A LOW LATENCY, HIGH THROUGHPUT AND SCALABLE HARDWARE ARCHITECTURE FOR FLOW TABLES IN SOFTWARE DEFINED NETWORKS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

GÖKSAN ERAL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

SEPTEMBER 2016

Approval of the thesis:

A LOW LATENCY, HIGH THROUGHPUT AND SCALABLE HARDWARE ARCHITECTURE FOR FLOW TABLES IN SOFTWARE DEFINED NETWORKS

submitted by **GÖKSAN ERAL** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Gülbin Dural Ünver
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Tolga Çiloğlu
Head of Department, **Electrical and Electronics Engineering** _____

Assoc. Prof. Dr. Ece Güran Schmidt
Supervisor, **Electrical and Electronics Engineering, METU** _____

Examining Committee Members:

Prof. Dr. Gözde Bozdağı Akar
Electrical and Electronics Engineering Department, METU _____

Assoc. Prof. Dr. Ece Güran Schmidt
Electrical and Electronics Engineering Department, METU _____

Assoc. Prof. Dr. Cüneyt Fehmi Bazlamaçcı
Electrical and Electronics Engineering Department, METU _____

Assoc. Prof. Dr. İlkay Ulusoy Parnas
Electrical and Electronics Engineering Department, METU _____

Assoc. Prof. Dr. Ali Ziya Alkar
Electrical and Electronics Engineering Dept., Hacettepe Uni. _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: GÖKSAN ERAL

Signature :

ABSTRACT

A LOW LATENCY, HIGH THROUGHPUT AND SCALABLE HARDWARE ARCHITECTURE FOR FLOW TABLES IN SOFTWARE DEFINED NETWORKS

Eral, Göksan
M.S., Department of Electrical and Electronics Engineering
Supervisor : Assoc. Prof. Dr. Ece Güran Schmidt

September 2016, 145 pages

Software Defined Networking (SDN) is a new paradigm which requires multi-field packet classification for each received packet by looking up Flow Tables which contain a large number of rules and corresponding actions. The rules are defined by upto 15 packet header fields including IP source and destination address. If more than one rule rule matches then the action of the highest priority rule is executed. Furthermore rules with wildcard fields are possible. The SDN Flow Table should scale with the rule count while providing high throughput supporting the Gbps network data rates. In addition, recent data center applications such as high frequency/speed trading require ultra low latency. Motivated by these requirements, this thesis proposes Fast Scalable SDN Table (FASST), a hardware architecture for a low latency, scalable and high throughput SDN Flow Table Implementation. FASST provides a high throughput up to 200 Mega-Packet-Per-Second (MPPS) while achieving a very low average latency. To this end, FASST caches the frequently accessed rules exploiting the known temporal locality in the network traffic. FASST is implemented and evaluated on real hardware using Altera Stratix-V state-of-the-art FPGA. For a net-

work characteristics showing strong locality, FASST always achieves a lower average latency compared to recent works with a decrease of up to %97.

Keywords: Software Defined Networks, Packet Classification, TCAM, Bit Vector, Field Programmable Gate Array (FPGA)

ÖZ

YAZILIM TANIMLI BİLGİSAYAR AĞLARI'NDAKİ AKIŞ TABLOLARI İÇİN DÜŞÜK GECİKMELİ, YÜKSEK VERİ HACİMLİ VE ÖLÇEKLENDİRİLEBİLİR BİR DONANIM MİMARİSİ

Eral, Göksan
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi : Doç. Dr. Ece Güran Schmidt

Eylül 2016 , 145 sayfa

Yazılım tabanlı bilgisayar ağları, çoklu alanlı paket sınıflandırmaları için kullanılan yeni bir yaklaşımdır. Bu sınıflandırma, yüksek sayıda kural içeren akış tabloları içerisinde gerçekleştirilmektedir. Akış tabloları içerisinde her bir kural, IP adreslerini de içeren 15 paket başlığından ve bir aksiyon alanından oluşmaktadır. Bu aşamada, SDN tanımlı ağ anahtarları, her bir alınan paketi bu kurallarla karşılaştırarak sınıflandırma işlemlerini gerçekleştirmektedir. Eğer bir paket birden fazla kurala uyarsa, bu paket için en yüksek öncelikli kuralın aksiyonu uygulanır. Ayrıca, akış tablosu içerisindeki kurallar 'wildcard' alanlarını da içerebilmektedir. Yazılım tabanlı bilgisayar ağlarındaki akış tabloları, Gbps ağ hızlarını destekleyecek şekilde yüksek veri hacimi sağlamalı ve artan kural sayılalarıyla ölçeklendirilebilir olmalıdır. Ayrıca, yüksek frekans/hız işlemi gibi veri merkezleri uygulamaları için çok düşük gecikme zamanı isterleri bulunmaktadır. Tüm bu motivasyonlar doğrultusunda, bu tez, yazılım tabanlı bilgisayar ağları için düşük gecikmeli, yüksek veri hacimli ve ölçeklendirilebilir bir

donanım mimarisi (FASST) sunmaktadır. Bu mimari, bir önbellek mekanizması kullanarak, çok düşük gecikme zamanlarında saniyede 200 milyon paket sınıflandırma yeteneğine sahiptir. Yüksek veri hacminde düşük gecikme zamanını sağlamanın arkasındaki ana fikir, bilgisayar ağlarındaki geçiçi bölgeselliği kullanmaktır. Sunulan mimari (FASST), Altera Stratix-V FPGA üzerinde entegre edilmiş ve performans değerlendirmeleri yapılmıştır. Güçlü geçici bölgesellik gösteren bilgisayar ağları için, bu mimari %97 oranına varan oranlarda ortalama gecikme zamanını düşürebilmektedir.

Anahtar Kelimeler: Yazılım tanımlı ağlar, paket sınıflandırma, TCAM, Bit vektör, Alanda Programlanabilir Kapı Dizinleri

To my family

ACKNOWLEDGMENTS

Foremost, I would like to express my deepest gratitude to my advisor, Assoc. Prof. Dr. Ece Güran Schmidt, for her excellent guidance and continuous support of my research. Besides my advisor, I would like to thank the rest of my thesis committee; Prof. Dr. Gözde Bozdağı Akar, Assoc. Prof. Dr. Cüneyt Fehmi Bazlamaçcı, Assoc. Prof. Dr. İlkay Ulusoy Parnas and Assoc. Prof. Dr. Ali Ziya Alkar for their encouragement and insightful comments.

I wish to thank ASELSAN A.Ş. for giving me the opportunity of continuing my post-graduate education and providing an appropriate environment to develop my studies. I would also like to express my special appreciation to my colleagues and seniors from workplace for their contributions on the improvement of my engineering skills.

I also thank TUBITAK for supporting my post-graduate thesis with a scholarship program.

Last but not the least, I wish to thank my all friends who never hesitated from giving their supports to me.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

$FASST$	Fast Scalable Software Defined Network Flow Table
SDN	Software Defined Networks
BVM	Bit Vector Module
RAM	Random Access Memory
$TCAM$	Ternary Content Addressable Memory
$FPGA$	Field Programmable Gate Array
$SRAM$	Static Random Access Memory
NOS	Network Operating System
API	Application Programming Interface
QoS	Quality of Service
GbE	Gigabit Ethernet
MA	Match Monitor
$FIFO$	First In First Out
MAC	Media Access Control
RTL	Register Transfer Level
FSM	Finite State Machine
SoC	System on Chip
NoC	Network on Chip
IP	Intellectual Property
$I2C$	Inter-Integrated Circuit
$MPPS$	Mega Packet Per Second
EPE	Early Power Estimator
DSP	Digital Signal Processor

IO	Input Output
LUT	Look Up Table

CHAPTER 1

INTRODUCTION

In contemporary computer networks, network applications such as QoS (Quality of Service) support or Access Control are realized by first classifying the packets into flows based on multiple number of header fields and then taking flow based actions. Packet classification is implemented in network nodes such as routers or firewalls using a number of techniques including heuristic algorithms, basic search algorithms or hardware-specific search algorithms. Packet classification is a difficult operation because of the need for wire-speed processing, support of thousands of rules and satisfying performance metrics such as latency, throughput and power.

Each network node is configured or reconfigured separately using device-specific interfaces which result in a big inefficiency in network management [1] in traditional IP networks. Software Defined Networking (SDN) is a new paradigm which aims for the flexibility of network device programmability [4]. SDN promotes centralization of network control by separating control and data planes in a forwarding device. In other words, network nodes are data plane devices that are only responsible for running classification algorithms on packet headers and applying related actions according to the rules decided by control plane. This infrastructure introduces new abstractions in networking by creating flexibility in layered architecture.

The communication between SDN control plane and data plane devices is provided with a well-defined programming interface. Through this interface, control plane devices (SDN controllers) send the set of rules for packet classification to SDN-enabled data plane device (SDN switches). These set of rules are stored in tables named Flow Tables in SDN switches. SDN Flow Table can have thousands of rules defined by

arbitrary combinations of up-to 15-tuple header fields. An SDN Flow is a group of packets that match a given rule or a defined set of rules. Rules with widcard fields are possible. If a packet matches more than one rule, then the rule with the highest priority is selected.

The SDN Flow Table lookup is a data plane function that is executed for each received packet. Hence, hardware implementations are required to support the high data rates in the order of 10s Gbps. Scalability of the implementation is important as the Flow Table sizes increase in time. In addition to such legacy metrics, the recent data center applications such as high frequency/speed trading require ultra low packet processing latencies. This new requirement is recognized by vendors [5] to produce specific low latency hardware.

Ternary Content Addressable Memory (TCAM) chips with O(1) look-up time are used in most of today's commodity switches for IP networks, but they are expensive [6] and consume a lot of power [7] in SDN applications due to the high number of SDN rules. Well-known hardware-based classification algorithms such as SRAM-based TCAMs, Parallel Bit Vector (BV), standard or parallel Bloom Filters (BF), hash-based schemes present convincing performance results for different metrics.

Motivated by the performance requirements listed above, this thesis presents the design, implementation and evaluation of FASST (Fast Scalable SDN Table), a pure hardware SDN Flow Table architecture, to be employed in SDN-enabled data plane devices (switches). FASST provides a very high throughput while achieving a very low average latency. The core idea behind scalability and achieving low average latency is exploiting the known temporal locality in the network traffic and caching the frequently accessed rules. To this end, we employ a Bit Vector Machine and an SRAM-based TCAM cache to store the entire flow table and the frequently accessed rules, respectively. A very recent work [8] proposes a complete hardware architecture for SDN Flow Table look-up with very extensive pipelining to increase the throughput. However, this design results in high packet latency. Caching frequently accessed rules is explored in another recent work [9]. However, the design and implementation of the lookup engine is in software.

The contributions of the thesis can be summarized as follows:

- FASST architecture and operation described with all details about gate-level design justifying the design decisions.
  - Development of novel hardware modules to run two matching engines with different latencies concurrently and maintain the frequently accessed rules in the TCAM.
  - Implementing and programming the soft processor for the caching functions.
- Design improvements of the relevant existing work.
  - Adapting the Bit Vector architecture in [8] including hierarchical organization and different rule partitioning.
  - Adapting the SDN rule dependency checking algorithm in [9] for hardware implementation.
- Design and implementation of the SRAM-based TCAM for SDN rule lookup.
- Full scale implementation and functional verification of FASST on Altera Stratix-V using Altera Signal Integrity (SI) Development Board. A detailed presentation of the implementation.
- Detailed performance evaluation of FASST with both simulation and on the development board together with a test bench to generate packets. Measurement of throughput, latency and power consumption and comparison with relevant previous work.
- Discussion of the generalization of the design and implementation of components that affect QoS performance for this generalization.

The results show that FASST can perform the lookup of 512 SDN rules reaching a throughput of 200 Mega-Packet-Per-Second (MPPS) while achieving a very low average latency down to 25 nsec. For these throughput and latency values, the power consumption is monitored as 5.27 Watts in real time. We show that FASST offers a tradeoff between average lookup latency and power consumption. Moreover, due to SRAM-based infrastructure, our architecture is scalable with respect to clock speed and SDN rule set size.

The rest of the thesis is organized as follows:

In Chapter 2, Software Defined Networking and packet classification techniques are explained. For this manner, a vertical hierarchy between SDN-enabled control and data plane devices is discussed. Moreover, packet classification concept regarding flow tables and a well-known programming interface ,OpenFlow, are presented. The literature overview on network packet classification techniques used in SDN are mentioned afterwards. Here, data structures and hardware-based classification algorithms are presented. Among the existing algorithms, the hardware based parallel Bit Vector (BV) algorithm is discussed in detail. Rule caching problem is also analyzed in this chapter. The detailed improvements in this thesis regarding Bit Vector implementation and Rule Caching are discussed in Section 2.4.3.

In Chapter 3, the proposed hardware architecture FASST, which provides a high throughput and very low average latency on SDN packet classification, is explained in detail. For this manner, conceptual design details of all functional elements including high speed engine Bit Vector Module (BVM), low-latency cache TCAM, and locality detection units are presented. Adaptation of the rule-dependency algorithms to our hardware-based platform is also given.

The hardware implementation of FASST is presented in Chapter 4. Design procedure which is completely dependent on hardware resources on FPGA is shown by demonstrating communication interfaces between blocks. Moreover, in this chapter, the results of static (functional) tests of main hardware blocks inside FASST are given to verify the functionality. Furthermore, System on Chip (SoC) design using a soft processor in order to generate rule dependency graph and monitor power consumption is given to provide a complete understanding.

In Chapter 5, performance evaluation of the complete design of FASST is provided. For this purpose, synthetic rule sets and example traffic traces are generated and given to FASST as input. For all these synthetic data traces, FASST is tested and verified at each phase. Complete functional simulation models and real time hardware test results are presented. Achieved average latencies, throughput and power consumption values are given with different cache hit rates. Moreover, scalability of our architecture with respect to clock rate, rule size and power consumption is analyzed with

real-time data. Furthermore, a comparison with recent works in terms of throughput, latency and power consumption are made. Lastly, resource utilization on FPGA is given for different rule-set size.

In Chapter 6, the environment for hardware tests and simulations is presented. Here; a designed interface board, a FPGA developer kit, and utilized software tools are explained. In order to give a comprehensive understanding of the verification and implementation environment, the data flow between these components are stated.

The summary of this thesis and possible future works are presented and discussed in Chapter 7.

CHAPTER 2

RELATED WORK

2.1 Software Defined Networking

The network device functionality is described as two planes; the *data plane* and the *control plane*. Data plane is responsible for forwarding of each incoming packet. *Packet forwarding* consists of table look-ups and accordingly switching of packets to output ports and further processing. The control plane configures the look-up tables of the data plane resulting in the desired forwarding behavior. On the one hand, packet forwarding must be low complexity such that it can be implemented in hardware for high throughput. On the other hand the control functions have high complexity and involve communicating with the other devices and running certain algorithms. Hence, they are implemented in software. In the traditional networking paradigm, these two planes co-exist in the same network device resulting in fully distributed control.

Software Defined Networking (SDN) is a new approach to computer networking that provides an abstraction of lower-level functionality in order to manage network services. The architecture of SDN decouples the network control and forwarding functions and enables the network control to become directly programmable through programming interfaces. In Software Defined Networking, the packet forwarding tables in the network devices are *remotely* configured by SDN controllers through a programming interface. do not implement the control functions. Instead they are are remotely controlled by Controllers, and only responsible for packet forwarding. The SDN Controllers offer a centralized view of the overall network.

2.1.1 SDN Infrastructure

In traditional IP networks, each network switching node is loaded with its own features, operating system and a specialized packet forwarding hardware. Due to this existing bundle between operating system and forwarding hardware inside a node, network manageability is very complex and hard. Each packet is processed individually by the switch.

Software Defined Networking breaks the vertical integration of control plane and data plane on the same device. In other words, in order reconfigure the network core, there is no need to manage each network node separately using vendor specific configuration processes. By separating the control logic from switching hardware, a flexibility is introduced while creating new abstractions and policies in networking. Different than traditional IP networks, packets are classified into flows and processed accordingly. An *SDN flow* is a group of packets some common features which are determined by the SDN control logic together with the forwarding actions that should be taken at the SDN switches.

SDN infrastructure includes a similar set of network devices such as routers and switch except for the fact that physical devices in SDN are simple forwarding devices without an embedded control and software to take autonomous decisions [1]. Overall SDN architecture and building blocks are depicted in Fig. 2.1. Control logic is moved to an upper level external module, the so-called SDN controllers or Network Operating System (NOS). The control logic is directly connected to simple forwarding devices through a southbound Application Programming Interface (API) and sends a set of instructions used in forwarding process. Therefore, southbound Programming Interfaces defines the communication protocol between forwarding device and controller. OpenFlow is the first and most common open standard southbound interface among many others [2].

Control plane elements are connected to different application layers via a northbound API. Hence, this northbound interface provides an abstraction for the low level instructions that control plane creates to program and manage data plane elements [1].

Although SDN can be adopted to traditional IP network environment such as enter-

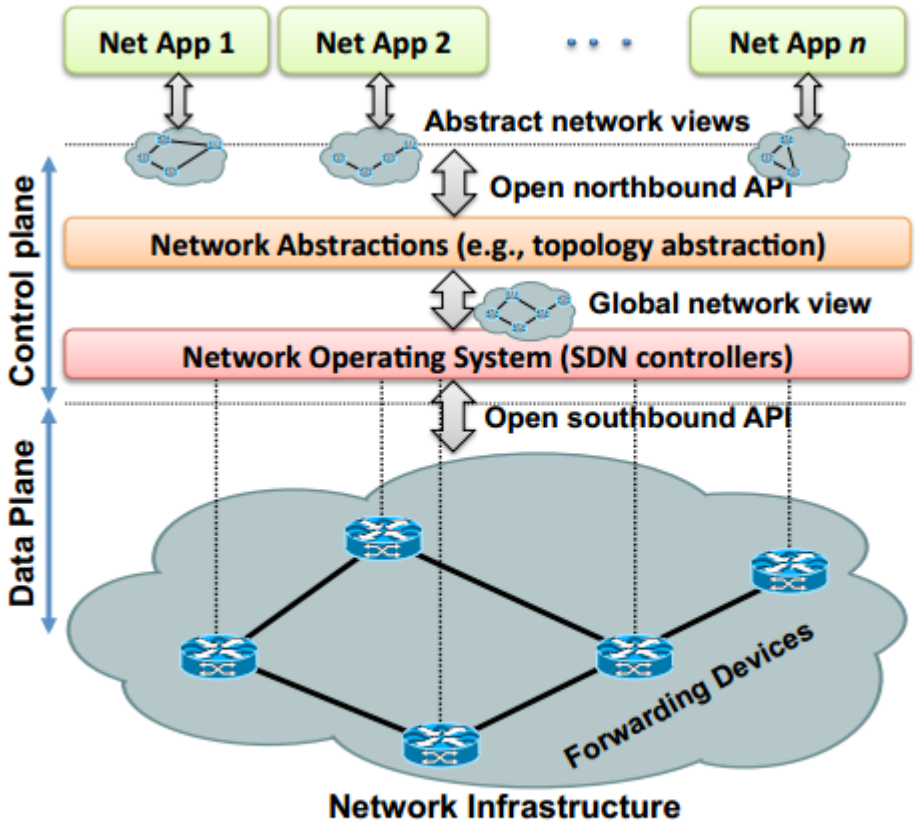

Figure 2.1: Layered architecture of SDN [1]

prise networks and data centers, there are many other network applications that dictate different QoS requirements. For example, traffic engineering and measurement, traffic monitoring, security, firewall, data center networking require different type of performance metrics. SDN applications can be deployed in all of these networks due to its flexibility on configuration, reconfiguration and management. Moreover, from the technical standpoint, most companies such as cloud providers and large enterprises with a significant data center investment regard SDN as one the most attractive network infrastructure [10].

2.1.2 OpenFlow Protocol

OpenFlow is a pioneer and flexible protocol that can be used to manage network traffic between the control plane and data plane in Software Defined Networking ap-

proach. Using OpenFlow protocol, packet processing policies and messages defined by the control plane devices (SDN controllers) can be sent to forwarding devices. As a result, it enables researchers to develop and run experimental protocols on a variety of network environment.

OpenFlow defines two key SDN switch components: secure channel and *Flow table* [2]. As long as the correct functionality is preserved, these two components can be designed and developed in many ways by researchers. Secure channel is the communication interface that connects an OpenFlow-enabled forwarding device to a controller. In other words, secure channel is the implementation of configuration and managing interface.

The flow table stores flow entries which define the packet features and the corresponding action for the SDN flows. OpenFlow protocol has a flow entry format as seen in Fig. 2.2. A flow entry basically consists of 3 distinct fields: Header fields, counters and actions. Header fields are used in order to compare the incoming packet headers and determine whether a match between packet and rule entry occurs. A flow entry can have up to 15-tuple header fields, where any combination of wildcard fields exist. It is possible that a packet conforms to more than one flow entry. OpenFlow enabled devices, packets are matched against flow entries based on entry prioritization.

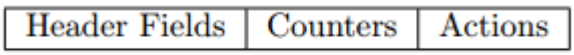

Figure 2.2: Flow entry format for OpenFlow v.1.1.0 [2]

Counters are used to record the conditions for the associated flow entries when a match condition is observed on the header fields. For example, the number of received packets or the number of received bytes can be updated by using counters in a flow entry when a matching condition for that flow entry is identified. Each flow entry has an action field associated with a header field. When the flow table identifies a matching flow entry, the related action field is applied.

Table. 2.1 shows the match fields, or header fields, inside a flow entry formatted according to OpenFlow v.1.1.0. There are a total of 15 match fields for packets.

These match fields are applicable for any incoming packet headers. Each flow entry cell contains a specific value, or ANY, which matches any value [2]. Moreover, for IPv4 source and IPv4 Destination fields, the prefix lengths for each entry should be specifically defined.

Table 2.1: Header fields for OpenFlow v.1.1.0

Header Field	Ingress Port	Metadata	Source MAC	Dst MAC	Eth. Type
Length	32 bits	64 bits	48 bits	48 bits	16 bits
Header Field	VLAN ID	VLAN Priority	MPLS Label	MPLS Priority	IPv4 Src
Length	12 bits	3 bits	20 bits	3 bits	32 bits
Header Field	IPv4 Dst	IP Protocol	ToS	Src Port	Dst Port
Length	32 bits	8 bits	6 bits	16 bits	16 bits

2.2 SDN-Enabled Switches

An SDN-enabled switch performs a matching process for each received packet and takes the associated action for the matching flow table entry. A packet matches a flow table entry if the headers fields of the incoming packets match the header fields of a rule entry in flow table. During matching process, prefix lengths of IPv4 Src and IPv4 Dst and mask bits for other header fields are considered. For example, if a header field in a rule entry has ANY value in flow table, which means a masked field, then, the corresponding header fields of all incoming packets match this header field of the rule entry. However, for a complete match, all header fields of incoming packets somehow match all header fields of a rule entry in flow table. Checking a match condition of header fields can be carried out either sequentially or concurrently.

2.2.1 SDN-Enabled Switches on the Market

There are many SDN compatible commercial and open source products on the market. Most companies present hardware and software solutions for the current SDN

applications, ranging from small business equipments to the high processing data center products. A list of OpenFlow-enabled devices from various companies or organizations are illustrated in Fig. 2.3 [1].

Most of the OpenFlow-enabled hardware switches perform packet classification using Ternary Content Addressable Memories (TCAM). This is because using TCAMs supports a big set of policy based features while maintaining line rate forwarding. For example, Exchip NP-4 provides a TCAM memory that stores from 125K to 1000K flow table entries and can process packets at wire speed for 100 Gigabit Ethernet (GbE) [11]. Similarly, NoviFlow offers an OpenFlow version 1.3 compatible switch called NoviSwitch-1248 with wire-speed performance for a 200 Gbps throughput [12]. NoviSwitch-1248 has a TCAM memory that supports up to 1 million wildcard or 3 million exact match flow entries. However, both of these switches suffer from high power consumptions due to TCAMs being power-hungry devices. Moreover, the power consumption increases linearly with the size of stored flow entries. For example, typical power consumption of NoviSwitch is around 500 W, which is high value for many network providers aiming to meet strict requirements for power [12].

