator for large-scale pattern matching using a dynamically reconfigurable bit parallel NFA(Non-deterministic Finite Automaton) (Dynamic BP-NFA) architecture.
The peripheral consists of several logically linked Pattern matching modules (say number of such modules = K) , each of which can match an incoming stream of data with complex patterns independently.[1]
Designing the accelerator as a peripheral allows it to be used with any system. Besides that, the proposed architecture solves a dual purpose in large scale pattern matching -

- It can support independent parallel pattern matching for data streams from different sources to match different complex patterns. This mode of operation (called MODE 0) is done by processing different patterns in different modules and sending data streams from different sources to the corresponding modules.
- It can support high speed data stream processing by matching the same data stream with multiple complex patterns simultaneously. This mode of operation (called MODE 1) is done by processing different patterns in different modules and sending the single data stream to all the modules for simulation at the same time.

We hence try to reduce the latency by using an efficient pattern matching algorithm implemented with dedicated circuitry, and increase throughput by bringing in multiple modules and parallelism in handshaking,storing, processing and transfer of data.

3.1 EXT Pattern and constructs

Our target class of patterns are EXT (Extended Pattern) which form the most important subclass of regular expressions. The pattern allows union of characters (character classes) and Kleene stars (unbounded repeats). Patterns are given as input to the program as strings in the form of regular expressions used in bash.
The allowed constructs[2] to represent EXT patterns in our program are :

- **Escape character :** \ represents an escape character. It removes any special meaning attached to the subsequent character (if it exists) and at times gives a special meaning to the subsequent normal character to represent escape sequences used in languages like C (like \n, \b, \a).
  Examples : \[, \n , \" , \\
- **Character :** A single character at a position in the pattern is represented by c_1. Escape characters (if present) may/may not change the meaning of the subsequent character, but the resulting character is treated as a single character. Examples : 1, b, \[, \n, \\
- **Character Class :** Allows union of characters for a given position in the pattern. It is basically a subset of the entire set of characters Σ which is the set of 256 ASCII characters here. Represented as $[c_1\ c_2...c_n]$ (the characters allowed is placed within square brackets). Some special constructs include representing a range of characters as c_1-c_2 within the square brackets and using $\bigwedge$ right after opening the square bracket to represent all characters except the ones mentioned. A single character (described above) is also a character class with one character. A character or character class is represented by β. Examples : [abc123$], [a-zA-Z_0-9], [Ô-9abc], [{ }[\]]
- **Optional :** Allows zero or one occurrence of the character or character class. Represented as β?. Examples : [a-z0-9]?, [\n]?, \\?, c?
- **Unbounded Repeat :** Allows zero or more repetition of the character or character class in the pattern. Represented as β^*. Examples : [a-z0-9]*, [\n]*, *, c*
- **Unbounded At Least Once Repeat :** Allows one or more repetition of the character or character class in the pattern. Represented as β+. Equivalent to $\beta\beta^*$. Examples : [a-z0-9]+, [\n]+, \\+, c+
- **Bounded Repeat :** Represented as β{x,y}. Allows repetition of the character or character class, minimum x and maximum y times in the pattern. Equivalent to (β?)y-xβx. Examples : [a-z0-9]{1,3}, [\n]{0,6}, \\{1,1}, c+{2,3}

3.2 Processing in the peripheral

Peripheral receives four different types of inputs:

- **No operation:** Pattern matching module does nothing whenever it receives a no operation input.
- **Preprocessing input:** It is used to load description of input patterns. Preprocessing input has 64 bit mask data which is to be stored in Block RAM, ModuleId to specify the target Pattern matching module and Mask Address to specify the address in block RAM where the mask data is to be stored.
  Format of preprocessing input:

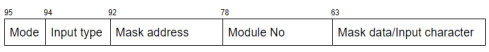

Figure 1: The Preprocessing Input

- **Run-time input:** Run time input has input character and ModuleId to specify pattern matching module(PMM). When PMM receives an input character, it makes a state transition for the target NFA by a fixed circuitry,detects matches and emits a matching information as output.
- **Reset input:** When PMM receives reset input, the finite state machine is reset to initial state.

We use bit-parallel pattern matching approach in which we first compactly encode the information of an input non-deterministic finite automation(NFA) i.e patterns in bit masks stored in a collection of registers and block RAMs through pre-processing inputs.Then the NFA is efficiently simulated by fixed circuitry using bitwise and arithmetic operations consuming one character per peripheral clock cycle regardless of the actual contents of an input text.If a PMM detects a match, then we display the matching information.