There are also specialized programmable hardware switches such as NetFPGA in order to overcome some of the restrictions composed by TCAM chips [13]. NetFPGA is an hardware implementation of OpenFlow switch on a Xilinx Vertex-II Pro 50 FPGA. This switch implements SDN flow tables by using a series of on-chip TCAMs and off-chip Static Random Access Memories (SRAMs). In contrast to native TCAMs utilized in most commercial switches, NetFPGA gives a RAM-based TCAM architecture that supports wildcard and exact look-up features. A remote controller can manage NetFPGA via OpenFlow API.

2.2.2 Performance Requirements and Application Characteristics

SDN is first proposed for and still frequently employed in data centers. Recently, applications such as high frequency/performance trading (HF/PT) are introduced to be implemented in data centers. These applications require ultra-low latency under 1 usec in addition to the high throughput of 100s of MPPS. Prominent network equipment manufacturers already take this trend into consideration in their products [1]. In

Group	Product	Type	Maker/Developer	Version	Short description
Hardware	8200zl and 5400zl	chassis	Hewlett-Packard	v1.0	Data center class chassis (switch modules).
	Arista 7150 Series	switch	Arista Networks	v1.0	Data centers hybrid Ethernet/OpenFlow switches.
	BlackDiamond X8	switch	Extreme Networks	v1.0	Cloud-scale hybrid Ethernet/OpenFlow switches.
	CX600 Series	router	Huawei	v1.0	Carrier class MAN routers.
	EX9200 Ethernet	chassis	Juniper	v1.0	Chassis based switches for cloud data centers.
	EZchip NP-4	chip	EZchip Technologies	v1.1	High performance 100-Gigabit network processors.
	MLX Series	router	Brocade	v1.0	Service providers and enterprise class routers.
	NoviSwitch 1248	switch	NoviFlow	v1.3	High performance OpenFlow switch.
	NetFPGA	card	NetFPGA	v1.0	1G and 10G OpenFlow implementations.
	RackSwitch G8264	switch	IBM	v1.0	Data center switches supporting Virtual Fabric and OpenFlow.
	PF5240 and PF5820	switch	NEC	v1.0	Enterprise class hybrid Ethernet/OpenFlow switches.
	Pica8 3920	switch	Pica8	v1.0	Hybrid Ethernet/OpenFlow switches.
	Plexxi Switch 1	switch	Plexxi	v1.0	Optical multiplexing interconnect for data centers.
	V330 Series	switch	Centec Networks	v1.0	Hybrid Ethernet/OpenFlow switches.
	Z-Series	switch	Cyan	v1.0	Family of packet-optical transport platforms.
Software	contrail-vrouter	vrouter	Juniper Networks	v1.0	Data-plane function to interface with a VRF.
	LINC	switch	FlowForwarding	v1.4	Erlang-based soft switch with OF-Config 1.1 support.
	ofsoftswitch13	switch	Ericsson, CPqD	v1.3	OF 1.3 compatible user-space software switch implementation.
	Open vSwitch	switch	Open Community	v1.0-1.3	Switch platform designed for virtualized server environments.
	OpenFlow Reference	switch	Stanford	v1.0	OF Switching capability to a Linux PC with multiple NICs.
	OpenFlowClick	vrouter	Yogesh Mundada	v1.0	OpenFlow switching element for Click software routers.
	Switch Light	switch	Big Switch	v1.0	Thin switching software platform for physical/virtual switches.
	Pantou/OpenWRT	switch	Stanford	v1.0	Turns a wireless router into an OF-enabled switch.
	XorPlus	switch	Pica8	v1.0	Switching software for high performance ASICs.

Figure 2.3: Hardware and software switches for OpenFlow Protocol [1]

OpenFlow enabled data plane devices, using cut through switches in order to meet strict latency requirements is not as effective as in conventional network equipments [14]. This is due to the fact that the number of OpenFlow enabled packet headers can reach up to 15 distinct fields, which must be assembled to generate a flow identifier.

The contemporary network implementations and network applications define the performance requirements of a SDN-enable forwarding device as follows:

- *Throughput*: Throughput is defined as the packet processing rate in computer networks. Today, most commercial SDN switches is designed to achieve high throughput of 100s of MPPS. In order to meet the needs of high performance networks such as data centers, pipelined architectures [15], [8], or new flow schedulers [16] are developed. As a result, desired line rate operations can be achieved.

- *Latency*: Latency is the amount of delay experienced by a packet inside a switch. In SDN environment, low latency lookup processing is preferred due to the nature and criticality of the applications [17]. Recently, applications such as high frequency/performance trading (HF/PT) are introduced to be implemented

in data centers. These applications require ultra-low latency under 1 μsec in addition to the high throughput of 100s of Mpps. Prominent network equipment manufacturers already take this trend into consideration in their products [5].

- *Power*: Certain network environments require low power consumption during look up process. While TCAMs, which are mostly utilized in today's commercial SDN switches, provide *O(1)* performance, they are power hungry devices. Some power-gating techniques in recent works using activation schedulers [18], or prediction circuitry [14] are developed to meet power requirements.

- *Scalability*: Scalability of designs in SDN environment is very important due to newly emerged applications. OpenFlow provides a flexible environment in SDN that evolves very quickly. This results in new applications that impose different number of match fields, or rule entries. As a result, scalability of designs with respect to certain parameters such as rule size, header size and throughput is desired.

- *Flow Table Size*: The number of entries/rules in the flow tables in SDN-enabled switches is growing in order to meet the needs of future SDN deployments [1]. Most of the fast classification engines such as TCAMs have a small size of flow tables due to high number of wildcard entries. There are studies to decompose large SDN tables into small ones and distribute them across the network [19]. However, a stand alone classification engine that stores a high number of rule entries in flow table is always desired for manageability purposes.

Among the performance requirements described above, there is a strict correlation between throughput and latency. To increase the throughput, the lookup process can be pipelined, which causes an increase in latency. On the other hand, to keep the latency at a reasonable value, packet processing rates can be decreased. Therefore, achieving high throughput and low latency design is desirable for high performance and latency sensitive data centers.

Our work in this thesis exploits the temporal locality of packet flows created by network applications. [9] reports a 97% of look ups match 10% of the rules based on real network trace experiments. Furthermore, it is found that 55% to 80% of the traffic

consists of large volume (elephant) flows [20] that match the same entries in the SDN flow table.

2.3 Hardware Packet Classification

2.3.1 Problem Definition

Packet classification is performed on packet headers, which consist of single-field or multi-field bit arrays. A classifier includes rules or policies [21], where each rule has multiple fields if multi-field classification is considered. A packet is regarded to match a rule if each individual field of the packet matches each field in the rule.

Different from single-field packet classification, multi-field packet classification induces partial overlapping conditions between rules. In other words, a single packet can match two or more rules in a classifier at the same time. This is basically caused from various matching criteria in multiple fields in rules using mask bits or IP prefix lengths. As the number of individual fields in rules increase, the probability of confronting overlapping conditions also increase. Therefore, priority based classification is performed in multi-field classifiers. As a result, lookup performance is directly affected.

Considering the recent capabilities of OpenFlow in SDN, there is a rapid increase in the number of supported header fields and flow table numbers inside a classifier [22]. For example, OpenFlow v.1.3.0 describes the flow match fields as OpenFlow Extensive Match (OXM) format, which is a typer-length-value (TLV) format [23]. The length of OXM TLV can reach up to 40 distinct header fields with 259 bytes. This high number of fields of rules in flow table will require excessive amount of processing capabilities in order to cope with too much overlapping conditions with priorities. Therefore, designing a classifier with the optimum capabilities such as high throughput and and scalability with the newly emerged SDN applications is the problem of multi-field packet classification problem.

2.3.2 Data Structures and Algorithms

Packet classification algorithms considering hardware platforms are categorized into several classes in [24], including basic data structures and hardware based classification algorithms.

Basic data structures includes linear search, hierarchical tries and set-pruning tries [25]. Linear search basically performs lookup operation for the incoming packet header sequentially for the linked-listed rules. For multi-field packet headers as in SDN OpenFlow protocol, this search provides poor scalability in terms of field numbers. Hierarchical trie, or multi-level trie, is a recursively constructed data structure, which can be multi-dimensional. Multi-dimension tries are constructed by regarding prefix rules. A traversal algorithm can be utilized to perform classification for the incoming packet. The query complexity of a multi-level trie is $O(W^d$, where W is the dimension range as $[0,2^W-1]$, and d is the number of dimensions. For 15-tuple SDN rules where each header consists of at least 356 bits, this query time introduces high latencies for the incoming packets. Set-pruning tries [25] is very similar to multi-level tries with a lower query time, which is $O(dW)$. This is accomplished by replicating the rules during traversal. However, this process increases the memory storage utilization, which is inapplicable for finite-storage switches. Moreover, $O(dW)$ query time is still too much for latency sensitive applications such as high frequency/performance trading (HF/PT).

Hardware based algorithms include TCAMs, bit-map intersection [26], bit vector algorithm [8], [15], [27], [28], bloom filters [29], [30], [31] as decomposition-based algorithms, and other decision-tree-based algorithms such as HyperCuts [32].

2.3.3 Selected Works on Hardware Classification Algorithms

Most of the hardware packet classification algorithms are considered in two categories: Decomposition-based and decision-tree-based algorithms.

In decomposition-based packet classification, each fields in a packet header is searched independently, and final result is obtained by combining these each search result. Dis-

tributed Crossproducting of Field Labels (DCFL) is one of the decomposition-based algorithm [33]. In this approach, independent lookup processes on each field in a multi-field packet header are performed. After that, a multi-stage aggregation network is utilized in order to combine each lookup result for the purpose of avoiding huge number of combinations in matching fields. This method is impractical for the classification of multi-field headers. For example, for SDN OpenFlow v.1.0.0 packets, the existing 15-tuple headers leads to an exponential increase on the resource consumption together with latency.

Using Parallel Bit Vector (BV) algorithm is another example of decomposition-based approaches. This method is proposed by Lakshman *et al.* [27], and presents parallel lookups on each individual field first. The result of each lookup process is a bit vector, where each bit in this bit vector represents a rule. A Bitwise AND operation of each local bit vector results gives the final matched rule index in bit vector. Due to abundant parallelism in FPGAs, BV algorithm provides high throughput as well as low resource utilization. In [15], [28], pipelined architecture of BV approach is developed and implemented on FPGAs. A two-dimensional pipelined architecture on BV is proposed to increase the achieved throughput value [8].

Bloom filters are mostly used in packet classification algorithms with new adaptations such as parallel bloom filters [29], dynamic bloom filters [30], compressed bloom filters [31]. Bloom filters possess high query efficiency and very low resource consumption for hardware based classifications. However, bloom filters do not store the actual data on memory, instead, they use independent hash functions to map the packet fields into linear vectors. Therefore, due to collisions in hash functions, there exist false positives, which means a match condition although there is no match at all. In order to minimize the probability of false positives, parallel bloom filters are proposed, which is actually a decomposition-based algorithm. In [29], data of multiple-fields are stored in independent bloom filters and parallel lookup is performed in each of them. Relevance information between independent lookup processes are handled by using twice verification hash functions. Since each bloom filter has a probability of false positive, using independent bloom filters decrease overall false positive probability (FPP) because of exponential degree resulted from parallelism. However, as the number of individual fields increase, the required hash functions should be recon-

figured to optimize FPP at each time. Moreover, this approach is not convenient for the networks that force priority based classification due to the fact that no priority information is utilized in these data structures.

TCAMs are employed in many multi-field classification solutions [34], [35], [36] due to their *O(1)* lookup performance. However, TCAMs are not scalable with respect to clock rates and rule set sizes. Moreover, rapid increase in flow table size in SDN costs too much resource and power consumption. In order to optimize TCAM approach in terms of scalability and power, SRAM-Based TCAM implementations on hardware utilized [37], [38]. Moreover, the combination of TCAM-BV architecture is proposed in [39].

Decision-tree-based algorithms deals with a search space, which is cut recursively into smaller spaces by using the information of header fields in packets. In decision based algorithms, each rule defines a hypercube in a n-dimension space, where n is the number of header fields in packet. HiCuts [40] and HyperCuts [32] are two similar examples of decision-tree based hardware classification algorithms. While HiCuts constructs a decision tree to determine next dimension to cut and the number of cuts in this dimension, HyperCuts provides an independent cutting process for each field, which leads to a shorter decision tree.

2.3.4 Implementations on Different Hardware Platforms

Field Programmable Gate Arrays (FPGA) are widely used in order to develop and evaluate packet classification algorithms. Due to existing abundant parallelism, FPGAs can provide an appropriate platform to design low power, high throughput, low latency classification engines. Furthermore area/delay/power gaps between FPGA–ASIC platforms are closing increasing the viability of FPGA for fast hardware implementations [41].

FPGAs have limited on-chip resources which leads to small rule set implementations. Although using off-chip memories can solve this problem, additional latency and power introduced by these off-chip resources and decreased clock rate cause poor performance during lookup process [8].

Naous *et al.* implement an OpenFlow switch on NetFPGA [13], which is a Xilinx Virtex-2 Pro 50 FPGA board. Classification is performed for 10-tuple OpenFlow headers using a combination of on-chip TCAMs and an off-chip SRAM. The overall design can handle 1 Gbps line rate, together with 32K exact match entries and 32 wildcards entries. This sustained throughput, 1 Gbps, is very small compared to recent studies. Moreover, since TCAMs are not scalable with clock rate and header size, NetFPGA does not provide a flexible architecture to further increase rule set and header size.

Jiang *et al.* uses a SRAM-based TCAM classifier and implements the architecture in Xilinx Virtex 7 FPGA [37]. The design can support 1K rules with 150-bits headers in TCAM with sustaining a throughput of 200 MPPS. The latency exposed by each packet is about 6 clock cycles. Due to SRAM-based TCAM approach, low power dissipation is achieved at the desired throughput. However, in [37], while power consumption is linearly scalable with TCAM width, the the results are obtained by estimation. Real-time power measurement can give nonlinear relations with respect to increasing header bits.

In [42], hardware implementation of a pipelined decision tree approach is performed on Xilinx Vertex 5 FPGA. This design can store 1K 12-tuple rules in a single FPGA by achieving a throughput of 40 Gbps for the minimum size (40 bytes) packets. However, this decision tree approach is rule-set dependent, which may lead to rule set expansion and excessive utilization of resource utilization. Taylor *et al.* propose an architecture named Distributed Crossproducting of Field Labels (DCFL) and implement the architecture on Xilinx Virtex-II Pro FPGA [33]. An optimized implementation of DCFL is predicted to achieve 100 MPPS throughput while supporting 200K rules. However, in this approach, standard 5-tuple match is utilized. In SDN, the number of individual fields in headers can reach up to 15 in our case, which leads to an exponential increase in matching combinations in aggregation network used to combine individual field results. Moreover, using Bloom Filters results in false positives, which can be undesirable in specific applications.

Song *et.al* [39] propose an architecture called BV-TCAM for multi-match packet classification and implement the design on Xilinx XCV2000E FPGA. In this work, while

TCAM performs prefix match or exact match, a tree-bitmap implementation of the BV algorithm handles source and destination port lookup. However, in [39], overall FPGA implementation details and results are not provided. Furthermore, the estimated sustainable throughput, which is 10 Gbps, is very low compared to other BV [8], [15], [28] approaches.

In [8], [15], [28], pipelined architecture of Bit Vector Algorithm is proposed and implemented in Xilinx FPGAs. Although pipelining provides high throughput and scalability with respect to rule size, the latency introduced by these design is quite high due to high number of pipeline stages.

2.4 Works Closely Related To This Thesis

The motivation of FAAST is achieving a very small overall latency during SDN packet classification together with sustaining very high throughput. Moreover, while satisfying these performance requirements, we also study on the scalability of the design for future needs of SDN environment such as increasing flow table size or increasing field size for packets. For this purpose, we mostly focus on two recent works [8], [9]. In order to achieve high throughput, a two-dimensional pipelined architecture using bit vector algorithm is proposed in [8]. However, due to high number of pipeline stages, this design introduces too much latency for the incoming packets in lookup. As a result, we used the idea of caching frequently used rules in a fast and separate lookup engine to decrease this latency. Caching popular rules in network classifiers is not new. However, compared to traditional IP networks, for newly-emerged SDN classification, there are multiple requirements that must be considered such as partial overlapping. In [9] a different caching scheme for SDN OpenFlow is proposed by taking these concepts into account.

2.4.1 Bit Vector Based Pipelined SDN Flow Table Implementation on FPGA

Among hardware packet classification algorithms, bit vector is a specific technique, where a lookup process for each field or a partition of field in a packet header is performed separately. The result of each lookup is *N-bit* vector. The bit-wise AND

operation for all fields give the final match result. Bit Vector algorithm is very suitable due to the abundant parallelism of hardware in FPGA. There are many FPGA implementations of BV algorithm [8], [43], [15], [28]. In [8], a different approach to BV algorithm is proposed: Two dimensional Pipelined Architecture of Bit Vector for SDN packet classification. Main reason of this architecture is to achieve very high throughput by decreasing the routing delays on FPGA. Hence, two dimensional pipeline architecture can be considered as a FPGA-optimization of BV algorithm.

This approach claims that instead of performing lookup in overall packet header in a complete rule set, modular units that are responsible for specific field bits and rules can be designed. This can be achieved by dividing packet headers consisting of L bits into small partitions called strides of s bits, and dividing total rule set of N rules into smaller n groups. After that, all these modular units that are responsible for a specific header stride and group of n rules can be connected in a pipelined fashion in both horizontal and vertical direction. The block diagram of this proposed approach can be seen in Fig. 2.4. For example, dividing N rules into small groups constitutes the vertical pipeline, whereas diving header bits into strides constitutes horizontal pipeline.

The query operation is performed by firstly splitting the header bits into constant *s-bit* strides. After that, a lookup is performed on the first processing element (PE), which is shown as PE(0,0). The result of this lookup process is *n-bit* BV. Thereafter, this BV result is sent to next PEs in both horizontal and vertical direction. At this time, these next PEs perform lookup for their related header stride and rule set and apply bit-wise AND operation with the previous *n-bit* BV. This process continues until PE(N/n,L/s) provides the final bit result. Therefore, *n-bit* BVs travel in both horizontal and vertical direction throughput the two dimensional pipeline architecture.

The lookup process in each PE is carried out using data-associative element. In [8], these elements are designed using logic gates. Fig. 2.5 illustrates two PEs with two data-associate elements for a rule set consisting of 4 -rules with one Header(7:0). Stride size s is selected as 4-bits. Therefore Header(7:0) is divided into two 4-bit strides; Header(7:4) and Header(3:0). When R_1 with '0101_1100' is inserted, the contents of data-associate memories are changed depending on the stride value. For

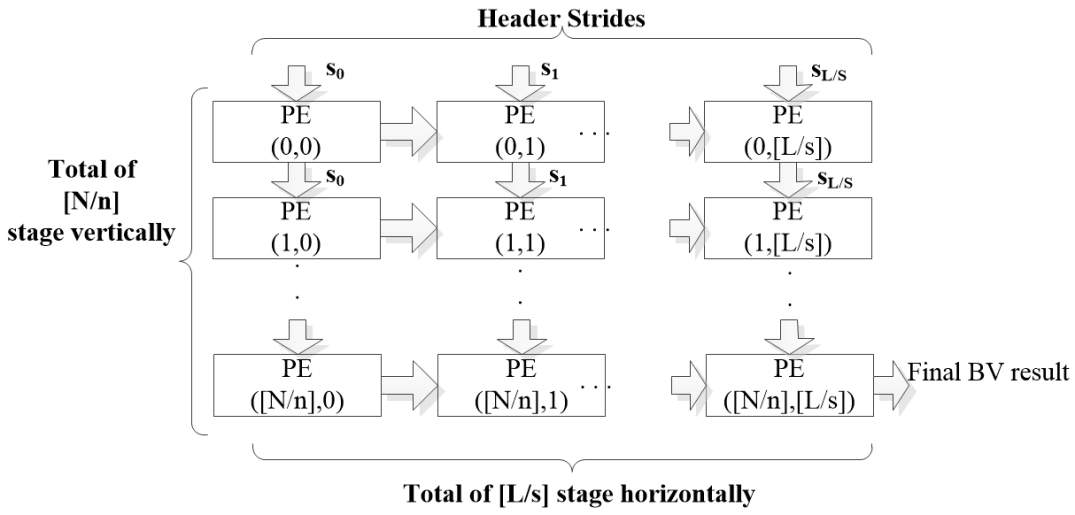

Figure 2.4: Connection of PEs in pipelined BV architecture

example, since Header(7:4) of R_1 is '0101', the second bit (R_1 bit) at the address '0101' is set to 1 for the first data-associative memory. Similarly, the second bit at the address '1100' is set to 1 for the second data associative memory for Header(3:0) of R_1. As observed, when a rule with wildcard match is inserted, then, corresponding bits at all addresses are set to 1. For example, Bit 3 (R_3 bit) at all addresses is set to '1' due to wildcard match of R_3 Lookup operation is mainly searching the '1's in data associative memories for the incoming strides. For example, when a packet with '0101_1100' arrives, BV result of first and second element will be '1010' and '1010' respectively. Bit-wise ANDing these two BVs will result in '1010' which means that packet with header of '0101_1100' has match with R_1 and R_3 at the same time. If priority-based classification is utilized, then a priority encoder can be utilized to select only one of the rules. In [8], this is achieved by designing a rule-decoder entity inside each PE.

Two dimensional architecture in [8] is implemented in high performance Xilinx Virtex 7 FPGA. Due to pipelining connections, the routing delays are decreased significantly and 324 MPPS peak throughput is sustained. However, dividing rule set and header bits into smaller parts leads to a significant increase in latency. This is because, in order to obtain a final BV result, this architecture has to wait for a total of (N/n + L/s) cycles. As n and s decreases, routing delays are further minimized, but, in turn, latencies are further increased.

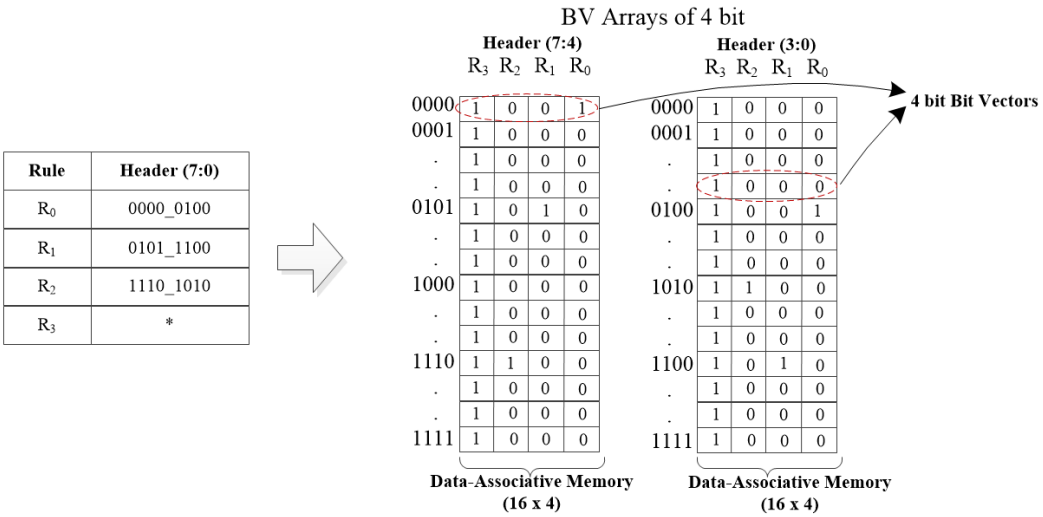

Figure 2.5: Internal structure of data-associative elements

As a result, decreasing the latencies introduced by this pipelined BV architecture is required. In FASST, this is solved by using a separate, fast classification engine, SRAM-based TCAM. TCAM is used as a cache in our application, where popular rules are dynamically stored.

2.4.2 Rule Caching Algorithms By Exploiting Temporal Locality For Flow Tables in SDN

In [9], a caching scheme is proposed in order cache *popular* rules for OpenFlow enabled SDN packets. Rather than traditional caching solutions where individual rules or compressed-version of rules are cached, in this scheme, a rule dependency graph is generated. In this dependency graph, rules are cached with their dependent rules in order preserved the network semantics.

Different from IP networks where there are only containment relations between IP fields, in [9], partial overlap conditions are also considered during computing rule dependencies. Since there are distinct header fields (15-tuple headers) in SDN packets, there exist many combinations of partial overlaps between rules. An example of rule set and the corresponding dependency graph is illustrated in Fig. 2.6. In this rule set, there are 6 rules with 4 distinct header fields. Furthermore, R_1 has higher priority and

R_6 has the lower priority.

Rule	Match	Action
R1	0000	Action
R2	11**	Action
R3	000*	Action
R4	1*1*	Action
R5	00**	Action
R6	10*1	Action

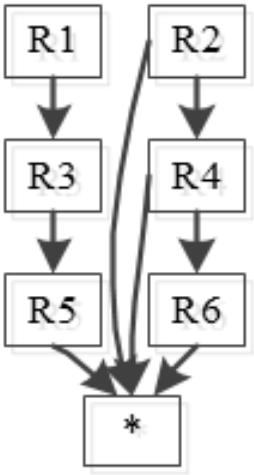

Figure 2.6: Rule set with indirect dependencies

For example, partial overlaps can be observed between R_2 and R_6. Direct overlap condition between R_2 and R_4 can be detected simply by computing intersections of fields between these rules (R_4 is dependent on R_2). However, the matches of R_2 and R_6 do not intersect. However, if a dependency graph is constructed, then, indirect dependency between R_2 and R_6 should be detected. This is because the matches of R_4 and R_6 are dependent on R_2. While caching popular rules; R_2, R_4 and R_6 are stored in cache as a group.

The requirement of caching dependent rules comes from the priority based classification in SDN. Consider that R_6 is popular rule and indirect dependencies are not computed. Therefore, only R_4 and R_6 are stored in cache by computing only direct

dependencies. A packet with header 1*1* that should match R_2, would match R_4 in this case, which causes a malfunction in classification.

In [9], computing rule dependencies are carried using a recursive algorithm. The input of the algorithm is a rule set with different priorities, and the output of the algorithm is the prioritized list of ordered rules as seen in Fig. 2.6. Algorithm starts to generate graph by firstly sorting all rules based on priorities. After that, for each rule, it checks the intersection conditions recursively. For example, firstly R_6 is captured and intersection operation is performed, and R_4 is detected. From that point, the intersection checking continues from R_4 to find R_2.

There are additional algorithms in order to increase the effectiveness of the caching scheme in [9]. For example, in order to provide caching on large flow tables, splicing rule dependency graph into smaller groups are proposed. Moreover, at each rule insertion, overall rule dependency is graph is not cleared. Instead, incremental updates are performed.

The evaluation of the caching scheme in [9] indicated that, for network behaviors with strong temporal locality, up to 97% cache hit rates can be achieved due to caching dependent rules. This outcome is very useful for our proposed architecture FASST that aims to decrease overall latency of packets during classification.

2.4.3 Design Developments of FASST Compared To Related Works

At this point, it is important to specify that we did not blindly apply these existing works to our proposed architecture FASST by simply combining them, which is impossible to achieve indeed. The adaptation and developments of the existing proposed works in FASST can be summarized listed as in the following:

- The hardware platform for FASST and the work in [8] are completely different; hence, the design stages of two dimensional bit vector architecture show a lot of variations. For example, in [8], overall architecture is developed using logic gates, which is simpler to manage and control to increase throughput. However, in FASST, embedded memory blocks are used, which gives us a more scalable

solution for increasing flow table size and packet header size. The design steps of managing these embedded memory blocks are more sophisticated.

- The partitions of rules and headers in FASST and [8] are completely different. In other words, while [8] uses a one level design hierarchy to design overall two dimensional pipeline architecture by partitioning rule set size and headers, we developed FASST by designing a two-level hierarchy. This means that we did not partition the rule set size and headers in the same design level. At top level, we split the rule set size first by dividing overall rule set into 32 rules, and at second hierarchy level, we split the header bits into different stride numbers. This modular and two level design provides us a more manageable and scalable solution in terms of flow table sizes.

- While [8] uses a constant *stride size s* for header partitions throughout the design, FASST utilizes different stride sizes, which are determined and configured depending on the field type.

- FASST extracts locality information from two level bit vector design, which is not applied in [8].

- In [9], the proposed caching scheme named *CacheFlow* is a complete software-based approach. In other words, prototype and evaluations of the caching algorithms are all carried out in high-level software environment. There is no design step for hardware switches. A commercial hardware switch Pica8 is utilized to be used as a cache. However, in FASST, all operations of lookup and caching popular rules are performed on o single-hardware, which is Altera FPGA. A SRAM-based TCAM is designed to be use as a cache in FASST.

- Different from [9], in which there is no software memory limitation, we adopted and optimized caching algorithms to be able to fit a System on Chip (SoC) design in FASST. Since we use a soft processor inside FPGA by designing a SoC platform, there are limited embedded (on-chip) memory blocks that this soft processor can utilize as program memory. For example, we did not support incremental updates and operation of splicing long chains presented in [9].

- Moreover, we did not apply recursive algorithms provided in [9] due to the fact that recursive operations of the algorithm for SDN 15-tuple headers in-

crease the stack memory utilization exponentially in our SoC design. Instead, we changed these algorithms to iterative versions. This enables us to have more stack memory area at the expense of longer generation time of the rule dependency graph. However, execution time on rule dependency generation is not our focus in this thesis. As a result, we generated the rule dependency graph by considering each rule separately, which is different from [9]. For example, for the example rule set given in Fig. 2.6, indirect dependencies between R_2 and R_6 are not found at the time of generating rule dependency graph. In our graph, R_6 is only connected to R_4 and R_4 is only connected to R_2. However, adapting a graph searching algorithm, the indirect relation between R_2 and R_6 can be observed. This is because, we deployed a depth-first search on dependency graph, which provides a search on R_6, R_4 and R_2 in order. Hence, all dependent rules including direct and indirect dependencies are detected during graph traversal.

- In [9], the details of SDN rule headers are not given in detail due to using high-level software environment. In FASST, all design steps are provided regarding partial overlaps and IP containment relationships during lookup process.

- In [9], when a packet arrives to be classified, lookup operation is only performed on cache. If it is found in cache, the result is provided. On the other hand, if packet does not match any rule in cache, hardware switch sends this packet to a software agent, which stores overall rule set in a software environment. This causes high trip delays for packets when packets are actually found in software agent. In FASST, we perform parallel lookup process in BVM and TCAM Cache, which has no effect on overall design when a packet is not found in cache.

CHAPTER 3

FASST: FAST SCALABLE SDN TABLE

3.1 FASST Overview

SDN switches look up every packet in a flow table of rules that are defined by 15 fields in the packet headers at 10s of Gbps line rates. Our proposed hardware architecture FASST always achieves a very low average latency as well as high throughput. The core idea in FASST is to exploit the well known temporal locality in the network traffic and to offer a tradeoff between average look up latency and power consumption.

FASST hardware architecture can be deployed in SDN applications where ultra-low latency and high bandwidth are the primary concern such as data centers. Compared to previous studies, there is an increasing interest in latency for mainstream applications in data centers with tens of thousands of servers. This is because a significant amount of computing, storage and communication is shifting to current data centers [44]. Moreover, ultra-low latency applications in data centers such as high-frequency trading, high performance computing require operations of request-response loop and overall operation is completed in case all requests are satisfied [45]. Therefore, end-end latency for these applications in data centers should be very low to provide the quality of service. Furthermore, achieving very low latency will have a significant positive effect for existing applications such as Google's statistical machine translation [46]. Improving the latency performance will enable these applications to be developed in a faster and more scalable way as well as more data-exploration in real time [47].

To this end, we define the entire flow table as a set of rules $\mathcal{R} = \{R_1, R_2, \ldots\}$ with

size (number of rules) $|\mathcal{R}|$. We define a time varying set of frequently accessed rules in this flow table $\mathcal{F} \subset \mathcal{R}$ with size $|\mathcal{F}|$. By the results of [9, 20] we assume $|\mathcal{F}| << |\mathcal{R}|$. The throughput, latency and power metric values for packet look-up as defined in Section 2.2.2 decline with increasing size of flow table. To this end, one can adopt a fast lookup engine that stores $\mathcal{F}$ returns most of the match results and $\mathcal{R}$ is stored in a slower engine for infrequent references whenever required.

FASST realizes this cache-like approach with two hardware-based classification engines named Bit Vector Module (BVM) and Ternary Content Addressable Memory (TCAM) as seen in Fig. 3.1. Bit Vector is a decomposition-based architecture that performs parallel look-ups on each individual field, and it provides a scalable and high throughput classification solution in recent work [8], [43], [15]. In our architecture, BVM stores $\mathcal{R}$ in the table as bit vector arrays and produces partial matches between input strides and corresponding bit arrays. Moreover, 2D pipeline architecture in BVM is implemented with particular updates similar to the recent work to concatenate the partial results at high clock rates [8]. The latency of the BVM is t_{slow} clock cycles because of pipeline stages in BVM. This latency increases with the number of pipeline stages, which are determined by the number of rules $|\mathcal{R}|$ and header size of the rules. The FASST TCAM is deployed as a flow cache for a fast look up in order to decrease the average latency of SDN packets. In this sense, TCAM cache only stores $\mathcal{F}$ and produces matches in t_{fast} clock cycles latency, where $t_{fast} << t_{slow}$. In our current implementation, $t_{slow} = 80$ cycles and $t_{fast} = 3$ cycles. Both BVM and TCAM return Rule IDs as match results.

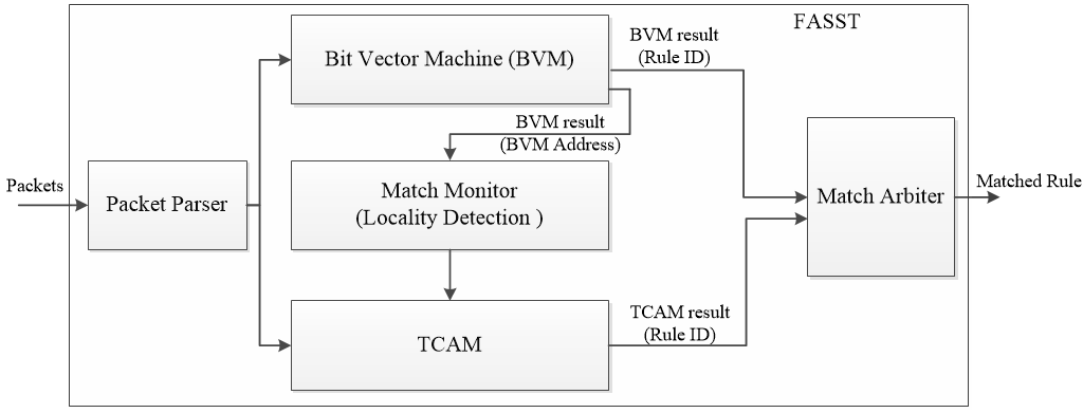

Figure 3.1: Block diagram of FASST architecture

3.2 FASST Operation

We represent SDN rules in FASST as <Rule ID><List of Header fields to match><Rule Priority>. The input to FASST is the extracted list of header fields of the incoming packet. If there is at least one matching rule to the input header list, we call this result a *positive*. In this case, the output is the Rule ID of the matching rule with the highest Rule Priority. We assume that the corresponding actions for the matching rule are stored in an on-chip RAM which can be accessed with a simple address mapping. If there are no matching rules the result is a *negative*. In this case, the output is a default value.

Multi-field packet classification in FASST begins with a Packet Parser (PP) functional unit. PP extracts different types of packet headers up to 15-fields, which are compatible with Openflow protocol. By the definition in Section 2.1.2, we define *flow* F_j as the group of packets that match to Rule ID j. The ith packet arriving on F_j is denoted $P_{i,j}$.

Following the field extraction by packet parser, the list of headers is processed by both BVM and TCAM. It is important to note that TCAM might return a false negative result for a given packet as it only stores $\mathcal{F}$ while BVM does give any false negative result as it stores $\mathcal{R}$. To this end, Match Monitor (MM) continuously traces BVM match results and dynamically updates TCAM with the current $\mathcal{F}$.

While BVM result arrives in t_{slow} clock cycles, TCAM cache produces a result in t_{fast} clock cycles. The lookup is finished in t_{fast} clock cycles for packets with positive result from the TCAM. If Rule ID j is stored in TCAM, all packets on flow F_j are matched by TCAM in t_{fast} clock cycles. The corresponding match results for the same packets to Rule j from BVM arrive in t_{slow} clock cycles. The Match Monitor (MM) eliminates the duplicate late positive results from BVM at the output. Note that all packets of F_j are matched by TCAM with the same latency and their order is preserved at the output. The lookup finishes in t_{slow} clock cycles with a positive or negative result for packets that did not produce a positive result from the TCAM.

3.3 Two Dimensional Bit Vector Machine

FASST utilizes a Bit Vector Machine (BVM) in order to perform parallel lookups on header fields for each incoming packet. BVM is implemented in a 2-dimensional pipelined fashion in order to minimize the routing delay and achieve high clock rate in hardware [15]. Two dimensional pipeline architecture is observed to provide efficient QoS regarding rule set size and the number of header fields at high clock rates [8]. Vertical pipeline enables BVM to become scalable with respect to rule set size $\mathcal{F}$, whereas horizontal pipelining provides scalable solutions for high number of fields in a multi-field header.

In FASST, BVM is designed as a two-level hierarchy of pipelines, which is different from [8]. Two level hierarchy enables FASST to support different rule set size in a more modular way. In other words, a modular PE in [8] only produces the match result for exactly 1 rule, and overall 2D pipelined architecture is constructed by connecting these 1-rule PEs. If rule set size is required to increase or decrease, then reconfiguring the connections between these PEs take time at top design. However, in FASST, PEs are gathered in a pipelined manner to create a higher level PE, which is responsible for the match results of more rules. At top design, these higher level PEs are also connected as pipelined manner and utilized. As a result, changing the rule set size $|\mathcal{R}|$ is more easy by simply adding and removing higher level PEs. In FASST, lower level PEs are named as *Stride BVM* and higher level PEs are named as *Rule BVM*. Each level has its own 2-dimensional architecture. Top level design consists of 2 horizontal and 32 vertical pipeline stages as illustrated in Fig. 3.2. At top level, there exists a 16 *Rule BVM*. Moreover, there is another 2-dimensional pipeline architecture inside each *Rule BVM*, which constitutes the second hierarchy. Each *Rule BVM* is connected to two priority encoders (POEnc) and an another *Rule BVM*. Since each *Rule BVM* has also 2 stage vertical internal pipeline, total vertical pipeline stages are equal to 32, which can also be seen by observing the number of *POEnc*. *Rule BVM* is a modular architecture and supports a look up operation of 32 flow entries among total rule set size $\mathcal{F}$. There are two match outputs for 32 rules stored in a *Rule BVM*: match result for the former 16 rules, and match result for the latter 16 rules. FASST has $n = 16$ for our implementation where n is the number of rules that

can be connected in a pipelined fashion in both horizontal and vertical direction as defined in 2.4.1. The algorithmic latency between these two outputs is a single clock cycle due to internal pipeline architecture of *Rule BVM*. For example, consider $|\mathcal{R}|$ rules are stored in FASST BVM. If a *Rule BVM* at top hierarchy is responsible for the classification of rules between (R_i, R_{i+31}), then first match result of a modular Rule BVM is for the rules (R_i, R_{i+15}) and asserts at *t* clock cycle, and the second match result is for the rules (R_{i+16}, R_{i+31}) and asserts at *t+1* clock cycle. Each of these match results is a parallel bit vector of length 16, where each bit denotes a match condition for a particular rule entry if the value is logic 1. The structure of bit locations in a bit vector is similar to Fig. 2.5 in Section. 2.4.1. Since *Rule BVM* has two match outputs, there are two *POEnc* to report highest-priority match associated for that *Rule BVM*. Therefore, each *POEnc* is responsible for 16 rule entries. The match result of *POEnc* is 16-bit vector, similar to *Rule BVM*. However, since *POEnc* finds the highest priority match among multiple match results, only a single bit can be logic 1 at the output of *POEnc*. Each *POEnc* gives the result to another *POEnc*. As a result, priority encoding is also carried out in a pipelined manner in our work.

In FASST hardware architecture, the number of rules of which each *POEnc* is responsible can be decreased and increased. However, the change of the rule number in each *POEnc* leads to a trade-off between throughput and latency due to hardware considerations in pipelined design. In other words, if the number of rules in each *POEnc* is increased to 32 instead of 16, then the maximum achieved throughput in FASST BVM design decreases due to the fact that the combinational processes carried out in each *POEnc* can only meet the timing requirements in a lower frequency. On the other hand, since the overall number of *POEnc* units are decreased at top level hierarchy , the latency of FASST BVM is decreased. Similarly, if the number of rules of which each *POEnc* is responsible is decreased, then priority encoding can be carried out within a narrow clock period, which results higher clock rates. However, in this case, pipeline latency in FASST BVM is increased based on the overall number of *POEnc* units.

Internal architecture of a *Rule BVM* consists of 47 horizontal and 2 vertical pipeline stages. For each direction, a total of 94 modular units called *Stride BVM* is implemented as seen in Fig. 3.3. *Stride BVM* is the basic unit of classification process

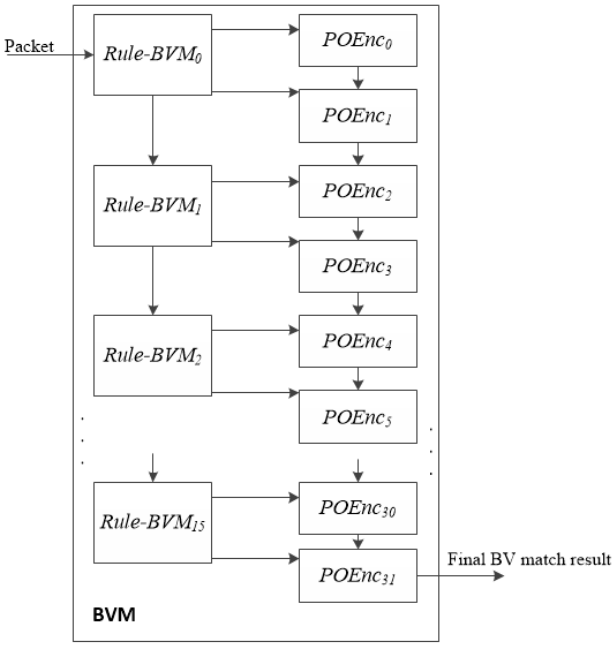

Figure 3.2: 2-D top level architecture of BVM

inside BVM. Two horizontal pipelines are responsible for lookup process of the first 16 rules and the second 16 rules, respectively. We can use a coordinate system to map each *Stride BVM*. For example, for $RuleBVM_i$, *Stride BVM* units at the upper horizontal pipeline can be named as $RuleBVM_i(0,0)$, $RuleBVM_i(0,1)$ up to $RuleBVM_i(0,46)$. Similary, *Stride BVM* units at the lower horizontal pipeline will be $RuleBVM_i(1,0)$, $RuleBVM_i(1,1)$ up to $RuleBVM_i(1,46)$.

The input to each *Stride BVM* is a header stride of *s* bits, as defined in Section. 2.4.1. In FASST BVM, *s* can be 3, 4, 6 or 8 bits depending on the header type. For example, for 6-bit *ToS* field, a single *Stride BVM* is enough to perform look-up. On the other hand, splitting 64-bit *Metadata* field into 8 subfields is required. Therefore, there are identical 8 *Stride BVM* units for *Metadata* field and *s* is 8 bits for each of these *Stride BVM* units. As a result, the focus of each *Stride BVM* on horizontal direction is to look-up for distinct patterns of header strides.

Stride BVM units at the *same* vertical stage are responsible for the same header stride patterns. However, the rule entries which they are responsible for are different. For instance, 47 *Stride BVM* units at upper horizontal pipeline produce parallel bit vectors for a 15-tuple packet P for rules (R_i, R_{i+15}). After single clock cycle, 47 *Stride BVM*

units at lower horizontal pipeline produce bit vectors for rules (R_{i+16}, R_{i+31}) for the same packet P. For example, $RuleBVM_i(0,0)$ and $RuleBVM_i(1,0)$ performs look-up for header stride s_0 in Fig. 3.3. However, the lookup time of $RuleBVM_i(1,0)$ is delayed with single clock cycle from the lookup time of $RuleBVM_i(0,0)$.

Stride BVM has access a data associative on-chip Random Access Memory (RAM) to generate a match or no-match result. Each result of *Stride BVM* is a parallel bit vector of length 16, where each bit indicates the corresponding rule entry number. The local bit vectors of each *Stride BVM* are bit-wise ANDed with bit vectors provided from previous *Stride BVM*. As a result there is total of 47 logical AND gates in horizontal pipeline.

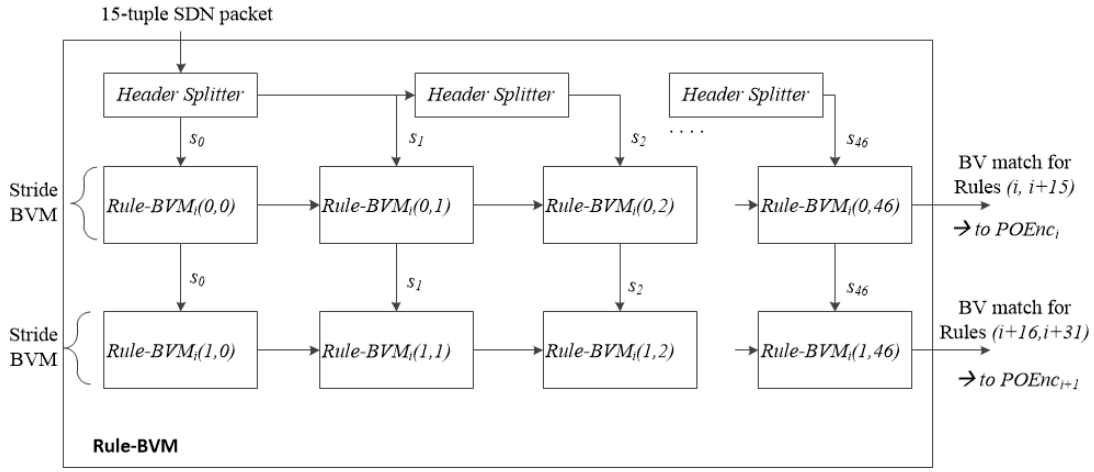

Figure 3.3: Internal pipelined architecture of Rule BVM

In FASST, BVM stores and performs look-up for 512 flow table entries; hence $|\mathcal{R}|$ is 512. Considering 2-dimensional pipeline in two-level hierarchy, overall design can be labeled as $RuleBVM_l(i,j)$ and $POEnc_k$; where *l* labels the *Rule BVM* number at top level for $0 \leq l \leq 16$ and $0 \leq k \leq 31$. Note that, if $RuleBVM_l(i,j)$ is connected to $POEnc_k$ and $POEnc_{k+1}$, then *l*=*k*. Moreover, *i* and *j* denotes 2-dimensional coordinate system for *Stride BVM* inside *Rule BVMs*; where $0 \leq i \leq 1$, $0 \leq j \leq 46$.

Consider two SDN packets with 15-tuple headers arrive to FASST BVM. The total number of header bits in a 15-tuple packet is 356 bits as defined in Section. 2.1.2. We split 356 bit header into subfields of *s* bits, where *s* can be 3, 4, 6, and 8. This splitting

of header bits results in 47 horizontal stages in each *Rule BVM*. Hence, if we label the subfields as s_i, $0 \leq i \leq 46$, the look-up process flow inside BVM for packets P occurs as illustrated in Table. 3.1 with respect to clock cycles. For this packet flow, we assume that a new packet P_i arrives to FASST BVM at each clock cycle.