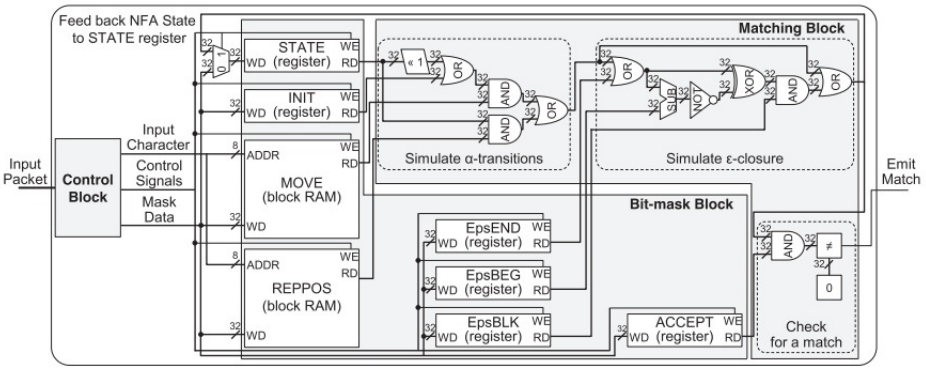

Figure 2: Pattern Matching Module with w=32[1]

3.3 NFA using the Extended Shift-And method

Given below is the state transition characteristic for the NFA.[2]

$$\text{State} = (((\text{State} << 1)|\text{Init})\&(\text{Move[t]})|(\text{State}\&\text{Rep\_Pos[t]}); \tag{1}$$

Simulating the ϵ transitions (ie spontaneous transitions of the NFA) with state mask.

$$\text{High} = \text{State}|\epsilon\text{\_end}; \tag{2}$$

$$\text{Low} = \text{High} - \epsilon\text{\_beg}; \tag{3}$$

$$\text{State} = (\epsilon\text{\_blk}\&((\tilde{\text{Low}}) \bigoplus (\text{High}))|\text{State}; \tag{4}$$

Finally, whether the pattern has been matched or not is tested by considering State&Accept. If it is 1, then it means that a match has been found.
The bitmasks used in the above equations are described below.

- **Init:** 64 bit mask that sets 1 at the bit position for the first state.
- **Accept:** 64 bit mask that sets 1 at the bit position for the final state.
- **Move[t]:** 64 bit mask that indicates all bit positions of the backbones with the character t in the NFA.
- **ϵ_beg:** 64 bit mask that sets 1 at the previous bit position of the lowest bit position of every ϵ block.
- **ϵ_end:** 64 bit mask that sets 1 at the highest bit position of every ϵ block.
- **ϵ_blk:** 64 bit mask that sets 1 at all bit positions of every ϵ block.
- **Rep_Pos[t]:** 64 bit mask that indicates all bit positions labeled with a character class β such that t is in β.

4 Work done outside the course

- **Handshaking - Design and Implementation:**

  - Frequency of cpu clock and accelerator clock are not same.We learnt to synchronize two hardware with different clocks using **handshaking process**.This is done in order to make the accelerator more efficient by choosing an optimal clock cycle based on the critical path.
  - We use a Source initiated handshaking process using the signals DATA_READY and DATA_ACCEPTED.Signal is sent to the peripheral from the CPU when the instruction for peripheral is ready and peripheral sends CPU a signal when the peripheral completes execution of an instruction so that the CPU can send the next instruction.

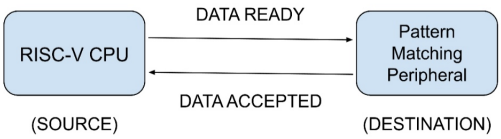

Figure 3: The Handshaking Process

- **Automaton and Regular Expressions:**
  There are many algorithms for exact string (substring) without pattern matching like Rabin Karp, Z-algorithm, Dynamic Programming, String Hashing, etc.
  However, they may not be deterministic or may not work for complex patterns. Using the fact that patterns (including complex ones) usually give rise to a small number of states when constructed as a Non-deterministic finite automaton (NFA), we decided to use the highly efficient Extended Shift And Method for simulating the resulting NFA.[1]
  The number of states is limited by the length of register used in the peripheral (we have used W = 64, but finally all registers are addressed by bytes, and can be extended by a simple increase in address space).
  We learnt some automaton theory, regular expressions, string matching and efficiency of many algorithms to come up with the optimal solution to the problem we were attempting to solve, and use it properly.