Table 3.1: Packet tracing flow inside BVM for 512 rules

Clock Cycle	Look-up Phase
n	s_0 of P_0 arrives to $RuleBVM_0(0,0)$ s_1 of P_0 passes to $RuleBVM_0(0,1)$ by $RuleBVM_0(0,0)$ s_0 of P_0 passes to $RuleBVM_0(1,0)$ by $RuleBVM_0(0,0)$
n+1	BV match result from $RuleBVM_0(0,0)$ for s_0 of P_0 s_0 of P_1 arrives to $RuleBVM_0(0,0)$ s_1 of P_1 passes to $RuleBVM_0(0,1)$ by $RuleBVM_0(0,0)$ s_0 of P_1 passes to $RuleBVM_0(1,0)$ by $RuleBVM_0(0,0)$
n+2	BV match result from $RuleBVM_0(0,1)$ for s_0 ,and s_1 of P_0 BV match result from $RuleBVM_0(1,0)$ for s_0 of P_0 BV match result from $RuleBVM_0(0,0)$ for s_0 of P_1 s_0 of P_1 passes to $RuleBVM_1(0,0)$ by $RuleBVM_0(1,0)$
n+48	BV match result from $RuleBVM_0(0,46)$ for s_0, s_1 up to s_46 for P_0 BV match result from $RuleBVM_0(1,45)$ for s_0, s_1 up to s_45 for P_0 BV match result from $RuleBVM_1(0,44)$ for s_0, s_1 up to s_44 for P_0
n+49	BV match result from $POEnc_0$ for the rules (R_0, R_{15}) using BV from $RuleBVM_0(0,46)$ for P_0
n+50	BV match result from $POEnc_1$ for the rules (R_0, R_{31}) using BV from $POEnc_0$ and $RuleBVM_0(1,46)$ for P_0
n+51	BV match result from $POEnc_2$ for the rules (R_0, R_{47}) using BV from $POEnc_1$ and $RuleBVM_1(0,46)$ for P_0
n+80	BV match result from $POEnc_{31}$ for the rules (R_0, R_{511}) using BV from $POEnc_{30}$, and $RuleBVM_{15}(1,46)$ for P_0

Final result obtained from the last *POEnc*, which is $POEnc_{31}$, is 16-bit parallel bit vector. However, in order to determine the Rule ID of the matched rule, we need group information of the matched bit in parallel bit vector. In other words, since we split 512 rules into small groups of 32 rules, a 32 bit register is defined. Each bit in this register defines the group number. For example, bit 0 in this register denotes (R_0, R_{15}), whereas bit 1 denotes (R_{16}, R_{31}). This register is set by all *POEnc* units during lookup process with a pipelined manner. If current *POEnc* observes the

highest priority match for its rule range, then it clears all bits in this register, and sets the corresponding bit, which it is responsible for. For this purpose, at the end of $POEnc_{31}$, FASST provides 16-bit match vector, and 32-bit group register information. If final bit vector $0x0008$ and final register content is $0x00000002$, then the *address* of the matched rule is determined as 19. Using this address, we can simply figure out the Rule ID of the matched rule by using a <Address><Rule ID> on chip memory. Hence, the output of BVM is Rule ID of the matched rule.

3.4 Ternary Content Addressable Memory (TCAM)

In FASST, TCAM is utilized to generate match results for frequently accessed rules over the recent past. Due to TCAMs superbly fast $O(1)$ look-up, the match result for these frequently accessed rules is produced in a 3 clock cycles, $t_{fast} = 3$ cycles. The implementation details of FPGA-based TCAM in FASST is given in Fig. 3.4.

TCAM design has similar implementation structure with BVM except for 2-dimensional pipeline architecture. It can be considered as a single ***Rule BVM*** with no vertical or horizontal pipelining. Instead of passing input strides s_i, $0 \leq i \leq 46$ to ***Stride TCAM*** units in a pipelined fashion, each input stride s_i goes to all ***Stride TCAM*** units concurrently for lookup process. In other words, $StrideTCAM_0$ outputs the match result for s_0 at the same with $StrideTCAM_{46}$, which outputs the match vector for s_{46}. In a ***Rule BVM***, $RuleBVM_0(0, 46)$ produces match vector for s_{46}, 46 clock cycles after $RuleBVM_0(0, 0)$ outputs the match result for s_0.

Due to exploiting extreme amount of parallel computation during TCAM look-up, the number of stored rules are kept as minimum as possible in order to avoid clock rate deterioration. Therefore, at the first stage of FASST design, TCAM supports 32 SDN flow entries, which means $|\mathcal{F}| = 32$. While a single ***Rule BVM*** searches over 16 rules with 2-dimensional pipeline architecture, TCAM cache searches over 32 rules with no pipelining at all. Moreover, different from ***Rule BVM***, TCAM has its own internal priority encoder ***POEnc-TCAM*** to return highest priority match.

When a packet arrives FASST, it is routed to both BVM and TCAM. At this point,

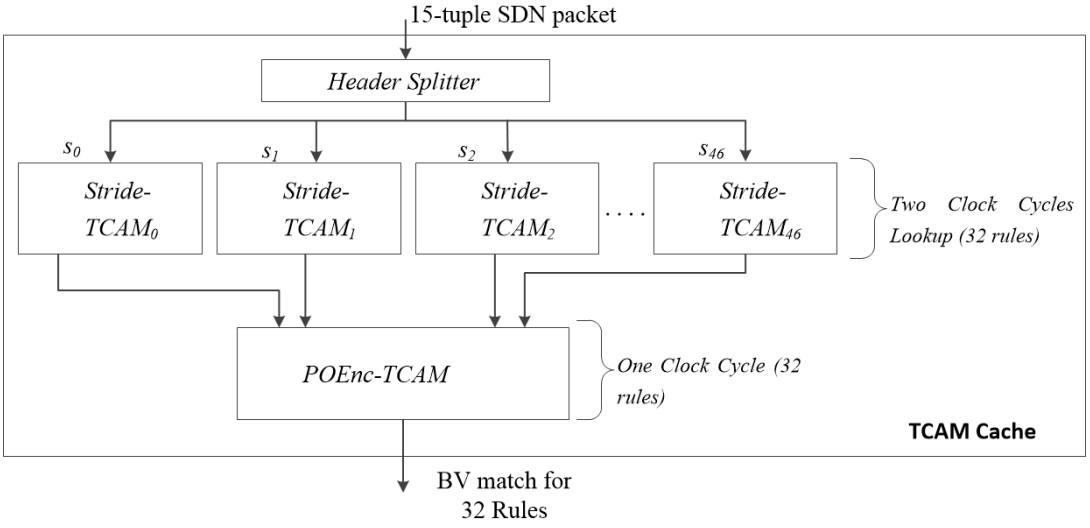

Figure 3.4: Internal parallel architecture of TCAM

TCAM splits the 15-tuple (356 bits) header into 47 subfields with same stride configuration of *s* as *Rule BVM* and sends all of these distinct subfields to *Stride TCAM* units concurrently. After two clock cycles, bit vectors from each *Stride TCAM* assert to report matching status over 32 rules. After one clock cycle, *POEnc-TCAM* generates 32-bit highest priority match vector. As a result, end to end algorithmic latency of TCAM is 3 clock cycles.

Since there is no rule splitting as in BVM (splitting $\mathcal{R}$ to groups of $n = 16$), there is no need for a register for group information. 32-bit final parallel bit vector result provided by *POEnc-TCAM* gives directly the matched rule address. Using a similar <Address><Rule ID> embedded memory, FASST provides Rule ID of the matched packet in TCAM.

3.5 Match Monitor (MM) – Locality Detection

Match monitor (MM) measures the network traffic over the recent past by analyzing matched results asserted by BVM continuously. Note that, the input to MM is the BVM address of the matched rule entries, but BVM output is the Rule ID of the matched packets. However, as explained in Section. 3.3, 16-bit final bit vector and 32-bit group information are also asserted by BVM, which helps to find BVM address of matched rule. Actually Rule ID is found by using BVM address. As a result of

this measurement, MM detects the most popular BVM addresses of rules by exploiting well known temporal locality. Instead of traditional caching solutions, MM also computes a dependency chain similar to the method as introduced in Section 2.4.2 to respect rule dependencies between these frequently accessed rules. As a result, MM caches the rules together with their dependencies.

While IP route caching with rule dependencies is considered on earlier works, IP prefixes only have simple containment relationships [48], [49]. However, in OpenFlow enabled SDN, partial overlaps may exist due to any match combination of multiple fields including exact match and wildcard match, which can result in indirect dependencies. Naga *et al.* proposes rule-caching algorithms for SDN called *Infinite CacheFlow*; however, implementation and evaluation of these algorithms are carried out only in software environment [9]. Cache misses are directed to a external software agent that searches the entire flow table, which corresponds to BVM in FASST. For low cache hit rates, this leads to long round trip delays for packets. MM in FASST implements an adapted and developed version of Infinite CacheFlow for hardware-based applications considering both containment relationships for IP fields and partial overlaps for other header fields.

Detailed block diagram for MM to handle locality detect and caching is shown in Fig. 3.5. The input to MM is BVM address of the matched entries. BVM address is 9 bits long in our case, since $|\mathcal{R}| = 2^9 = 512$. Locality Detection unit measures the traffic over a time window W_s that is used to catch legitimate changes in network load. The choice of W_s is extremely dependent upon the network characteristics. In other words, W_s should be long enough to prevent thrashing and short enough to adopt new changes [9]. Moreover, we employ another parameter named Threshold Size Thr used in locality detection. If incoming packets match a flow entry F_j k times with $k > Thr_s$ in BVM over the time window W_s, F_j is considered as a frequently used rule and BVM address j of F_j is stored in RAM-1 by Locality Detection. Note that Locality Detection unit stores BVM address j to RAM-1 instead of list of header fields to match.

Locality Detection unit only detects a specific number of frequently used rules due to limited storage capacity of RAM-1, which is $512x9$. Moreover, since FASST also

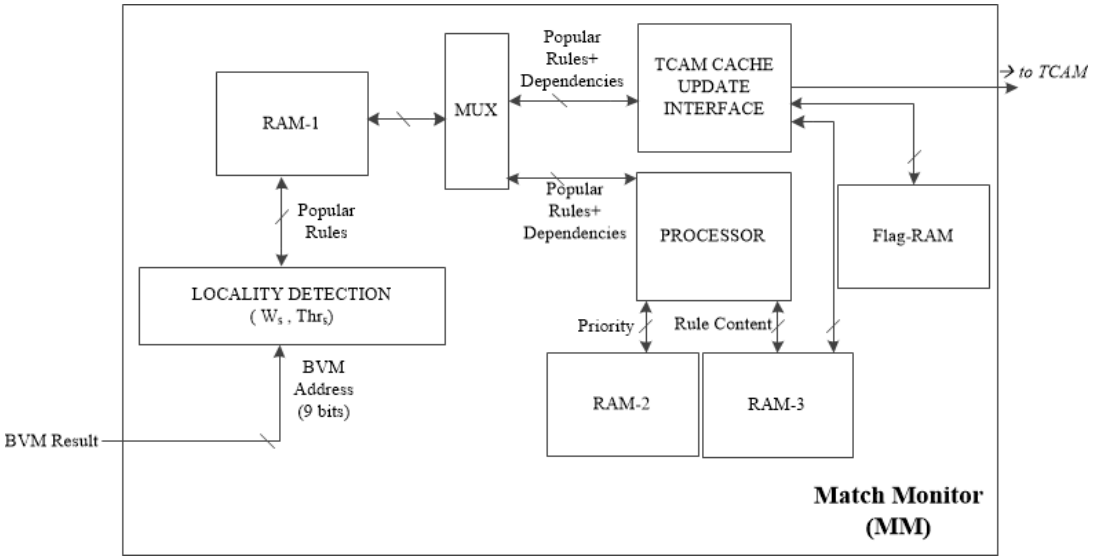

Figure 3.5: Conceptual design of Match Monitor

computes the rule dependency chains, *rule depth* of the detected popular rules should also be taken into consideration while maintaining rule detection. *Rule depth* is the maximum number of rules that are dependent on a popular rule. In FASST, this parameter is considered as constant, which is 7. This means that, FASST flow table can store SDN rule entries which have dependency chains with maximum depth 7. Flow entries in REANNZ research and education network for an SDN-enabled Internet eXchange Point (IXP) show that most dependency chains have depth 1 [50]. However, to be on the safe side, we split RAM-1 into subparts of 8-words, where each subpart stores a popular rule and its maximum dependent rules.

The organization of RAM-1 is depicted in Fig. 3.6. $Address_0$ and $Address_{511}$ are used for handshaking operations between Locality Detection, Processor and TCAM Cache Update Interface units for process scheduling. Detected popular BVM addresses of rules are written to $Address_i$, where $i = 8k+1$, for $0 \leq k \leq 31$, by Locality Detection Unit. After all popular BVM addresses are written to RAM-1, Locality Detection unit sets bit 0 at $Address_0$ to inform the completion of rule detection. At this time, processor continuously polls bit 0 at of $Address_0$. After completion of rule detection, processor reads popular BVM addresses from RAM-1, determines the corresponding dependent rules by inspecting BVM addresses of them, writes them back to RAM-1 again. Each BVM address of a dependent rule is written to the particular subpart of RAM-1 of the associated popular rule in a prioritized order. $Address_{257}$ to

$Address_{510}$ in RAM-1 are reserved addresses for future purposes.

The processor in MM has two functions in FASST: Generating a rule dependency graph for BVM rules as soon as rules are inserted to BVM, and searching the rule dependency graph in order to find rule dependencies among these rules at run time. Generating the rule dependency graph is an independent process from searching the graph. In other words, as soon as Packet Parser (PP) completes the writing of rule entries to BVM, processor starts to generate this graph. While generating the graph, the processor needs <Rule ID><Rule Priority><Rule Header Content> information. This information is provided by PP during rule insertion phase. RAM-2 stores <Rule ID><Rule Priority>, and RAM-3 stores <Rule Header Content>. While rules are written to BVM addresses, PP also extracts <Rule ID><Rule Priority> information of these rules at this time, and writes <Rule ID><Rule Priority> pairs to RAM-2 . BVM address is used as address bus of RAM-2 for the beginning. In other words, <Rule ID><Rule Priority> pair at $Address_0$ of RAM-2 is <Rule ID><Rule Priority> content of the rule located at $Address_0$ of BVM. Moreover, while PP writes the rule contents to BVM, it also writes them to RAM-3 at the same time. However, although rules are written to BVM as rule-expansion (expanding wildcard entries using data associative memory), RAM-3 stores the rules without expanding the rules. In other words, rule contents are stored in RAM-3 with the information of mask bits. Processor can read <Rule Header Content> from RAM-3 using BVM addresses later. As a result; content of RAM-2 and RAM-3 is provided to processor by PP at rule insertion phase. Before building dependency graph, processor also reads <Rule ID><Rule Priority> from RAM-2 and creates an internal array. This internal array stores 512 elements, and each element stores <BVM address>. The index of this internal array is Rule IDs of the rules. As a result, mapping of <Rule ID> to <BVM Address> is achieved, where index 0 means Rule ID=0, and content at index=0 is BVM address of the rule with Rule ID=0. Remember that address bus of RAM-2 is BVM address at the beginning. Therefore, <BVM address> and <Rule ID> information can be traced for this case. To this end, in order to generate a dependency graph, we have two on-chip RAM, RAM-2 and RAM-3, and an internal array.

For dependency graph generation, processor firstly sorts <Rule ID><Rule Priority> pairs in RAM-2 with descending priority order. In other words, after sorting process,

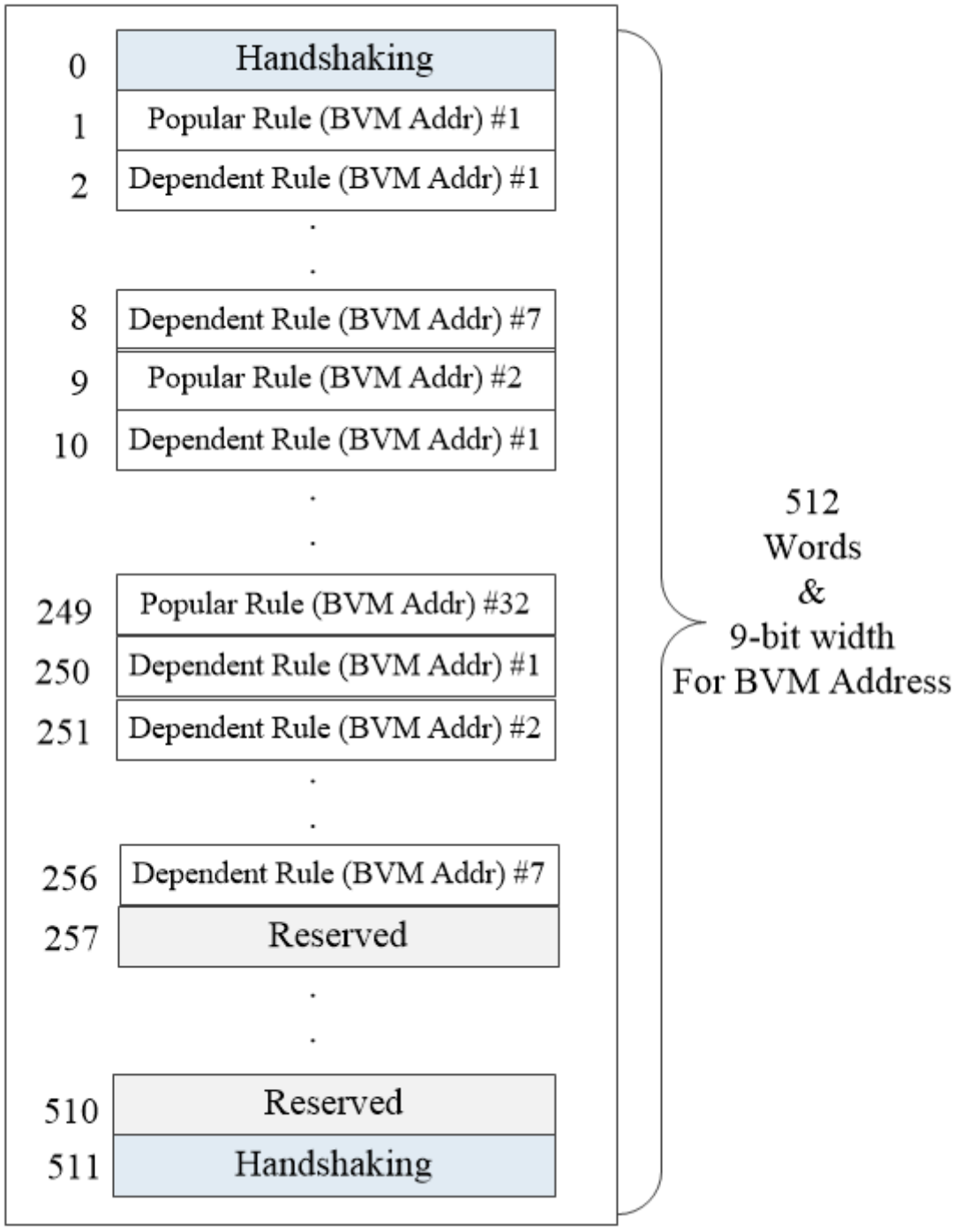

Figure 3.6: Data organization in RAM-1 to detect popular rules

$Address_0$ of RAM-2 stores <Rule ID><Rule Priority> pair with the least priority value (lower priority value means higher priority). Note that Rule ID can be anything at $Address_0$ of RAM-2 after sorting, because sorting is carried out on priority fields. Therefore, the order of RAM-2 is changed by processor after PP fills it. After that, processor starts to read $Address_0$ of RAM-2, where Rule ID of highest priority rule exists. Since processor knows Rule ID of highest priority at this time, now it can use internal array to find out BVM address of this rule. After obtaining BVM address, processor can read <Rule Header Content> from RAM-3. Because address bus of RAM-3 is actually BVM address. As a result, rule content of the rule with highest priority is obtained. After all contents are acquired, FASST creates a linked list data structure consisting of 512 nodes. Nodes are named from 0 to 511, where node 0 corresponds the rule at $Address_0$ of BVM. Therefore, each node in linked list denotes a rule in BVM and has <Rule ID>, <Rule Priority> and <Pointer Array of Dependent Rules>. Hence, while building dependency graph, firstly data at $Address_0$ of RAM-2 is read, and Rule ID of the highest priority rule is obtained. After that, using internal array and this Rule ID, BVM address of the highest priority rule is acquired. Thereafter, using this BVM address, rule content of the highest priority rule is read from RAM-3. This procedure continues for all rules with descending priority. After obtaining rule content at RAM-3, containment and overlap conditions are investigated to build dependency graph. Checking these relations is carried out between one rule against all other rules with lower priorities. For example, when $Address_0$ of RAM-2 is read, data of highest priority rule is obtained from RAM-3. After that, $Address_1$ to $Address_{511}$ of RAM-2 is read consecutively and rule contents are obtained. Checking is made between the rule for $Address_0$ and other rules from $Address_1$ to $Address_{511}$ at RAM-2. Note that we modify the rule dependency computation in [9] as presented in Algorithm 1 for feasible hardware implementation as discussed in Section 2.4.3.

Rule priorities in RAM-2 are used because input to rule-dependency chain algorithm are n rules R_1, R_2, ..., R_n, where rule R_i has a higher priority than R_j for $i \leq j$ as in [9]. The output of the algorithm is the prioritized list of n rules, where $n = 512$ in our case. If R_j is dependent on R_i, then the Rule Priority of R_j is smaller than R_i. The purpose of using rule contents in RAM-3 is to identify partial overlap or containment conditions between rule entries. RAM-3 stores the rule entry contents

in a specific format such that FASST hardware architecture determines dependencies on multiple fields easily. The format of RAM-2 and RAM-3 is illustrated in Fig. 3.7. In RAM-2, processor stores $< RuleID_i >< RulePriority_m >$ pairs for all rules in the descending priority order where $0 \leq i \leq 511$ and $0 \leq m \leq 511$. In Fig. 3.7, $RuleID_i$ has highest priority and $RuleID_k$ has lowest priority where $RulePriority_m \leq RulePriority_n \leq RulePriority_l$. In RAM-3, rule header fields are stored without expanding wildcards, instead, mask fields and prefix lengths are investigated to compute header overlaps. In other words, the contents in RAM-3 is a special copy of rule headers in BVM. Moreover, RAM-3 is organized in such a format that first address stores the rule content with BVM address =0. Both RAM-2 and RAM-3 have depths of $|\mathcal{R}|$, which is equal to 512 for our implementation.

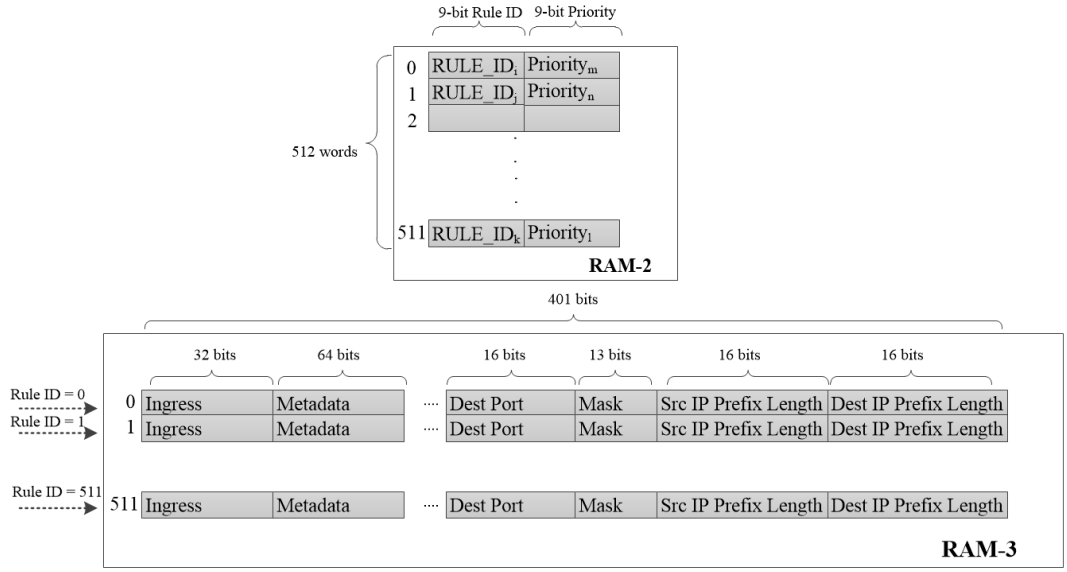

Figure 3.7: Data organization in RAM-2 and RAM-3 to observe dependencies

The pseudo code for rule dependency graph generation on processor can be found in Algorithm 1. Note that, Rule dependency graph is generated for one time for a static 512 rules. While generating dependency graph, processor first sorts 512 rules based on their 9-bit Rule Priority values from 0 to 511 in RAM-2, then it writes the prioritized list of the Rule IDs into RAM-2 as $< RuleID_i >< RulePriority_m >$ pairs. Thereafter, processor reads the Rule IDs from RAM-2 starting from the highest priorities at $Address_0$ and $Address_1$. Algorithm 1 considers dependencies for each

rule separately. As the algorithm proceeds, it determines whether the next rule read from RAM-2 has a dependency for the current highest priority rule for that round. If it does, then current highest priority rule is added to dependency list of the next read rule from RAM-2. Note that for dependency check, run contents are required. When RAM-2 is read, Rule ID is acquired. Furthermore; using internal array; BVM address is accessed for this Rule ID and using RAM-3; content of the rule at this BVM address is obtained. Dependency list of a rule is actually <Pointer Array of Dependent Rules> of each node. Remember that, nodes are named from 0 to 511 and node 0 denotes the rule at $Address_0$ of BVM. For example, consider that the rules are ordered from highest to lowest priority as R_0, R_1, R_2, ..., R_{511} in RAM-2. At the same time, linked list consisting of 512 nodes is created. Node 0 does not have to be R_0. Because, in this example, R_0 denotes the rule with highest priority, not the rule at $Address_0$ of BVM. Then, for the first round (line 14), processor in MM firstly reads BVM Addresses of R_0 and R_1 from internal array and header fields of R_0 and R_1 from RAM-3 and checks whether R_1 has dependent on R_0. At this case, assume that node m corresponds to R_0 and node n corresponds to R_1. In case of a dependency, R_0 is added to the first place of R_1's dependency list. In other words, first data at <Pointer Array of Dependent Rules> of node n is the pointer address of node m. After that, algorithm proceeds with R_2, R_3, R_4, ..., R_{511} in order to check if these prioritized rules have dependencies on R_0. Dependency checking between R_0 and R_{511} concludes the first round of the graph. Thereafter, processor reads the next highest priorities, which are R_1 and R_2 in our case, and tracks all rules from R_2 to R_{511} to observe dependencies of these rules on R_1. The same situation repeats until Rule IDs and header fields of R_{510} and R_{511} are read from RAM-2 and RAM-3 to perform a dependency check. Current FASST design does not support dynamic rule updates. However, if rule entries in BVM are required to change at run time, then processor takes the new rule contents from RAM-3, new priorities from RAM-2 and constructs the dependency graph from the beginning. Moreover, incremental algorithms to build the graph without destroying the previous one can be employed as in [9]. Support for dynamic rule update on FASST BVM and dependency graph will be designed as a future work.

While constructing the rule dependency graph, the core idea is to figure out the par-

Algorithm 1 Generating Direct Acyclic Graph for 512 Rules

1: **for all** i such that $0 \leq i \leq 511$ **do**
2: **for all** i such that $0 \leq i \leq 511$ **do**
3: **if** *Priority*$_j$ greater than *Priority*$_{j+1}$ **then**
4: store(*RuleID*$_j$, *Priority*$_j$) at address *j+1* of RAM-2
5: store((*RuleID*$_{j+1}$, *Priority*$_{j+1}$)) at address *j* of RAM-2
6: **end if**
7: **end for**
8: **end for**
9: int FirstRound =0
10: int SecondRound =1
11: int IndexBase =1
12: int RuleIDFirst =READ(RuleID at *Address(FirstRound)* at RAM-2)
13: int RuleIDSecond =READ(RuleID at *Address(SecondRound)* at RAM-2)
14: int BVMAddressFirst =Internal Array (RuleIDFirst);
15: int BVMAddressSecond =Internal Array (RuleIDSecond);
16: **while** FirstRound is less than 511 **do**
17: RuleFirst =READ(*Address(BVMAddressFirst)* at RAM-3)
18: **for all** $SecondRound$ such that $IndexBase \leq SecondRound \leq 512$ **do**
19: RuleSecond =READ(*Address(BVMAddressSecond)* at RAM-3)
20: **if** RuleSecond is dependent on RuleFirst **then**
21: Add RuleFirst to Dependency Chain of RuleSecond
22: **end if**
23: SecondRound =SecondRound+1
24: RuleIDSecond =READ(RuleID at *Address(SecondRound)* at RAM-2)
25: BVMAddressSecond =Internal Array (RuleIDSecond);
26: **end for**
27: FirstRound =FirstRound+1
28: SecondRound =FirstRound+1
29: IndexBase =FirstRound+1
30: RuleIDFirst =READ(RuleID at *Address(FirstRound)* at RAM-2)
31: RuleIDSecond =READ(RuleID at *Address(SecondRound)* at RAM-2)
32: BVMAddressFirst =Internal Array (RuleIDFirst);
33: BVMAddressSecond =Internal Array (RuleIDSecond);
34: **end while**

tial overlaps or containment relationships between the rules. Algorithm 2 shows the pseudo-code for the algorithm to determine such a relationship. Since FASST deals with 15-tuple OpenFlow-enabled SDN packet format, there can be matches on multiple header fields that result in indirect dependencies. In order to manage these match combinations, each field in header must be examined separately by analyzing mask fields. While 13 fields in our case have exact match and wildcard match, IP source and IP destination fields have prefix lengths. Checking the intersection for 13 fields separately is easy: For the same header field; if at least one of the header fields between two rule entries has a wildcard, then there is an intersection. Similarly, for the same header field, if both of the header fields do not have wildcard, but if they are equal, then there is also an intersection. However, regarding IP Source and IP destination fields, for two headers to have a non-empty intersection, both fields must have the same bit value at every position if this bit is not a wildcard. If a bit at a particular location is wildcard, then intersection for this bit always occurs. In header space analysis [3], single bit intersection rule is presented, which is illustrated in Fig. 3.8. Same rule is applied in FASST to find out intersection between IP fields. Hence, 32 bits IP fields are firstly encoded to 64 bits by mapping '0' = 01, '1' = 10 and 'x' =11. For this mapping, IP prefix lengths are used. After that, simply a bit-wise AND operation is applied on the encoded headers between two IP fields for two different rule entries. At the end of bit-wise AND operation, if two consecutive bits are "00" for *(bit0, bit1)*, *(bit1, bit2)*, ..., *(bit62, bit63)* pairs in 64-bit encoded pattern, then there is no intersection. "00" means a 'Z' in our encoded scheme, which means no intersection.

After processor constructs the dependency graph, it waits for Locality Detection unit to write BVM addresses of the popular rules to RAM-1, as explained earlier. After that, processor reads these BVM addresses and searches for the dependent rules for each of them. Note that since BVM addresses are written to RAM-1, processor use these BVM addresses in linked list. In other words, in linked list, nodes are named from 0 to 511 and node 0 corresponds to rule at $Address_0$ of BVM. As a result, when processor reads BVM addresses of popular rules in RAM-1, it can directly access <Pointer Array of Dependent Rules> field of the corresponding node in linked list. Depth-first search is used in MM processor to find out dependencies using <Pointer

Algorithm 2 Finding Dependecies Between Two Rules

```
1: bool InterSection =true
2: int RuleHeaderFirst, RuleHeaderSecond =0 {Header fields without IP fields}
3: int ExpIPSrcFirst, ExpIPSrcSecond=0 {63-bit expanded Source IP fields}
4: int ExpIPDstFirst, ExpIPDstSecond=0 {63-bit expanded Dest IP fields}
5: RuleHeaderFirst =READ(Address(BVMAddressFirst) at RAM-3)
6: RuleHeaderSecond =READ(Address(BVMAddressSecond) at RAM-3)
7: RuleIPSrcFirst =READ(Address(BVMAddressFirst) at RAM-3)
8: RuleIPSrcSecond =READ(Address(BVMAddressSecond) at RAM-3)
9: RuleIPDstFirst =READ(Address(BVMAddressFirst) at RAM-3)
10: RuleIPDstSecond =READ(Address(BVMAddressSecond) at RAM-3)
11: for all i such that 0 ≤ i ≤ 12 do
12:    if RuleHeaderFirst[i] is not masked and RuleHeaderSecond[i] is not masked
       then
13:       if RuleHeaderFirst[i] is not equal to RuleHeaderSecond[i] then
14:          bool InterSection =false
15:       end if
16:    end if
17:    for all i such that 0 ≤ i ≤ 63 do
18:       ExpIPSrcFirst =ExpIPSrcFirst AND ExpIPSrcSecond
19:       ExpIPDstFirst =ExpIPDstFirst AND ExpIPDstSecond
20:    end for
21:    for all i such that 0 ≤ i ≤ 63 do
22:       if ExpIPSrcFirst[i]==0 and ExpIPSrcFirst[i+1]==0 then
23:          InterSection =false
24:       else if ExpIPDstFirst[i]==0 and ExpIPDstFirst[i+1]==0 then
25:          InterSection =false
26:       end if
27:    end for
28: end for
```

b_i \ b_i'	0	1	x
0	0	z	0
1	z	1	1
x	0	1	x

Figure 3.8: Header space analysis for single bit intersection [3]

Array of Dependent Rules>. Due to using depth-first search, indirect dependencies are also computed. Consider three rules are R_1, R_2 and R_3, where priorities are $R_1 > R_2 > R_3$. If we assume R_2 dependency on R_1, and R_3 dependency on R_2, but not on R_1, and if we assume that R_3 is a popular rule, then processor starts to trace starting from top rule R_3 in depth-first search. After that, processor visits R_2, which R_3 depends on, and visits R_1, which R_2 depends on. As a result, all dependent rules of R_3, which are R_1 and R_2 are computed. Processor writes the BVM addresses of dependent rules to the particular subparts of RAM-1, and sets bit-0 at $Address_{511}$ in RAM-1. Assertion of bit-0 at $Address_{511}$ is a handshaking process between processor and TCAM Cache Writer Interface, which means that writing the BVM addresses of dependent rules for each popular rule is completed by processor. After this point, TCAM Cache Writer Interface reads the most frequently used BVM addresses together with their dependencies from RAM-1, and accesses the header fields of all these rules from RAM-3. Note that address bus of RAM-3 is actually BVM addresses of rules. TCAM Cache Writer Interface deals with the problem of overwriting the same rules to TCAM by using another on-chip memory named Flag-RAM. TCAM Cache Writer Interface always stores a temporary copy of written Rule IDs in Flag-RAM, and constantly checks whether it writes the current Rule to TCAM previously.

TCAM Cache Writer Interface informs other functional blocks in MM when it com-

pletes the writing process. After that, another round on the detection of popular and dependent rules can begin. Note that, when new popular rules and dependent rules are written to RAM-1 by Locality Detection and processor, respectively, TCAM Cache Writer Interface deletes all cache contents and fills it with the new rules. This operation is actually required in order to support dynamic updates on $\mathcal{R}$ in BVM. If an incremental update on TCAM is carried out instead of deleting all rules, then, new dependencies on dependency graph cannot be determined at run-time.

An example of a rule set consists of 4 rules with 4 header fields are given in Fig. 3.9. In this set, each header field is demonstrated by a single bit. If we consider the order of rule priorities as $R_0 > R_1 > R_2 > R_3$, then the generated rule dependency graph by processor will be as in Fig. 3.10. If we define R_2 as a popular rule, then the dependent rules of R_2, which are R_0 and R_1 , are also written to RAM-1 in the respective 8-words subpart. If we only store R_2 to TCAM, then an packet of header "1011" matches with R_3 in TCAM instead of R_0 which leads to a mismatch in classification.

	H1	H2	H3	H4
R0	1	0	1	1
R1	1	0	*	*
R2	1	*	1	*
R3	1	0	0	*

Figure 3.9: Example rule set consisting of 4 rules

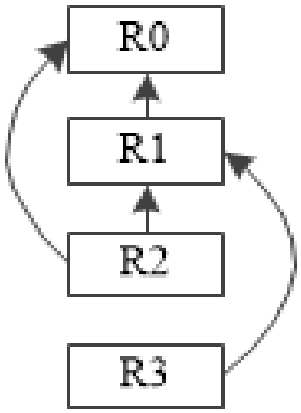

Figure 3.10: Example dependency graph for 4 Rules

Overall flow diagram between functional units in FASST MM is given in Fig. 3.11

for the example case of R_2 to be the only popular rule detected over the time window W_s for the example the rule set in Fig 3.9.

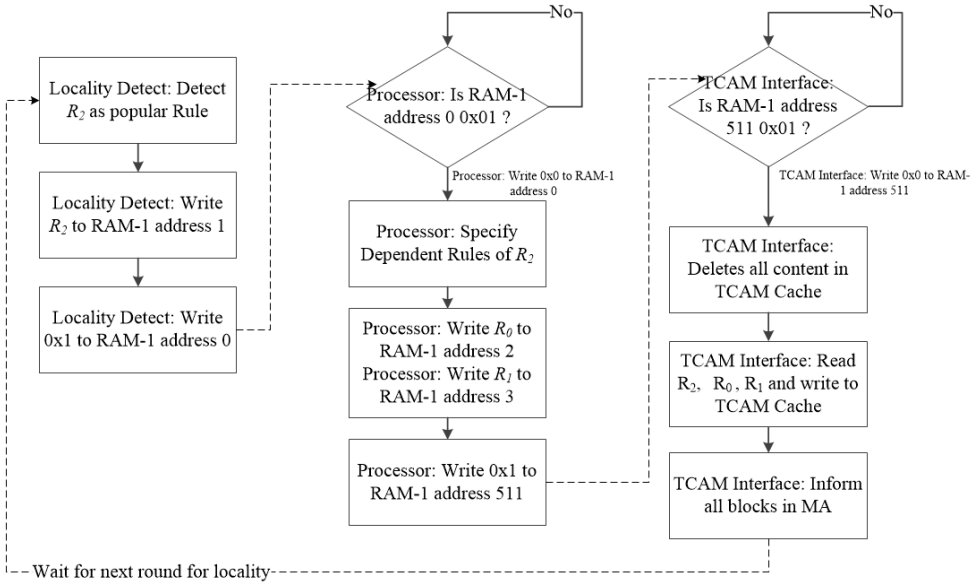

Figure 3.11: Process flow diagram in Match Monitor

3.6 Match Arbiter

FASST Match Arbiter (MA) continuously monitors TCAM and BVM results, which are Rule IDs of matched entries. Thereafter, it discards late duplicate match results of BVM if the same match is obtained from TCAM. Same match result (Rule ID) can be provided by both of these modules because of parallel processing, as explained in Section. 3.1. For example, consider a packet P matches a flow entry F_j with Rule ID j in both BVM and TCAM. The match result of Rule ID j from BVM asserts 80 clock cycles after P enters the pipeline, whereas TCAM provides the result of Rule ID j in 3 clock cycles. Therefore, MA discards this particular match of BVM, 80 clock cycles + processing delay after P enters BVM. Processing delay is the internal delay of MA.

Detailed block diagram that shows the data flow in MA is illustrated in Fig. 3.12. MA handles this discarding process by using a First-In First-out (FIFO) data structure in-

side Controller unit in MA. Each match result of TCAM fills this FIFO in Controller. Controller unit in MA ,which receives match results from BVM directly, checks whether the current matched Rule ID is already written to this FIFO by TCAM. If it is written, current Rule ID from BVM is ignored and the copy of Rule ID in FIFO is removed. The output of Controller unit in MA and the output of TCAM fills another two end FIFOs, namely FIFO-1 and FIFO-2. These FIFOs are basically used to perform fair scheduling process. Note that, FASST utilized *every* matched result of TCAM, which is Rule ID of matched rule. Therefore, output of TCAM is directly connected to FIFO-2. However, match Rule IDs of BVM should be filtered to remove late duplicates. For this purpose, filtering is performed by Controller unit, and the output of Controller, which is again Rule ID, is connected to FIFO-1. Note that, in order to avoid overflow conditions in FIFO-1 and FIFO-2, End Arbiter unit in MA is run at 400 MHz, which is twice of the running frequency of FASST.

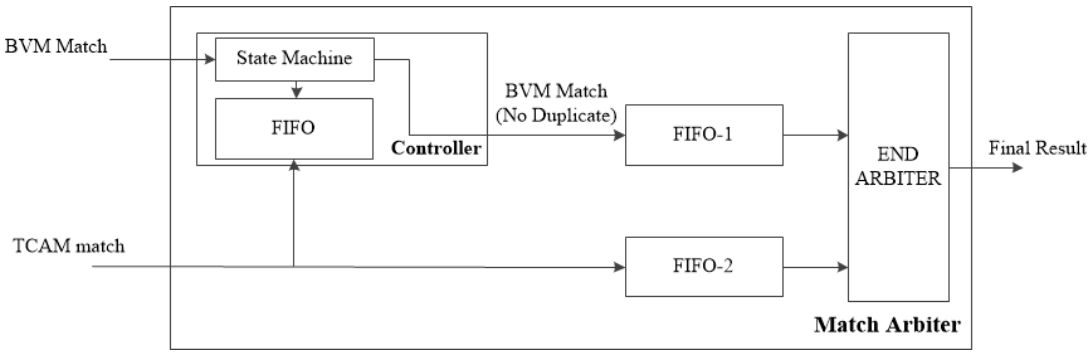

Figure 3.12: Internal block diagram of Match Arbiter Block

To this end, the packets that match to $\mathcal{F}$ are processed with t_{fast} latency while the packets that match $\mathcal{R} - \mathcal{F}$ and the packets that do not match any rule in $\mathcal{R}$ are processed with t_{slow} latency. Consequently the arrival order of packets is altered at FASST output. Distinct rule entries can have different action fields in OpenFlow protocol [2] and disordering of packets belonging to different flows is not a big problem. This is because out-of-order arrival of packets at the destination is a common situation in IP networks due to parallelism in network components or configuration [51], [52], [53]. Reordering is carried out at higher layers of network protocol stack as in Transport Layer in TCP/IP. All packets in a stream which are members of same flow require the same forwarding treatment in any SDN-enabled data plane switches [14].

As a result, packet order in same flow is a desired property.

3.7 Analysis of Packet order in FASST and Correcting the Transient Packet Order Changes

Altering the order of packets belonging to same flow is a transient operation that lasts a maximum of $t_{slow} = 80$ clock cycles inside FASST. Packet ordering is corrected automatically within this time interval using a tag field for each packet during query phase. In other words, when a packet matches a flow entry F_j with Rule ID j in TCAM, Rule ID j and a corresponding tag field are written to FIFO inside Controller unit in MA. Writing a tag field provides an additional information for the packet number within the same flow. That is to say; while Controller Unit in MA checks the FIFO content to discard the late duplicates of TCAM results, it reads both rule ID j and a tag field. As a result, it can separate the packets of same flow and make a correction in the order within a small interval. Tag field in our situation is a simple 7-bit counter value that counts up to 127 with overflow capability. FASST assigns increasing counter value to packets before sending them BVM and TCAM at the same time. Since $t_{slow} = 80$ clock cycles, 7-bit counter assures unique values within processing delay of BVM.

Table. 3.2 shows a query operation at the steady state condition of FASST. Steady state condition can be defined as a network idle condition for a limited time. For our work, when network line is idle for any 400 ns at run-time, which means no packet arrives to FASST within 400 ns for query, then BVM pipeline will be empty and steady state condition will be accomplished. For this case, packet orders of same flow are preserved. Consider that rule entries *A* and *B*, which correspond to flows of F_A and F_B, are stored in TCAM, BVM pipeline is empty at this time and the packets arriving to FASST at time [0,7] in terms of clock cycles are *A1, B1, C1, D1, A2, B2, C2*, where *(A1,A2), (B1,B2), (C1,C2)* and *D1* belongs to flows of F_A, F_B, F_C and F_D respectively. Moreover, end to end latency of BVM is 80 clock cycles, and TCAM outputs the match results in 3 clock cycles. As observed from MA output, the order of packets for *(A2,B2)* and *(C1,D1)* are different from the arriving order, which means a disorder in different flows. Packet orders of same flow are preserved. For example,

the order of *A1* and *A2* is the same for F_A.

Table 3.2: Rule query in steady state

Time (Clk)	0	1	2	3	4	5	6	7	8	.	80	81	82	83	84	85	86	87
Packets	A1	B1	C1	D1	A2	B2	C2											
BVM											A1	B1	C1	D1	A2	B2	C2	
TCAM				A1	B1			A2	B2									
MA				A1	B1			A2	B2				C1	D1			C2	

In order to demonstrate transient packet disordering of same flow, the situation in Table. 3.3 can be analyzed. This situation illustrates that at t=0, packets *A1*, *A2* and *B1* arrive to FASST for query. At t=73, writing the popular rules to TCAM, which are *A* and B in this case, is completed. This means that, after t=73, all packets are sent to BVM and TCAM concurrently. However, at t=73, packets *B1*, *A2* and *A1* are at stage 71, 72 and 73 in two dimensional pipeline, respectively. Therefore, there are another 7 clock cycles to output a match result for *A1* from BVM. Similar situation can be applied for *B1* and *A2*. Consider that, at t=73 and t=74, *A3* and *B2* arrives to FASST. As a result, before BVM, TCAM outputs the Rule IDs of *A3* and *B2* which belong to the same flow with *A1*, *A2* and *B1*. As a result, packet orders of same flows at the outputs are different from the order at the beginning .

Table 3.3: Packet orders of same flow

Time (Clk)	0	1	2	.	73	74	75	76	77	78	79	80	81	82	.	153	154
Packets	A1	A2	B1		A3	B2											
BVM												A1	A2	B1		A3	B2
TCAM								A3	B2								
MA								A3	B2			A1	A2	B1			

Packet disordering of same flow lasts only 5 clock cycles for example given in Table. 3.3. Consider that, after t=73, all packets are in the format A_i and B_i from t=73 to

t=t_x, such as *A3*, *B2*, *A4*, *B3*, *A5*, *B4*, ..., *An*, *Bn*. Such a characteristics on network traffic causes the worst case transient time in FASST. Hence, at time t=80, Controller in MA reads *A1* from BVM and checks FIFO. At this time, FIFO is field with *[A3, B2, A4, B3]* where *A3* is the first data. Controller reads the first data, which is *A3*, and observes that tag field '3' is incompatible with *A1*. At the same time, TCAM provides match result for *A5*. Hence, at t=80, there are two match results for F_A, which are *A1* and *A5*. Therefore, using a two times faster End Arbiter Module, at t=80, both *A1* and *A5* are provided from hardware as match results. Same procedure can be applied to *A2* and *B4* at t=81, and *B1* and *A6* at t=82. Hence, the order of same flow is preserved after t=82. The illustration of this sequence is given in Table. 3.4. Note that, after t=153, match results from BVM for the packets for F_A and F_B are discarded, because these match results are already written to FIFO by TCAM.

Table 3.4: Transient operation for rule disordering in same flow

Time (Clk)	0	1	2	.	73	74	75	76	77	78	79	80	81	82	83	84	85
Packets	A1	A2	B1		A3	B2	A4	B3	A5	B4	A6	B5	A7				
BVM												A1	A2	B1			
TCAM								A3	B2	A4	B3	A5	B4	A6	B5	A7	B6
MA								A3	B2	A4	B3	A1 A5	A2 B4	B1 A6	B5	A7	B6
Time (Clk)	153	154	155	156	157	158											
Packets	B41	A43	B42	A44													
BVM	A3	B2	A4	B3	A5	B4											
TCAM	B40	A42	B41	A43	B42	A44											
MA	B40	A42	B41	A43	B42	A44											

Worst case transient time for correction of the packet ordering in same flow occurs when there are packets at the first stage of BVM pipeline at the same time of the completion of writing rules to TCAM Cache. Moreover, the packets in this first stage should match rule entries that are also stored in cache. For this case, transient time

lasts 80 clock cycles (400 ns) in worst time. After 400 ns, packets in same flow will output from FASST in correct order. However, 400 ns is a very small interval compared to real world network characteristics. According to the prior study [20], 80% of the flow inter-arrival times are between 4 ms and 40 ms in university data centers. Moreover, across these data centers, 80% of these flows are smaller than 10 KB in size. This means that for a 10 GbE, most of the flows in data centers lasts only about 1000 ns, and there are big intervals of 4 ms - 40 ms among these flows. As a result, the probability of observing a flow on network line is 1/4000, which is roughly 0.025% in best case if we take inter arrival times of flows as 4 ms. Therefore, most of the time BVM pipeline will be in steady state, and even though FASST corrects the order of packets in same flow in 400 ns in worst case, this correction process will not be required in most of the time.