- **EXT Pattern Parser:**
  The algorithm we use, pre-processes the complex patterns and stores it in the form of bitmasks. But we can't expect the patterns to be input in that form.
  Hence we input patterns in regex form used in bash and in programs.
  We built our own EXT pattern parser program, which can validate an input pattern as valid EXT pattern, parse it and generate the bitmasks needed by the peripheral.

- **Pattern Matching Library:**
  When we build a peripheral, we also need some protocols and standard code for the user to interact with them easily, and get his/her job done.
  For this purpose, we also wrote functions which will enable easy and efficient usage of peripheral without any complications for the user.
  Though we have a well commented code to understand the purpose, arguments and return values of each function, we briefly describe the important functions here :

  - **PreProcess:** Generate bitmasks of a pattern for a particular module.
  - **PreProcessAll_M0:** Generate bitmasks of all given patterns and send it to the corresponding modules (using Mode 0 operation)
  - **PreProcessAll_M1:** Generate bitmasks of the same pattern and send it to all the modules. (using Mode 1 operation)
  - **SimulateNFA:** Sending input data stream to a particular module.
  - **SimulateNFA_All_M0:** Sending all input data streams to the corresponding modules.(using Mode 0 operation)
  - **SimulateNFA_All_M1:** Sending the same data stream to all the modules. (using Mode 1 operation)
  - **ResetNFA:** Resetting states for a particular module.
  - **ResetNFA_All_M0:** Resetting states for corresponding modules.(using Mode 0 operation)
  - **ResetNFA_All_M1:** Resetting states for all modules.(using Mode 1 operation)