CHAPTER 4

FPGA IMPLEMENTATION OF FASST HARDWARE ARCHITECTURE

The proposed hardware architecture FASST is designed and implemented on a state-of-the-art FPGA in order to provide a low latency, high throughput and scalable SDN Flow Table. FASST supports a strictly positive edge triggered synchronous design and portable HDL source code.

Bit Vector algorithms are currently used in FPGAs by exploiting massive parallelism [8], [43], [15]. Due to concurrent processing in FPGAs, the partial bit vector results on local functional units are bit-wise ANDed at a single clock cycle. However, based on SDN Flow Table implementations, as the rule set size increases, the clock rate deteriorates significantly due to the limited on-chip resources. Therefore, two-dimensional pipeline architecture together with bit vector approach has been proposed [8]. In this architecture, functional blocks process in a pipelined fashion such that computational operations are divided into multiple clock cycles, which provides a scalable solution in terms of clock rate in FPGAs.

In FASST FPGA implementation, bit vector algorithm is used in both BVM and TCAM design. All functional elements in BVM are connected in a pipelined fashion, including priority encoders due to high number of stored rules $|\mathcal{R}|$ which is 512. On the other hand, TCAM design is completely run with parallel processing due to less number of stored rules $|\mathcal{F}|$, which is 32 in our case. Moreover, both BVM and TCAM are implemented as a modular architecture that enables the design to support different number of rule sets, or different number of header fields for a variety of configurations. In other words, simply changing the number of *Rule BVMs*, *Stride*

BVM or *Stride TCAM* functional units provides lookup on different number of rule sets including $|\mathcal{R}|$ and $|\mathcal{F}|$.

FASST FPGA implementation presents a hardware-based approach to rule caching problem considering dependencies, compared to software-related approaches in earlier work [9]. All operations including generation of rule dependency graph, searching the graph with depth-first-search (DFS) and sorting rule priorities are completely carried out on a single hardware. Match Monitor (MM) in FASST can be reconfigured to support different types of dependency graphs in terms of rule depths or rule size.

All on-chip memory units such as RAM-1 in FASST are implemented using dual port or single port on-chip memory blocks of FPGA. These are FPGA specific embedded memory units called *M20K* and can be configurable for different *depth x width* selections. In BVM and TCAM, *Stride BVM* and *Stride TCAM* units have access to a series of these memory blocks for lookup process. As a result, different from recent work [8], our proposed architecture utilizes only embedded memory blocks instead of logic gates for the data-associative elements of bit vector approach. At normal conditions, maximum clock rates of embedded memory blocks is much more higher than logic gates inside FPGA, which enables us to reach high throughput values while maintaining classification. However, *depth x width* selection of these on-chip memory resources is crucial in order to minimize the resource usage on hardware and satisfy the timing requirements with respect to clock rate.

The design of overall architecture FASST is implemented on FPGA using VHDL hardware design language except for the processor unit in MM, which is run at a relatively lower clock frequency with respect to hardware blocks. Since latency is not a performance requirement while generating and searching dependency graph, FPGA-based soft processor called NIOS II is implemented inside FASST MM processor unit. This soft processor delivers an ideal embedded solution in terms of flexibility, high performance and low cost. NIOS II processor in our architecture communicates with the necessary blocks described in Section. 3.5 at real time.

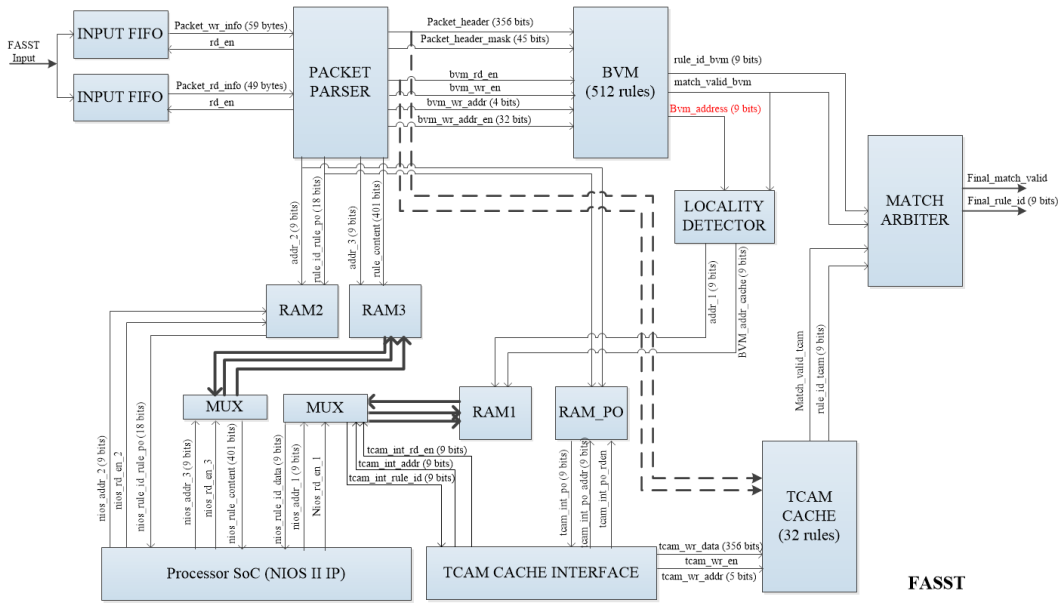

Figure 4.1: General block diagram of FASST FPGA implementation

4.1 General View of FPGA Implementation

The general view of FASST FPGA implementation can be seen in Fig. 4.1. Based on the description in Chapter 3, there is one BVM unit which supports look up for 512 rules with $t_{slow} = 80$ clock cycles latency, and there is one TCAM unit which supports look up for 32 rules with $t_{fast} = 3$ clock cycles latency. Locality Detector, TCAM Cache Interface, NIOS II as processor unit, RAM-1, RAM-2, RAM-3 and RAM-PO are the blocks of Match Monitor (MM) in FASST. RAM-PO is a on-memory that is used to store <Rule Priority> information of the rule entries in BVM.

Input to FASST is a specific data format that supports OpenFlow v1.1.0 rule entry content [2]. For testing purposes, FASST communicates with external world through RS-232 serial interface to get the input. An external serial interface controller writes the input data to two FIFOs named Input FIFO. Firstly, flow table entries, $\mathcal{R}$, are written to Upper Input FIFO, which means rule insertion phase. Afterwards, Lower Input FIFO is filled with network characteristic information through serial interface, which is used for query phase. Network characteristics in Lower Input FIFO is basically a data format that includes <Packet Header><Packet Count>. In other words, Lower Input FIFO gives the information of packet arrival process. Upper Input FIFO

stores the relevant content for 512 flow entries to be written to BVM at configuration, which includes <BVM Address><Rule ID><Rule Priority><List of Header fields to match><List of Mask bits of each field>. Lower Input FIFO stores a network scenario information which includes the similar content as in Upper Input FIFO. However, in lower Input FIFO there is no information for Rule ID, Rule Priority, BVM address, or mask bits, which are unnecessary data for query phase. However, there is an additional packet number information in this lower Input FIFO to indicate the number of packets that is requested to be classified in a time window. In summary, while upper FIFO gives total rule set content, $\mathcal{R}$, in SDN Flow table, lower Input FIFO stores the characteristic of network traffic (packet information P) for query.

Firstly, Packet Parser (PP) reads the upper Input FIFO to insert to rule entries to BVM. This is called as Rule Insertion Phase. PP writes the rules to BVM using header bits, mask bits and other control and selection bits such as *bvm_wr_en*, *bvm_wr_addr(3:0)* and *bvm_wr_addr_en(31:0)*. At the same time, the corresponding $< RuleID_i >< RulePriority_m >$ pairs are written to on-chip embedded memory RAM-2 by PP. Note that these pairs are not ordered. Later, processor in MM sorts these pairs in order to generate dependency graph. Moreover, the copy of the inserted rules are also stored in RAM-3 in rule insertion phase such that processor reads them while generating rule dependency graph. At rule insertion phase, PP has no effect on TCAM. Rule insertion to TCAM is carried out by TCAM Cache Interface dynamically depending on the locality information on network traffic. After rule insertion phase is completed, PP latches the network characteristic information from lower Input FIFO and sends series of packets P to BVM and TCAM for query phase. Hence, *bwm_rd_en* and *packet_header(355:0)* signals are also routed to TCAM in addition to BVM.

At rule insertion phase, FASST BVM receives the content of header list of rule entries from PP via a memory-mapped interface and stores its internal on-chip memories with rule header contents. These on-chip memories inside BVM are actually data-associative memories rather than address mapped memories as described in Section. 2.4.1. A total of 512 rule entries with 15-tuple headers with 356 bits are stored in BVM. During query phase, BVM receives the series of packet headers and outputs the match results as Rule ID named *rule_id_bvm (8:0)*. The start of query phase in BVM is signaled with *bwm_rd_en* from PP.

Locality Detect communicates with BVM and has access to RAM-1 during run-time. It continuously tracks the BVM addresses of matched rules and checks whether the number of matches for a particular BVM address exceeds the threshold level Thr. In case of a locality detection, the associated BVM addresses of rules are written to RAM-1 through *addr_1(8:0)* and *BVM_addr_cache (8:0)*.

Processor unit is implemented as a SoC in Match Monitor (MM) in FASST. It has direct access to RAM-1, RAM-2 and RAM-3. However, as seen in Fig. 4.1, one-side of memory mapped interface of RAM-1 and RAM-3 are shared between Processor and TCAM Cache Interface. This sharing is performed in order to reduce the memory utilization of on-chip resource in FPGA. Processor has no functionality during rule insertion phase. After rule insertion is completed by PP, it automatically constructs the dependency graph for the current rule set $\mathcal{R}$ in BVM. In query phase, processor reads RAM-1 to receive popular BVM addresses written by Locality Detector. Then, it computes the dependent rules of popular rules and rewrites the BVM addresses of dependent rules to RAM-1. At the same time, processor writes the associated <Rule Priority> of all popular and dependent rules to RAM-PO such that TCAM Cache Interface can read and send them to TCAM. The connection of processor to RAM-PO is not shown in Fig. 4.1. Furthermore, since FASST is a purely hardware architecture and tested on run-time, power consumption by FPGA core is constantly read by processor unit communication blocks and the values are reported to host PC via serial interface.

TCAM Cache Interface has access to RAM-PO, RAM-1 and RAM-3. The connection to RAM-3 is not shown in Fig. 4.1 due to complex wiring on diagram. TCAM Cache Interface only performs in query phase, similar to processor and Locality Detector units. If a temporal locality is observed on network traffic, this interface reads the BVM addresses from RAM-1, the Rule Priorities from RAM-PO and rule contents from RAM-3. After that, it writes the frequently used rules to TCAM. Rule priorities are also needed while inserting rules to TCAM due to implementation of *POEnc-TCAM*.

Two on-chip RAM blocks that are not illustrated in Fig. 4.1 are *Chain_RAM* and *Flag_RAM* embedded memories. *Flag_RAM* is only accessed by TCAM Cache In-

terface to ensure that each rule is written to TCAM only once. In other words, TCAM Cache Interface stores a log data while maintaining a write operation to TCAM. *Chain_RAM*, on the other hand, is the on-chip memory block that is used to store the number of dependent rules for each popular rule inside RAM-1. *Chain_RAM* is written by NIOS II and read by TCAM Cache Interface. When a rule insertion to TCAM occurs, TCAM Cache Interface needs this information in order not to exceed the size of TCAM in FASST.

TCAM is implemented as a parallel bit vector approach in FPGA similar to BVM design. However, instead of pipelined architecture, all header strides in a packet P are classified concurrently within a single clock cycle. Rule insertion to TCAM is conducted only by TCAM Cache Interface.

The output of both BVM and TCAM Cache is sent to Match Arbiter (MA) through *match_valid_bvm*, *rule_id_bvm(8:0)*, *match_valid_tcam* and *rule_id_tcam (8:0)*. The final result from Match Arbiter unit is signaled with match valid and Rule ID outputs.

In FPGA, overall architecture of FASST is run at a frequency of 200 MHz except for NIOS II processor. Due to the limitations caused by execution of instructions-per-cycle parameters in NIOS II, it is run at a frequency of 100 MHZ. However, this relatively lower frequency has no effect on run-time throughput and latency, because both BVM and TCAM Cache classifies the incoming packets at 200 MHz.

4.2 Input Packet Format for Rule Insertion and Rule Query Phases

In FASST, PP identifies only a specific packet format in order to start rule insertion and rule query. Packet format for a single rule entry at rule insertion phase is given in Table. 4.1. There is a total of 59 bytes to define a single rule entry in BVM. As a result, in order to store 512 rules, a total of 59 x 512 = 30208 bytes should be written to upper Input FIFO. Since the number of bits in headers are mostly dividable by 8, data width for Upper Input FIFO is 8-bits (1 byte). For the fields whose bit widths are not multiple of 8, remaining bits are ignored in rule insertion phase. For example, <Rule ID> and <Rule Priority> for 512 rules are defined with 9 bits, which are *Rule_ID(8:0)* and *Priority (8:0)*, in the range *(0, 511)*. Hence, least significant

bits (LSB) for <Rule ID> and <Rule Priority> are stored as single bytes in Byte 1 and Byte 3, respectively. On the other hand, the most significant bits (MSB), which are *Rule_ID(8)* and *Priority(8)*, are stored in Byte 2 and Byte 4 using only a single bit at the lowest bit location. Hence, remaining bits of Byte 2 and Byte 4 are ignored during rule insertion. Same situation is valid for *Vlan_Id (3:0)*, *Vlan_Po (2:0)*, *Mpls_Label (3:0)*, *Mpls_Po (2:0)*, *Tos (5:0)*, *Mask (4:0)* and *Write_Address(8)* fields.

Table 4.1: Packet format per flow entry for rule insertion phase

Byte 1	Rule_ID(7:0)	**Byte 2**	Rule_ID(8)	**Byte 3**	Priority(7:0)
Byte 4	Priority(8)	**Byte 5**	Ingress(31:24)	**Byte 6**	Ingress(23:16)
Byte 7	Ingress(15:8)	**Byte 8**	Ingress(7:0)	**Byte 9**	Metadata(63:56)
Byte 10	Metadata(55:48)	**Byte 11**	Metadata(47:40)	**Byte 12**	Metadata(39:32)
Byte 13	Metadata(31:24)	**Byte 14**	Metadata(23:16)	**Byte 15**	Metadata(15:8)
Byte 16	Metadata(7:0)	**Byte 17**	Src_MAC(47:40)	**Byte 18**	Src_MAC(39:32)
Byte 19	Src_MAC(31:24)	**Byte 20**	Src_MAC(23:16)	**Byte 21**	Src_MAC(15:8)
Byte 22	Src_MAC(7:0)	**Byte 23**	Dst_MAC(47:40)	**Byte 24**	Dst_MAC(39:32)
Byte 25	Dst_MAC(31:24)	**Byte 26**	Dst_MAC(23:16)	**Byte 27**	Dst_MAC(15:8)
Byte 28	Dst_MAC(7:0)	**Byte 29**	Eth_Type(15:8)	**Byte 30**	Eth_Type(7:0)
Byte 31	Vlan_Id(11:4)	**Byte 32**	Vlan_Id(3:0)	**Byte 33**	Vlan_Po(2:0)
Byte 34	Mpls_Label(19:12)	**Byte 35**	Mpls_Label(11:4)	**Byte 36**	Mpls_Label(3:0)
Byte 37	Mpls_Po(2:0)	**Byte 38**	Src_IP(31:24)	**Byte 39**	Src_IP(23:16)
Byte 40	Src_IP(15:8)	**Byte 41**	Src_IP(7:0)	**Byte 42**	Dst_IP(31:24)
Byte 43	Dst_IP(23:16)	**Byte 44**	Dst_IP(15:8)	**Byte 45**	Dst_IP(7:0)
Byte 46	Protocol(7:0)	**Byte 47**	Tos(7:0)	**Byte 48**	Src_Port(15:8)
Byte 49	Src_Port(7:0)	**Byte 50**	Dst_Port(15:8)	**Byte 51**	Dst_Port(7:0)
Byte 52	Mask(12:5)	**Byte 53**	Mask(4:0)	**Byte 54**	Src_IP_Mask_1(7:0)
Byte 55	Src_IP_Mask_2(7:0)	**Byte 56**	Dst_IP_Mask_1(7:0)	**Byte 57**	Dst_IP_Mask_2(7:0)
Byte 58	Write_Address(7:0)	**Byte 59**	Write_Address(8)		

In query phase, packet format for a single packet is given in Table. 4.2. This format is stored in Lower Input FIFO and very similar to the content in Upper Input FIFO. The difference is that BVM and TCAM need only <List of header bits> for look up

process in query phase. Therefore, <Rule ID>, <Rule Priority>, mask bits, prefix lengths, or address bits are removed from this format. Moreover, due to exploiting temporal locality in FASST, an additional information which gives the number packets to be queried successively is added to the format. The size of Lower Input FIFO depends on the network scenario determined by an external controller via serial interface. For example, consider that, there is a network characteristics consisting of two input packets P_1, and P_2. If *Packet_Count(15:0)* is set to *0x000A* for P_1 and *Packet_Count(15:0)* is set to *0x0032* for P_2, and if P_1 information is written to Lower Input FIFO first, then the order in query phase will be as in Fig. 4.2. For this scenario, the size of Lower Input FIFO is 49 bytes x 2 = 98 bytes. As network scenario becomes more complex, the size of Lower Input FIFO increases.

Table 4.2: Packet format per packet header for rule query phase

Byte 1	Packet_Count(15:8)	**Byte 2**	Packet_Count(7:0)	**Byte 3**	Ingress(31:24)
Byte 4	Ingress(23:16)	**Byte 5**	Ingress(15:8)	**Byte 6**	Ingress(7:0)
Byte 7	Metadata(63:56)	**Byte 8**	Metadata(55:48)	**Byte 9**	Metadata(47:40)
Byte 10	Metadata(39:32)	**Byte 11**	Metadata(31:24)	**Byte 12**	Metadata(23:16)
Byte 13	Metadata(15:8)	**Byte 14**	Metadata(7:0)	**Byte 15**	Src_MAC(47:40)
Byte 16	Src_MAC(39:32)	**Byte 17**	Src_MAC(31:24)	**Byte 18**	Src_MAC(23:16)
Byte 19	Src_MAC(15:8)	**Byte 20**	Src_MAC(7:0)	**Byte 21**	Dst_MAC(47:40)
Byte 22	Dst_MAC(39:32)	**Byte 23**	Dst_MAC(31:24)	**Byte 24**	Dst_MAC(23:16)
Byte 25	Dst_MAC(15:8)	**Byte 26**	Dst_MAC(7:0)	**Byte 27**	Eth_Type(15:8)
Byte 28	Eth_Type(7:0)	**Byte 29**	Vlan_Id(11:4)	**Byte 30**	Vlan_Id(3:0)
Byte 31	Vlan_Po(2:0)	**Byte 32**	Mpls_Label(19:12)	**Byte 33**	Mpls_Label(11:4)
Byte 34	Mpls_Label(3:0)	**Byte 35**	Mpls_Po(2:0)	**Byte 36**	Src_IP(31:24)
Byte 37	Src_IP(23:16)	**Byte 38**	Src_IP(15:8)	**Byte 39**	Src_IP(7:0)
Byte 40	Dst_IP(31:24)	**Byte 41**	Dst_IP(23:16)	**Byte 42**	Dst_IP(15:8)
Byte 43	Dst_IP(7:0)	**Byte 44**	Protocol(7:0)	**Byte 45**	Tos(7:0)
Byte 46	Src_Port(15:8)	**Byte 47**	Src_Port(7:0)	**Byte 48**	Dst_Port(15:8)
Byte 49	Dst_Port(7:0)				

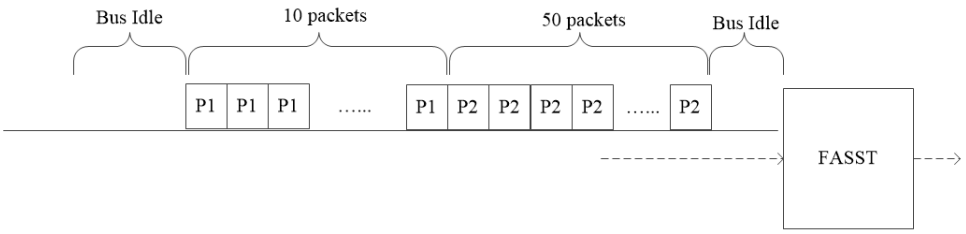

Figure 4.2: Order of incoming packets to FASST for rule query phase

4.3 Packet Parser

In FPGA implementation of FASST, Packet Parser (PP) has two functions: Reading the Upper Input FIFO to insert 512 rules to BVM over a memory mapped structure, and reading the Lower Input FIFO to send a series of packets to both BVM and TCAM for classification. Therefore, PP in Fig. 4.3 has a FIFO interface to read the contents of these FIFOs.

PP is implemented as a state machine design similar to all functional units. This state machine basically two phases: rule insertion phase and rule query phase. In rule insertion phase, PP waits for 59 bytes to be written to Upper Input FIFO by polling *fifo_wrused(14:0)* input port. This port constantly gives the number of stored data in FIFO in FPGA. As soon as it detects 59 bytes, it reads the data from Upper Input FIFO via *fifo_data_in(7:0)* in the received order as described in section. 4.2. Thereafter, PP asserts enable and address signals such as *bvm_wr_en*, *bvm_wr_addr(3..0)* and *bvm_wr_addr_en(31:0)*. Moreover, *packet_header(355:0)*, *mask(12:0)* are asserted by PP at this time. After a single clock cycle, it deasserts these signals to their default value, because BVM has the capability of receiving the rule contents in a single clock cycle due to its internal registers. At this point, *bvm_wr_addr_en(31:0)* means the selection of the corresponding *Rule BVM* block as explained in Section. 3.3. For example, if *Write_Address(8:0)* in Table. 4.1 is equal to *000010001*, PP sets *bvm_wr_addr_en(31:0)* to *0x00000002* to select second horizontal pipeline of $RuleBVM_0$, which includes $RuleBVM_0(1,0)$, ..., $RuleBVM_0(1,46)$. This is because address of *000010001* means address 17 and rules from address 16 to address 31

are stored in second horizontal pipeline of $RuleBVM_0$. Hence, the selection bits for each horizontal pipeline of *Rule BVM* are one-hot encoded in *bvm_wr_addr_en(31:0)*. The address bit of *bvm_wr_addr(3:0)* are actually the subaddress in the corresponding *Rule BVM*, which is *Write_Address(3:0)* in Table. 4.1. Before inserting a new flow entry, PP waits for *busy_bvm* signal to check whether BVM is ready to store next rule entry. Therefore, a handshaking process is performed during rule insertion phase. This cycle lasts until all 512 rules are received from Upper Input FIFO and written to BVM.

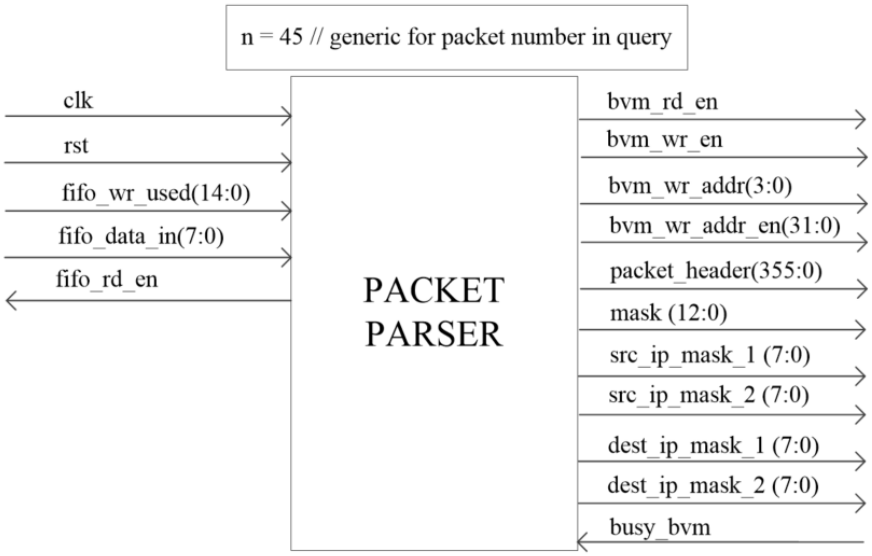

Figure 4.3: Packet Parser Block

When rule insertion phase is completed, PP waits for n x 49 bytes to be written to Lower Input FIFO to start the query phase by polling *fifo_wrused(14:0)* again, where n denotes for the number of packets with different headers. This n value is a generic constant for PP, and it can be changed at synthesis level. After all network characteristics are written to Lower Input FIFO, PP reads all contents first, fills its internal registers and asserts read enable signal *bvm_rd_en* and header data signal *packet_header(355:0)*. The purpose of storing all PP internal registers with network characteristics from Lower Input FIFO is to achieve the desired line rate while maintaining classification.

4.4 Implementation Bit Vector Module (BVM)

Bit Vector Module (BVM) is one of the two hardware based classification engines implemented in FASST. BVM is implemented as a two dimensional pipelined architecture to achieve high clock rates while maintaining classification. The main difference of 2D pipelined architecture of BVM from other packet classification techniques is that pipeline architecture in BVM enables the lookup module to receive incoming packets in every clock cycles. In other words, incoming packets for query are not waited. After pipeline is full, BVM outputs the match results at every clock cycle if line utilization is 100%. Therefore, throughput is achieved at maximum value.

The implementation details of BVM in FPGA are exactly compatible with the conceptual design described in section. 3.3. Firstly, BVM is designed and implemented as a modular architecture. In other words, in order to store and query 512 rules, BVM architecture is divided to 16 exact functional elements called *Rule BVM*, where each *Rule BVM* has the capability of storing 32 rules. Moreover, each *Rule BVM* has a connection to an another *Rule BVM* and two *POEnc* units. The connection of using two encoders for one *Rule BVM* is due to the fact that *POEnc* units implemented in FASST are designed and configured to find highest priority match among 16-rules to minimize logic gate delays.

For a clear understanding, connection diagram and interface details of two *Rule BVM* and four *POEnc* units are given in Fig. 4.4. The structure given in Fig. 4.4 stores a total of 64 rules, where $RuleBVM_0$ stores the first 32 rules (0 to 31) and $RuleBVM_1$ stores the next 32 rules (32 to 64). Note that rule numbers here are defined as BVM address. For example, rule 5 means the rule at BVM address 5. Dashed lines represent the interface signals which are only used in rule insertion phase by PP. Other signals such as *packet_header(355:0)* and *bvm_rd_en* are used in query phase. *bvm_rd_en* denotes the read enable active signal in query phase. Note that, *packet_header(355:0)* signal is utilized in both rule insertion and rule query phase. Moreover, 32-bit register signal, which represents the group information among 512 rules to find out Rule IDs as defined in Section. 3.3, is not shown in Fig. 4.4. This register is a part of *POEnc* units. *Rule BVM* units have nothing to do with this register signal.

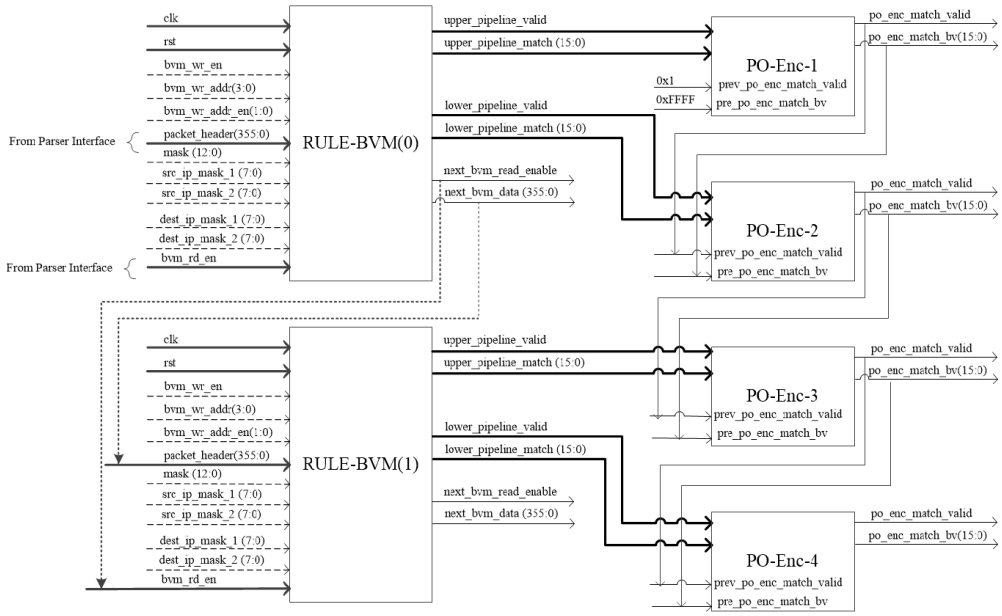

Figure 4.4: Connection diagram of two Rule-BVMs and four Priority Encoders

Moreover, since *Rule BVM* units have vertical pipeline connection between each other, read enable and output data ports of *Rule BVM*, which are *next_bvm_read_enable* and *next_bvm_data(355:0)*, are directly routed to next *Rule BVM* for pipeline processing. Hence, in query phase, the header bits of packets are only sent to $RuleBVM_0$ through *packet_header(355:0)*. After this time, all read enable and data bits are delayed by *Rule BVM* by insertion Flip-Flops (FF) in design and sent to other *Rule BVM* units to continue classification. Passing of read enable and data bits are for next *Rule BVM* in horizontal and next *Rule BVM* in vertical direction. Hence, since the routing delays on header bits, which are *packet_header(355:0)*, are significantly decreased due to insertion of FFs, high clock rates are achieved in this design.

Since FASST BVM is implemented as a two-level hierarchy, each *Rule BVM* has also internal 2D pipelined structure as described in Section. 3.3. This internal architecture of a *Rule BVM* consists of two horizontal pipelines, where each pipeline contains a total of 47 pipeline stages. Pipeline stages at the same horizontal level are responsible for the same set of 16 rules among 512 rules. Hence, each *Rule BVM* has 2-bit enable signal, *bvm_wr_addr_en(1:0)*, and this signal is used to select one of the horizontal pipelines to store rules in rule insertion phase. For example, if write address enable signal, *bvm_wr_addr_en(1:0)*, is equal to 0, upper horizontal pipeline is selected to

store the first 16 rules. In query phase, there is no functionality of *bvm_wr_addr_en (1:0)*. Upper and lower horizontal pipelines are connected to each other vertically to construct 2D pipeline architecture.

Signals of *upper_pipeline_valid* and *upper_pipeline_match(15:0)* represent the query result for the upper horizontal pipeline. The same situation is valid for lower horizontal pipeline signals, which are *lower_pipeline_valid* and *lower_pipeline_match(15:0)*. 16-bit BV (match results) has a meaning of one-hot encoding. For $RuleBVM_0$, if *lower_pipeline_valid* is '1' and *lower_pipeline_match(15:0)* is equal to *0x0004*, then there is a match for the rule at BVM address 18. Inside a ***Rule BVM***, upper and lower horizontal pipelines are connected to each other as described in section. 3.3 to decrease routing delays on high number of header bits by dividing them into strides of *s* bits. Therefore, the output *lower_pipeline_valid* asserts one clock cycle later than *upper_pipeline_valid* due to this internal connection.

The simulation results of $RuleBVM_0$ and $RuleBVM_1$ are illustrated in Fig. 4.5. The read enable signal from top level, *bvm_rd_en* is only routed to $RuleBVM_0$. After 2 clock cycles, $RuleBVM_0$ asserts *next_bvm_read_enable* to drive the read enable signal of $RuleBVM_1$.

Moreover, as observed in Fig. 4.5, all match valid signals assert to HIGH with intervals of single clock cycle. In other words, *lower_pipeline_valid* of $RuleBVM_0$ is HIGH one clock cycle after *upper_pipeline_valid* of $RuleBVM_0$ is HIGH. This demonstrates the internal two dimensional pipeline architecture of $RuleBVM_0$. Similarly, *upper_pipeline_valid* of $RuleBVM_1$ asserts HIGH one clock cycle after valid signal *lower_pipeline_valid* of $RuleBVM_0$ asserts HIGH. This timing verifies the top level vertical pipeline connection of ***Rule BVM*** units. The interval between the the first assertion of *bvm_rd_en* and *upper_pipeline_valid* of $RuleBVM_0$ is 48 clock cycles due to 47 horizontal stages inside Rule-BVMs. Additional clock cycle comes from the read latencies of data-associative memories inside Rule-BVMs. Since there are 32 *POEnc* units in overall design, 48+32 gives $t_{slow} = 80$ cycles, which is the latency of FASST BVM.

As seen in Fig. 4.4, the match results of upper and lower horizontal pipelines are

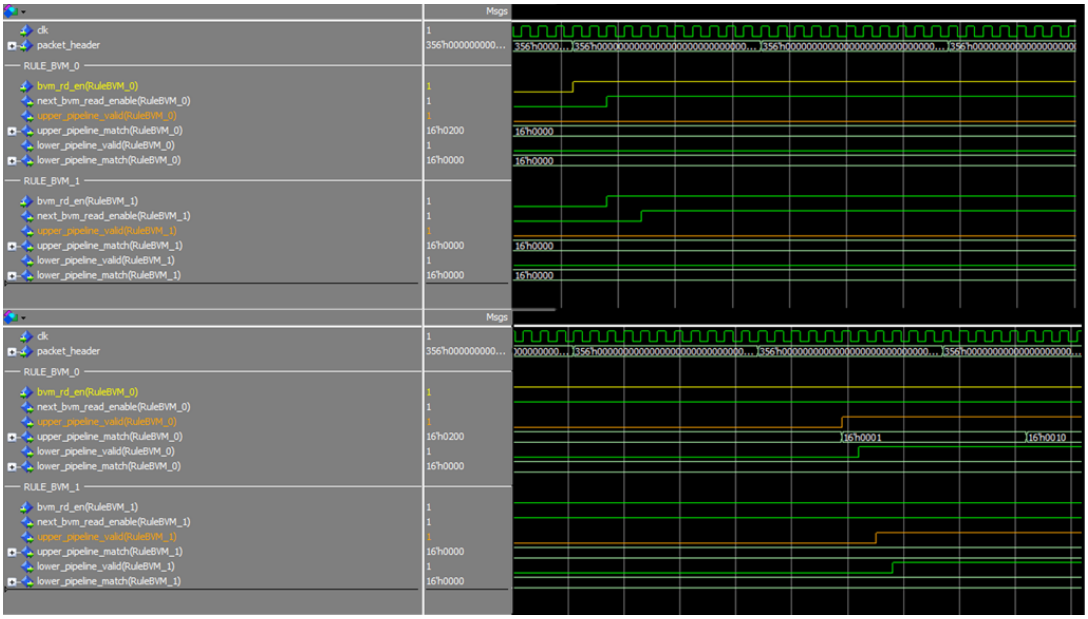

Figure 4.5: Functional simulation of Rule-BVMs for pipeline connection

sent to two *POEnc* units. Each *POEnc* reports the highest priority match between 16 rules and sends the local result to another *POEnc*. Therefore *POEnc* receives two bit vectors (match results): bit vector from previous *POEnc*, and bit vector from a horizontal pipeline of a *Rule BVM*. Since first *POEnc*, called *POEnc-1* in Fig. 4.4, has no interface connection to a previous *POEnc*, *pre_po_enc_match_bv(15:0)* port is set to '1' in order to indicate *All Match* condition. Moreover, *POEnc* units need <Rule Priority> information. These priorities are written to *POEnc* by PP over *rule_po(8:0)* port in rule insertion phase.

Simulation results of *POEnc* blocks are depicted in Fig. 4.6. Each *Rule BVM* is connected to two *POEnc* blocks through the match valid signals of two horizontal pipelines. These are *upper_pipeline_valid* for upper pipeline and *lower_pipeline_valid* for lower pipeline from $RuleBVM_0$ for *POEnc-1* and *POEnc-2*, respectively. The arrival interval between these valid signals is also single clock cycle due to internal pipeline connection of *Rule BVM*. Moreover, the latency of *POEnc* is single clock cycle. In other words, bit vector output port *po_enc_match_bv (15:0)* provides new bit vector results only 1 clock cycle after previous bit vector, *pre_po_enc_match_bv (15:0)*, input arrives. As a result, using 4 *POEnc* blocks, 64 rules can be priority encoded in a pipelined manner. For this example, encoding process lasts only 4 clock cycles, from the first match bit vector, *upper_pipeline_valid* of $RuleBVM_0$, to the

first encoded bit vector provided by *POEnc-4*, *po_enc_match_bv(15:0)*. Note that, Rule ID mapping of these bit vectors are carried out after final match result from the last *POEnc* is obtained. Using final BV and a 32-bit group information register, BVM address of the matched rule can be found. Using a <Address><Rule ID> embedded memory, FASST provides Rule ID of the matched packet in BVM as defined in Section. 3.3.

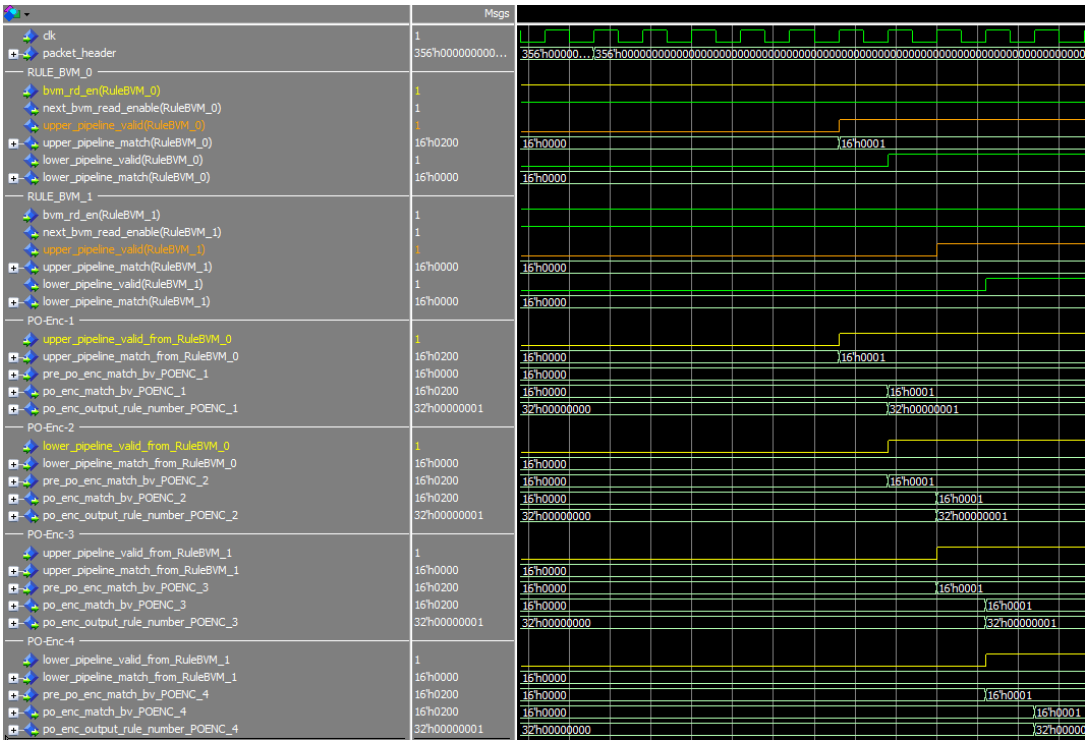

Figure 4.6: Functional simulation of 4 Priority Encoders with vertical pipeline connection

4.4.1 Details of Stride-BVM Blocks

Implementation of internal architecture of $RuleBVM_0$ is given in Fig. 4.7. $RuleBVM_0$ consists of 47 *Stride BVM* units in upper horizontal pipeline, and 47 *Stride BVM* units in lower horizontal pipeline. Note that only first two *Stride BVM* units in upper and lower horizontal pipelines are illustrated in Fig. 4.7. Same connection flow can be applied to all units in upper and lower pipelines. Each *Stride BVM* includes a *Stride BVM Controller* and *Stride BVM RAM*. The width of *Stride BVM RAM* is 16 bits because each *Stride BVM RAM* stores 16 flow rules. Moreover, the depth of *Stride*

BVM RAM is 256 because FASST uses 3-bit, 4-bit, 6-bit and 8-bit strides (s) according to type of header field, and it takes the maximum bits in these strides, which is $2^8 = 256$. In other words, FASST implements a fixed depth of *Stride BVM RAM* for a consistency in FPGA implementation. For example, 6-bit strides are used with *Tos(5:0)* header field, however, the depth of *Stride BVM RAM* is still 256. The only difference is that addresses from $2^6 = 64$ to 256 are not used for *Tos(5:0)* field.

Furthermore, as seen in Fig. 4.7, *Stride BVM RAM* is implemented as a dual-port memory in FPGA, which is shown as *PORT_A* and *PORT_B*. There are two possible applications of using an on-chip dual-port memory in BVM: Doubling the throughput by reading the query data from two ports at the same time, and supporting simultaneous rule insertion/rule query operation. Different from [8], the support for simultaneous rule insertion/rule query operation is chosen in FASST by considering future work. For this purpose, *PORT_A* is used for rule insertion and *PORT_B* is used for rule query in FASST.

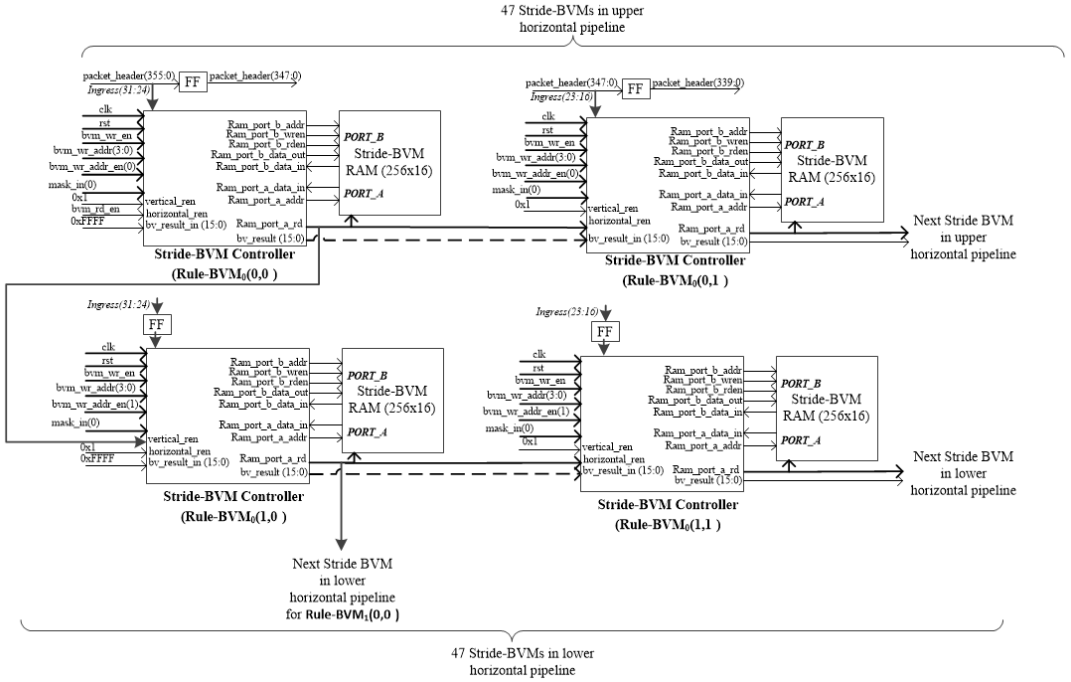

Figure 4.7: Internal architecture of a Rule-BVM for FPGA implementation

In rule insertion phase, the function of *Stride BVM Controller* is to write rule entries to *Stride BVM RAM* through ram memory mapped interface, which consists of the signals *Ram_port_b_addr*, *Ram_port_b_wren*, *Ram_port_b_rden*, 16-bit data bus

Ram_port_b_data_out and *Ram_port_b_data_in*. The utilization purpose of ram read enable signal, *Ram_port_b_rden*, and 16-bit ram data input port, *Ram_port_b_data_in*, is because of the fact that rule entries in *Stride BVM RAM* are stored in a data-associative manner. Address bits of *Stride BVM RAM* are actually the header bits for the associated header stride, which makes *Stride BVM RAM* a data-associative memory. Hence, in order to write a rule entry to *Stride BVM RAM*, *Stride BVM Controller* firstly reads the corresponding data from *Ram_port_b_data_in* at address of header stride, then sets a single bit to HIGH determined by rule number (BVM address), without changing other bits. For example, in Fig. 4.7, *Stride BVM Controller* of $RuleBVM_0(0,0)$ is responsible for writing the first 16 rules (rule 0 to rule 15) to connected *Stride BVM RAM* for header stride of *ingress(31:24)*. If we assume a rule entry insertion at BVM address 3 (rule 3), where *ingress(31:24)* is *0x05* with exact match (unmasked), then, *Stride BVM Controller* of $RuleBVM_0(0,0)$ firstly reads the data at address *0x05* from *Stride BVM RAM* and writes *"XXXXXXXX_XXXX1XXX"* to address of *0x05*. *"X"* in write data means the unchanged bits at address *0x05*. The organization of *Stride BVM RAM* for $RuleBVM_0(0,0)$ is depicted in Fig. 4.8. The least significant bits at all addresses give information of the rule number 0 , where most significant bits give the information of rule number 15. Similarly, for $RuleBVM_0(0,1)$, least significant bits give the information of rule 0. However, for $RuleBVM_0(1,0)$, LSB gives the information for rule 16.

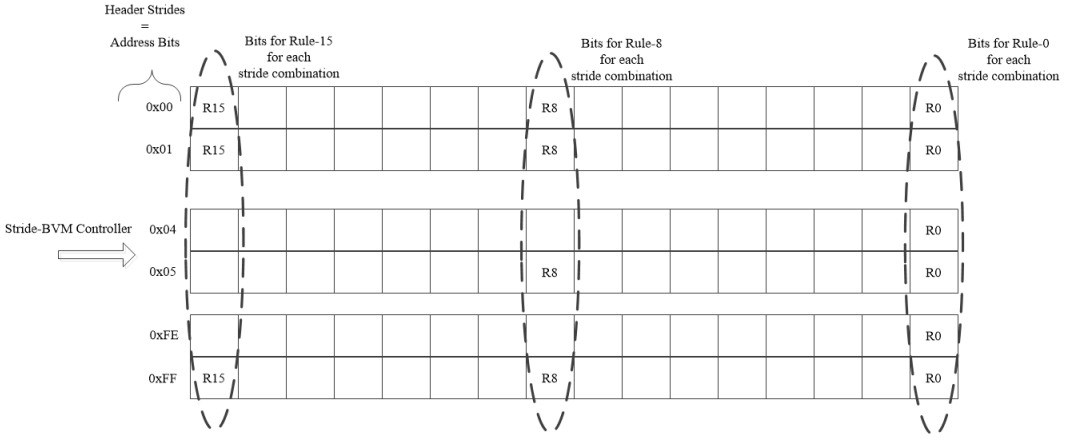

Figure 4.8: Address - Data organization of Stride-BVM RAM

In rule insertion phase, storing masked header fields to *Stride BVM RAM* is quite different from exact (unmasked) case. Wildcard match for a header field is signaled

with *mask_in* input port. For this case, *Stride BVM Controller* reads all data from all addresses from 0 to 255 at *Stride BVM RAM*, sets a single bit location at all these addresses again, without changing other bits.