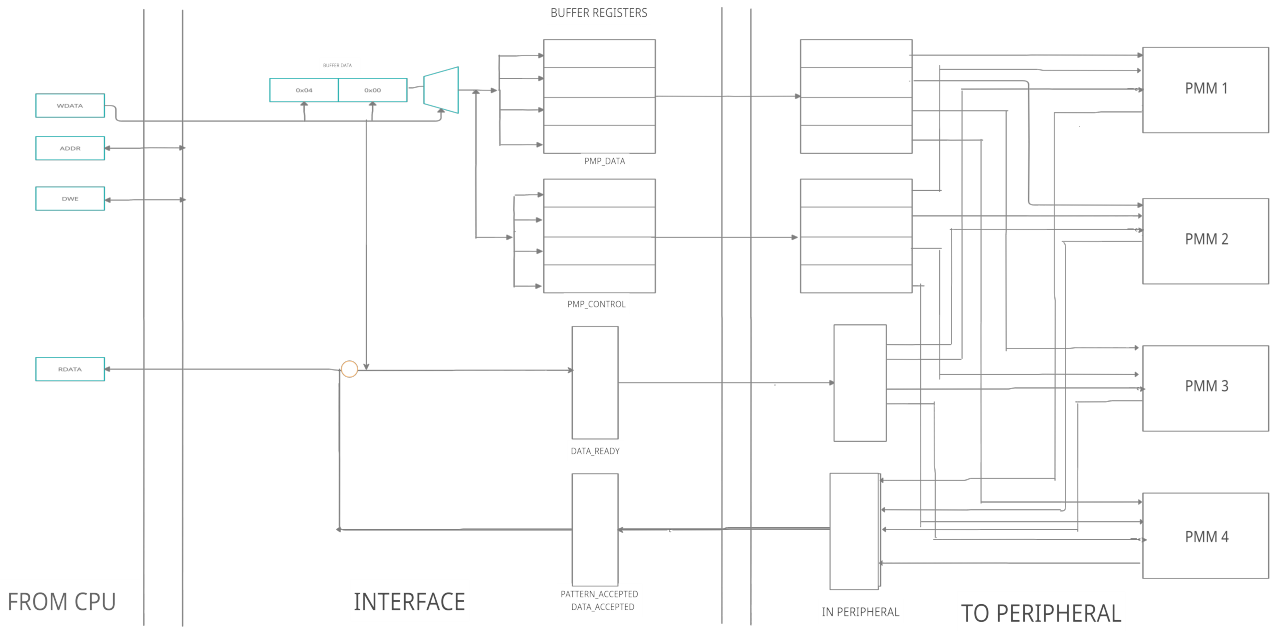

Figure 4: Schematic of the Hardware

5 Post Synthesis

An excerpt of the Post Synthesis Report containing the Component Statistics and the Heirarchial Component Statistics is shown below.

```
   Start RTL Component Statistics
---------------------------------------------------------------------------------
Detailed RTL Component Info :
+---Adders :
   3 Input     64 Bit       Adders := 4
   3 Input     32 Bit       Adders := 1
   2 Input     32 Bit       Adders := 2
+---XORs :
   2 Input     64 Bit         XORs := 4
   2 Input     32 Bit         XORs := 1
+---Registers :
               64 Bit    Registers := 32
               32 Bit    Registers := 4
               16 Bit    Registers := 8
                8 Bit    Registers := 8
                4 Bit    Registers := 1
                1 Bit    Registers := 8
+---RAMs :
             512K Bit         RAMs := 4
              16K Bit         RAMs := 8
             1024 Bit         RAMs := 1
+---Muxes :
  11 Input     32 Bit        Muxes := 1
  10 Input     32 Bit        Muxes := 1
   4 Input     32 Bit        Muxes := 2
   2 Input     32 Bit        Muxes := 11
   3 Input     32 Bit        Muxes := 1
   2 Input      8 Bit        Muxes := 48
   5 Input      6 Bit        Muxes := 1
  11 Input      4 Bit        Muxes := 1
```

```
   8 Input      4 Bit        Muxes := 1
   4 Input      4 Bit        Muxes := 1
   2 Input      4 Bit        Muxes := 2
   2 Input      2 Bit        Muxes := 2
   3 Input      2 Bit        Muxes := 1
   4 Input      2 Bit        Muxes := 1
   3 Input      1 Bit        Muxes := 13
   4 Input      1 Bit        Muxes := 5
   2 Input      1 Bit        Muxes := 78
---------------------------------------------------------------------------------
Finished RTL Component Statistics
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
Start RTL Hierarchical Component Statistics
---------------------------------------------------------------------------------
Hierarchical RTL Component report
Module alu32
Detailed RTL Component Info :
+---Adders :
   3 Input     32 Bit       Adders := 1
+---XORs :
   2 Input     32 Bit         XORs := 1
+---Muxes :
  11 Input     32 Bit        Muxes := 1
  10 Input     32 Bit        Muxes := 1
  11 Input      4 Bit        Muxes := 1
Module regfile
Detailed RTL Component Info :
+---RAMs :
             1024 Bit         RAMs := 1
Module decoder
Detailed RTL Component Info :
+---Muxes :
   4 Input     32 Bit        Muxes := 1
   2 Input     32 Bit        Muxes := 1
   5 Input      6 Bit        Muxes := 1
   2 Input      2 Bit        Muxes := 2
   3 Input      2 Bit        Muxes := 1
   4 Input      2 Bit        Muxes := 1
Module cpu
Detailed RTL Component Info :
+---Adders :
   2 Input     32 Bit       Adders := 2
+---Registers :
               32 Bit    Registers := 1
+---Muxes :
   3 Input     32 Bit        Muxes := 1
   2 Input     32 Bit        Muxes := 1
   4 Input     32 Bit        Muxes := 1
   8 Input      4 Bit        Muxes := 1
   4 Input      4 Bit        Muxes := 1
   3 Input      1 Bit        Muxes := 1
   4 Input      1 Bit        Muxes := 1
Module dmem
Detailed RTL Component Info :
+---RAMs :
             512K Bit         RAMs := 4
Module outperiph
Detailed RTL Component Info :
+---Muxes :
```

```
   2 Input     32 Bit        Muxes := 1
Module biu
Detailed RTL Component Info :
+---Muxes :
   2 Input     32 Bit        Muxes := 3
   2 Input      4 Bit        Muxes := 1
Module PMM
Detailed RTL Component Info :
+---Adders :
   3 Input     64 Bit       Adders := 1
+---XORs :
   2 Input     64 Bit         XORs := 1
+---Registers :
               64 Bit    Registers := 6
                1 Bit    Registers := 2
+---RAMs :
              16K Bit         RAMs := 2
+---Muxes :
   3 Input      1 Bit        Muxes := 2
   2 Input      1 Bit        Muxes := 7
   4 Input      1 Bit        Muxes := 1
Module PMP
Detailed RTL Component Info :
+---Registers :
               64 Bit    Registers := 4
               16 Bit    Registers := 4
                4 Bit    Registers := 1
Module PMP_interface
Detailed RTL Component Info :
+---Registers :
               64 Bit    Registers := 4
               32 Bit    Registers := 3
               16 Bit    Registers := 4
                8 Bit    Registers := 8
+---Muxes :
   2 Input     32 Bit        Muxes := 5
   2 Input      8 Bit        Muxes := 48
   2 Input      4 Bit        Muxes := 1
   3 Input      1 Bit        Muxes := 4
   2 Input      1 Bit        Muxes := 50
------------------------------------------------------------------------------
Finished RTL Hierarchical Component Statistics
------------------------------------------------------------------------------
```

6 Simulation Results

Patterns processed and stored in the 4 modules :

- **Module 1 :** abc+
- **Module 2 :**[abc]{0,3}d[e,f]?
- **Module 3 :** ab?[cd]
- **Module 4 :** [a-zA-Z][0-9]hello

SINGLE TEXT - SINGLE COMPLEX PATTERN MATCHING

Firstly, we give a character data as input to 3rd module alone :
INPUT :
a

OUTPUT :
Text :
a
— Matched Pattern :
ab?[cd]
@ 1st character
We then reset the state of NFA in Module 3.

MULTIPLE INDEPENDENT PARALLEL TEXT-PATTERN MATCHING

We pass the following 4 pieces of text to the respective modules independently and parallely (i.e) first character of all text would be sent and NFA simulated in the respective modules in parallel (till a character exists for a particular module).
Text for the modules,
INPUT:

- **Module 1 :** abdcabcc
- **Module 2 :** dafabdef
- **Module 3 :** acbdb
- **Module 4 :** c0a9hello0

OUTPUT :
Text :
acdbd
— Matched Pattern :
ab?[cd]*
@ 1st character

Text :
acdbd
— Matched Pattern :
ab?[cd]*
@ 2nd character

Text :
acdbd
— Matched Pattern :
ab?[cd]*
@ 3rd character

Text :
dafabdef
— Matched Pattern :
abc
0,3d[e,f]?
@ 6th character

Text :
abdcabcc
— Matched Pattern :
abc+
@ 7th character

Text :
dafabdef
— Matched Pattern :
abc
0,3d[e,f]?
@ 7th character

Text :
abdcabcc
— Matched Pattern :
abc+
@ 8th character

Text :
c0a9hello0
— Matched Pattern :
a-zA-Z
[0-9]hello
@ 9th character

We then reset the state of NFAs in all modules parallely.

SINGLE TEXT - MULTIPLE COMPLEX PATTERNS MATCHING IN PARALLEL

We pass a single data stream (in the form of text, here) to all the 4 modules parallely, where it is processed and results are obtained simultaneously as well.

INPUT :
abcdefghijklmnopqrtuvwxyz

OUTPUT:
Text :
abcdefghijklmnopqrtuvwxyz
— Matched Pattern(s) :
ab?[cd]*
@ 1st character

— Matched Pattern(s) :
ab?[cd]*
@ 2nd character

— Matched Pattern(s) :
abc+
ab?[cd]*
@ 3rd character

— Matched Pattern(s) :
abc
0,3d[e,f]?
ab?[cd]*
@ 4th character

— Matched Pattern(s) :
abc
0,3d[e,f]?
@ 5th character

7 Project Code

Given below is the link to the Master branch of the GitLab repository which contains all the codes used in this project.
GitLab Repository

8 Work Split Up

- **Abishek.S :** EXT Pattern Parser, Pattern Matching library in C for all functions that can be performed with the peripheral, Peripheral and Interface Architecture.

- **Puvi Arasu.N.P :** Decoding peripheral input and sending data and control signals to different modules,Handshaking process.
- **Abhishek Sekar :** Pattern Matching Module and Post Synthesis Report.

References

[1] Yusaku KANETA, Shingo YOSHIZAWA, Shin ichi MINATO, Hiroki ARIMURA, and Yoshikazu MIYANAGA. A dynamically reconfigurable FPGA-based pattern matching hardware for subclasses of regular expressions. *IEICE Transactions on Information and Systems*, E95.D(7):1847–1857, 2012.

[2] Gonzalo Navarro and Mathieu Raffinot. *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press, 2002.