The signals utilized in rule insertion phase are write enable signal, *bvm_wr_en*, write address signal, *bvm_wr_addr(3:0)*, address enable signal, *bvm_wr_addr_en(1:0)* and packet header information, *packet_header(355:0)*. As seen in Fig. 4.7, these signals are routed to all *Stride BVM Controller* inside $RuleBVM_0$ without pipelining. This routing indicates that rule insertion phase is carried out at the same time in all *Stride BVM* units without pipelining, which is different from query phase. Since the focus of this thesis is achieving a high throughput with a very low latency, rule updates such as inserting a new rule to BVM are performed in a multi-cycle process. Inserting a single rule to BVM takes a maximum of 256 cycles, because maximum bits in a header stride *s* is 8. For a masked header stride, $2^8 = 256$ gives the worst case rule insertion time.

Hardware architecture of internal *Rule BVM* is optimized to achieve high throughput in rule query phase. As described above, in query phase, local bit vector results are passed and bit-wise ANDed from one *Stride BVM* to another. For this purpose, read enable signal received from top level, which is *bvm_rd_en*, is only routed to *Stride BVM Controller* of $RuleBVM_0(0,0)$. After this time, each *Stride BVM Controller* passes the local bit vector results to another *Stride BVM Controller* horizontally and vertically at the same time. For this reason, *Stride BVM Controller* has *vertical_en*, *horizontal_en*, *bv_result_in(15:0)* input ports and *Ram_port_a_rd*, *bv_result(15:0)* output ports to perform pipelining. Input port *vertical_en* denotes a read enable signal received from a upper located *Stride BVM Controller* in vertical direction and input port *horizontal_en* denotes a read enable signal received from a previously located *Stride BVM Controller* in horizontal direction. Hence the connection diagram in Fig. 4.7 shows a direct connection from *Ram_port_a_rd* port of *Stride BVM Controller* of $RuleBVM_0(0,0)$ to *vertical_en* port of *Stride BVM Controller* of $RuleBVM_0(1,0)$ to indicate vertical pipelining. Similarly, connection from *Ram_port_a_rd* port of *Stride BVM Controller* of $RuleBVM_0(1,0)$ to *vertical_en* port of $RuleBVM_1(0,0)$ indicates the top level vertical pipeline connection between *Rule BVM* units. This connection is compatible with Fig. 4.4,

which shows the connection between $RuleBVM_0$ and $RuleBVM_1$. Moreover, for horizontal pipelining, direct connection from *Ram_port_a_rd* port of *Stride BVM Controller* of $RuleBVM_0(0,0)$ to *horizontal_en* port of *Stride BVM Controller* of $RuleBVM_0(0,1)$ is demonstrated in Fig. 4.7. In a similar manner, *Ram_port_a_rd* port of *Stride BVM Controller* of $RuleBVM_0(0,1)$ is connected to *horizontal_en* port of *Stride BVM Controller* of $Rule_BVM_0(0,2)$ to proceed in horizontal direction. One important note to mention here is that horizontal and vertical pipeline connection between *Stride BVM* units in a *Rule BVM* is made only at the first vertical stage. For example, connection of horizontal pipelines for $RuleBVM_0$ is made between *Stride BVM Controller* of $RuleBVM_0(0,0)$ and *Stride BVM Controller* of $RuleBVM_0(1,0)$. After that point, all horizontal read enable signals are connected between each other.

Furthermore, together with these pipeline connection of read enable signals, there are Flip-Flops(FF) located at each horizontal and vertical direction. FFs perform delaying operation on header strides. In other words, they provide the corresponding header stride to correct *Stride BVM Controller* at the correct time. This correct time means the assertion of corresponding read enable signal for a *Stride BVM Controller*. Since there must be a unique FF for each *Stride BVM*, there are 94 FFs in a *Rule BVM*. In horizontal direction, these FFs delay the header strides in *packet_header(355:0)* to meet timings.

At the same with delaying, these FFs also split *packet_header(355:0)* into header strides, which the next horizontal *Stride BVM Controller* is responsible for. For example, in Fig. 4.7, *Stride BVM Controller* of $RuleBVM_0(0,0)$ is responsible for *Ingress(31:24)* and *Stride BVM Controller* of $RuleBVM_0(0,1)$ is responsible for *Ingress(23:16)*. As a result, FF delays the subset of overall packet header instead of whole 356 bits at horizontal pipeline stages, which results in a delaying on *packet_header(347:0)* at the first stage in our case. Most significant 8-bits in header is *Ingress(31:24)*. This is because *Ingress(31:24)* will never be used in the following *Stride BVM Controller* units in horizontal direction, and there is no need to delay these bits. However, *Ingress(31:24)* will be used in other *Stride BVM* units located in lower vertical connection. Hence, FFs located between upper horizontal and lower horizontal pipeline perform one clock cycle delay for the same header strides. For

example, FF between *Stride BVM Controller* of $RuleBVM_0(0,0)$ and *Stride BVM Controller* of $RuleBVM_0(1,0)$ delays *Ingress(31:24)*.

Each *Stride BVM Controller* has *bv_result_in(15:0)* port to receive bit vector result from previous *Stride BVM Controller*. While maintaining pipeline processing, *bv_result(15:0)* is bit-wise ANDed with *bv_result_in(15:0)* to generate local bit vector match results, and the final bit vector is sent to *bv_result_in(15:0)* port of next *Stride BVM Controller*. This processing continues at each horizontal pipeline until *upper_pipeline_match(15:0)* and *lower_pipeline_match(15:0)* is provided. Note that; *upper_pipeline_match(15:0)* is actually *bv_result(15:0)* port of *Stride BVM Controller* of $RuleBVM_0(0,46)$ and *lower_pipeline_match(15:0)* is actually the output *bv_result(15:0)* of *Stride BVM Controller* of $RuleBVM_0(1,46)$, which is illustrated in Fig. 4.4.

4.4.2 Pipeline Processing Sequence at Signal level in FASST

In consideration of the information described in Section. 4.4 and Section. 4.4.1, pipeline processing sequence at signal level is explained below. For this processing sequence, it is assumed that; rule insertion is completed, BVM is already in query phase, *Ingress(31:24)* is *0x04* and *Ingress(23:16)* is *0xF1* for the first packet header. Moreover, it is assumed that only a single packet arrives to BVM for classification. Moreover, signal naming in sequence is completely compatible with Fig. 4.4 and Fig. 4.7. Therefore:

- At *Clock_cycle =n*, firstly, header bits arrive to $RuleBVM_0(0,0)$. Header bits *packet_header(355:348)* are equal to *Ingress(31:24)* and the value is *0x04*. At the same time, *bvm_rd_en* port, which is *horizontal_ren* input port of *Stride BVM Controller* of $RuleBVM_0(0,0)$, is set to '1'.
- At *Clock_cycle =n+1*, *Stride BVM Controller* of $RuleBVM_0(0,0)$ immediately drives its output ports as in the following, at the same time:
  - *Ram_port_a_addr = 0x04*
  - *Ram_port_a_rd = '1'*

- Furthermore, at *Clock_cycle =n+1*, *packet_header(355:0)* is also delayed and split into *packet_header(347:0)* by the first horizontal FF. Hence the assertion *Ram_port_a_rd = '1'* and *Ingress(23:16)*, which is *packet_header(347:340)* occurs at the same clock cycle. In other words, *Stride BVM Controller* of $RuleBVM_0(0,1)$ detects the assertion of *horizontal_ren* and *Ingress(23:16)* header stride at the same time.

- Similarly, at *Clock_cycle =n+1*, *packet_header(355:348)*, which is header stride of *Ingress(31:24)*, is delayed by the first FF in vertical direction. Hence, *Stride BVM Controller* of $RuleBVM_0(1,0)$ detects the assertion of *vertical_ren* port and *Ingress(31:24)* header stride at the same time.

- At *Clock_cycle =n+2 Stride BVM Controller* of $RuleBVM_0(0,0)$ receives the data content at address *0x04* via *Ram_port_a_data_in*. This content provides a 16-bit data for rule 0 to rule 15 to indicate whether any of the rules in this range has *Ingress(31:24) = 0x04* value.

- At *Clock_cycle =n+2*, *Stride BVM Controller* of $RuleBVM_0(0,0)$ performs bit-wise AND operation between the read data through *Ram_port_a_data_in* and *bv_result_in(15:0)* and sends the results over *bv_result*.

- At *Clock_cycle =n+2*, *Stride BVM Controller* of $RuleBVM_0(0,1)$ immediately drives its output ports as in the following, at the same time:

  - *Ram_port_a_addr = 0xF1*
  - *Ram_port_a_rd = '1'*

- At *Clock_cycle =n+2*, *Stride BVM Controller* of $RuleBVM_0(1,0)$ immediately drives its output ports as in the following, at the same time:

  - *Ram_port_a_addr = 0x04*
  - *Ram_port_a_rd = '1'*

- At *Clock_cycle =n+3*, *Stride BVM Controller* of $RuleBVM_0(0,1)$ performs bit-wise AND operation between the read data through *Ram_port_a_data_in* and *bv_result_in(15:0)* and sends the results over *bv_result*. Note that; previous match result, *bv_result_in(15:0)*, comes from *Stride BVM Controller* of $RuleBVM_0(0,0)$ for this case.

- At *Clock_cycle =n+3*, *Stride BVM Controller* of $RuleBVM_0(1,0)$ performs bit-wise AND operation between the read data through *Ram_port_a_data_in* and *bv_result_in(15:0)* and sends the results over *bv_result*.
- The processing sequence for one *Rule BVM* continues until *Stride BVM Controller* of $RuleBVM_0(1,46)$ performs bit-wise AND operation between the read data of *Ram_port_a_data_in* and *bv_result_in(15:0)* and sends the results to *POEnc* units.

The timing diagram for the processing processing sequence explained above is depicted in Fig. 4.9. Since *Stride BVM Controller* of $RuleBVM_0(0,0)$ and *Stride BVM Controller* of $RuleBVM_0(1,0)$ are the first *Stride BVM* units in horizontal pipeline stages, then *bv_result* is bit-wise ANDed with 0xFFFF for these blocks.

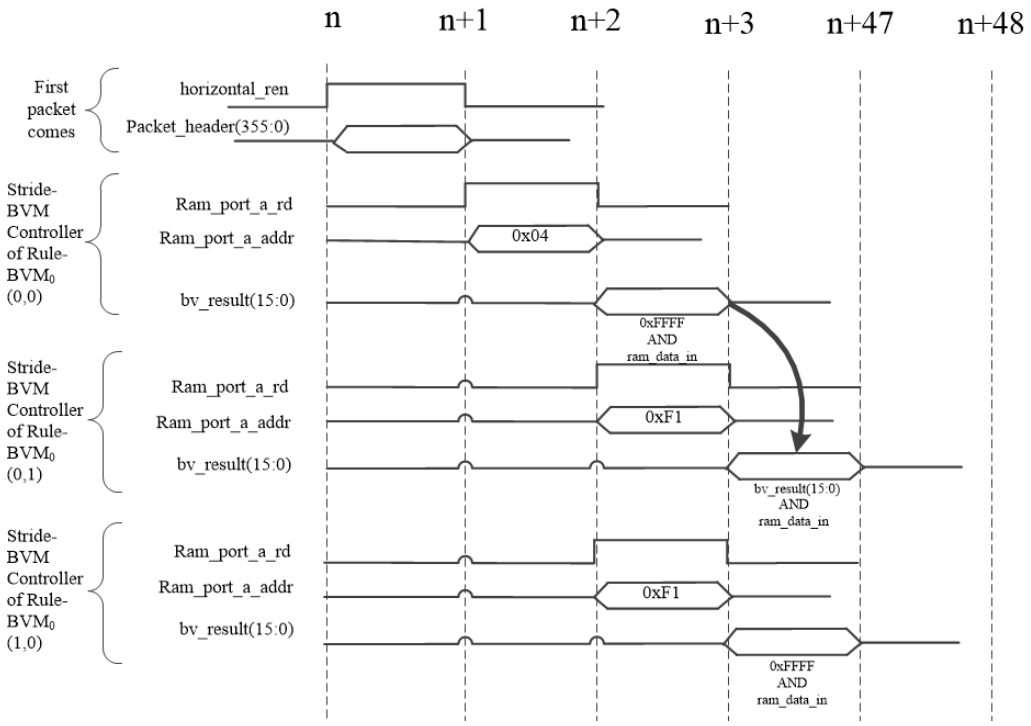

Figure 4.9: Signal level timing diagram for pipeline processing inside Rule-BVM

Functional simulation results for 3 *Stride BVM* units are presented in Fig. 4.10. The signals presented here are actually the ports of *Stride BVM Controller*s of the units $RuleBVM_0(0,0)$, $RuleBVM_0(0,1)$ and $RuleBVM_0(1,0)$. As observed in Fig. 4.10, top level read enable signal, *bvm_rd_en*, is directly connected to *horizontal_ren* of $RuleBVM_0(0,0)$. After a single clock cycle, *horizontal_ren* of $RuleBVM_0(0,1)$

and *vertical_ren* of $RuleBVM_0(1,0)$ assert HIGH, which is compatible with the connection diagram depicted in Fig. 4.7. As a result, the interval between the assertion of *bv_result(15:0)* of *Stride BVM* units is only one clock cycle. Since both $RuleBVM_0(0,1)$ and $RuleBVM_0(1,0)$ is connected to $RuleBVM_0(0,0)$ with a single pipeline stage, *bv_result(15:0)* for these two blocks asserts at the same time.

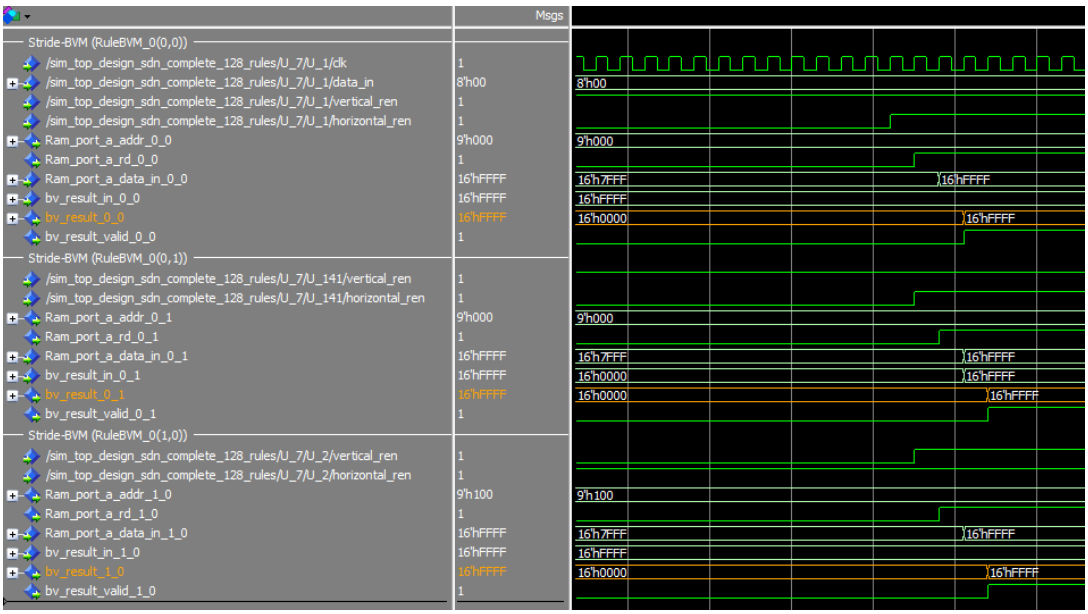

Figure 4.10: Functional simulation of 3 Stride-BVMs with pipeline connection

4.5 Implementation of TCAM

4.5.1 Details of Stride TCAM Blocks

FPGA implementation of TCAM in FASST is very similar to the design of a single *Rule BVM* except for some minor differences. Similar to *Stride BVM* units in BVM, there are *Stride TCAM* blocks inside TCAM to store and query rules. Each *Stride TCAM* consists of one *Stride TCAM Controller* and one *Stride TCAM RAM*. However, different from *Stride BVM*, *(Stride TCAM Controller, Stride-TCAM RAM)* design shows variations based on the header fields. The depth of *Stride BVM RAM* is independent of header stride size including IP fields in BVM, and it is always 256. In TCAM; however, header strides of IP fields are taken as single bits (s=1). This variation is caused from abundant parallelism in FPGA, as well as different combinations

of prefix lengths on IP fields during query phase. Hence, a total of 64 *Stride TCAM* units is used for 32-bit IP source and 32-bit IP Destination fields, where the depth of *Stride TCAM RAM* for these fields is $2^1 = 2$: one for logic '0' and one for logic '1'. Other remaining 13 fields in a 15-tuple SDN packet, *(Stride TCAM Controller, Stride TCAM RAM)* design is the same as *(Stride BVM Controller, Stride-BVM RAM)*. A total of 39 *Stride TCAM* units is implemented for these 13 fields. The width of *Stride TCAM RAM*s in TCAM is 32, since TCAM can store and classify 32 rules.

Detailed implementation diagram of TCAM except for *Stride TCAM* units for IP fields is presented in Fig. 4.11. TCAM implementation can be regarded as a single horizontal stage of *Rule BVM* without pipelining. In order to preserve design integrity, same functional blocks are used for *Stride TCAM Controller* and *Stride BVM Controller* with small modifications. The differences are given as in the following:

- Compared to *Stride BVM* in BVM, each *Stride TCAM* receives the header stride at the same time in TCAM. In other words, there is no pipeline connection of read enable signals. For this purpose, read enable signal *tcam_rd_en* from top level is routed to all *horizontal_ren* ports of *Stride TCAM Controller* units. At this point, *vertical_ren* can also be utilized. However, in our implementation, we set *vertical_ren* to '1' in all *Stride-TCAM Controller* units.
- There are no FFs to delay the packet headers in query phase due to the fact that all header strides are queried at the same time.
- Since there is no pipelined connection of read enable signals, *Ram_port_a_rd* output port is only routed to the associated *Stride TCAM RAM*.
- The width of bit vector match results is changed to 32-bits instead of 16-bits, which are *bv_result_in(31:0)* and *bv_result(31:0)*
- There is no bit-wise ANDed local bit vectors in pipeline stages. Therefore, *bv_result_in(31:0)* is set to default value 0xFFFFFFFF.
- Since *Stride TCAM Controller* units are only lying in the horizontal direction without pipelining connection, then, address enable port, *bvm_wr_addr_en*, to select between upper and lower horizontal stages is set to '1' to activate the blocks.

- Input port *tcam_wr_en* and *tcam_wr_addr* in Fig. 4.11 denote *bvm_wr_en* and *bvm_wr_addr* in Fig. 4.7. However, *tcam_wr_addr* is 5-bits long instead of 4-bits.

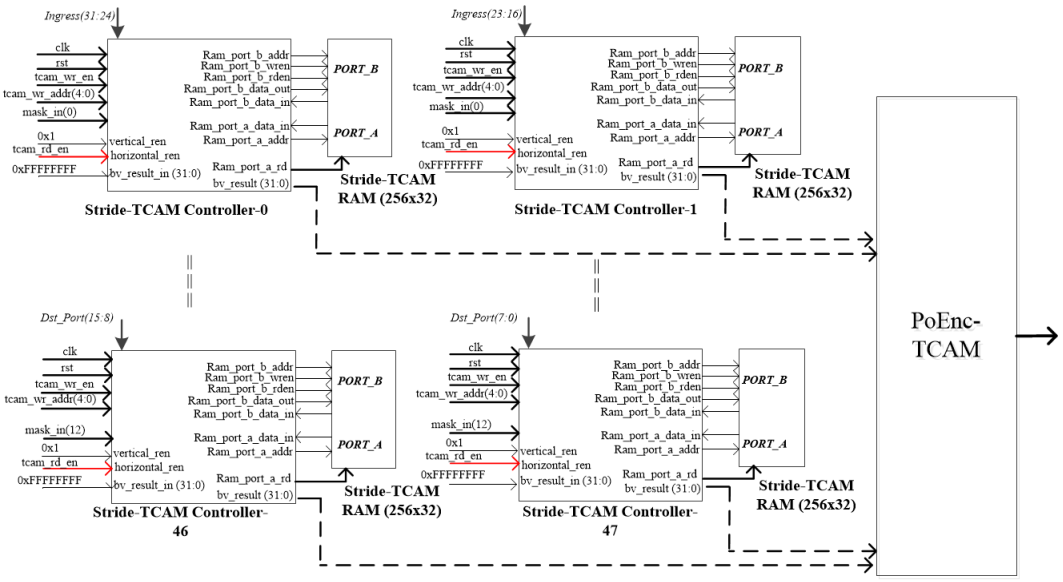

Figure 4.11: Stride-TCAM design in TCAM

The process of rule insertion to TCAM is the same as in BVM. *Port_A* and *Port_B* of *Stride TCAM RAM* are used for rule insertion and rule query phases, respectively. During rule insertion, each *Stride TCAM Controller* firstly reads the data from *Port_A*, modifies it by setting a single bit to '1' for each rule and writes the modified data back to *Stride TCAM RAM*. Similarly, masking fields are signaled with *mask_in* input port.

In rule query, all *Stride TCAM Controller* blocks assert read enable and address ports, which are *Ram_port_a_rd* and *Ram_port_a_addr*, at the same time. Note that *Ram_port_a_addr* is the header stride of the incoming packets, and different for all *Stride TCAM Controller* blocks. After a single clock cycle, *Stride TCAM Controller* units assign the value of *Ram_port_a_data_in* to the output port *bv_result(31:0)* at the same time. Hence, after one clock cycle, the bit vector results for all header strides are obtained.

Stride TCAM Controller blocks for IP fields perform similarly in rule insertion and rule query phase. The only difference is that the depth of *Stride TCAM RAM* is 2 due to 1 bit header stride, s=1. In FPGA, *Stride TCAM RAM* is implemented using logic

gates instead of embedded memory blocks. This is due to the fact that there is no *2x32* embedded on-chip ram block inside FPGA.

Functional simulation results for *Stride TCAM-0*, *Stride TCAM-4* and *Stride TCAM-76* are illustrated in Fig. 4.12. *Stride TCAM-0* and *Stride TCAM-4* are for the header strides of *Ingress(31:24)* and *Metadata(63:56)*, respectively. *Stride TCAM-76* is for bit 0 of IP Destination field. Different from *Stride BVM*, *horizontal_ren* ports of all *Stride TCAM* units assert at the same time without pipeline connection. As a result, *bv_result(31:0)* ports provide the bit vector match results at the same time except for *Stride TCAM-76* for Destination IP. This is due to logic gate implementation of *Stride TCAM RAM* for IP fields. Since logic gates are used for IP fields instead of embedded memory blocks, additional clock cycle coming from the read latency of embedded memory blocks is avoided. While logic gate-based memories can provide the data in single clock cycle, embedded memory blocks can provide in 2 clock cycles. Therefore, bit vector match results for IP fields are provided one clock cycle before other match results for 13-fields. End to end latency of lookup process in TCAM without priority encoding is 2 clock cycles.

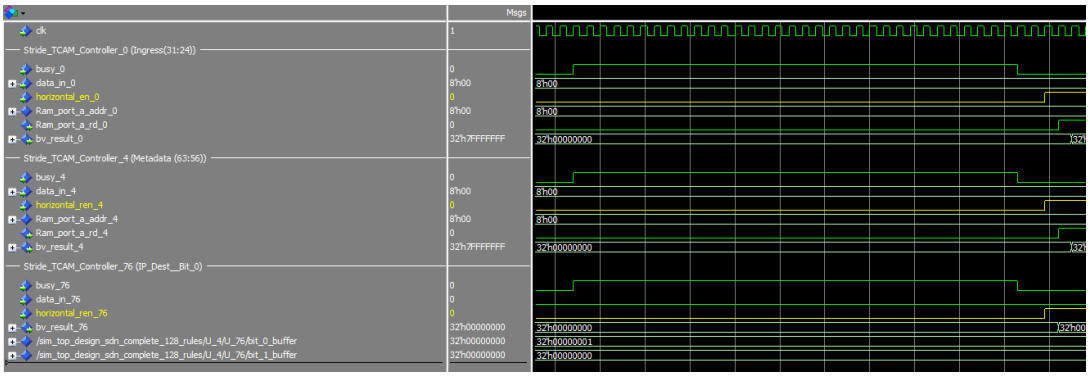

Figure 4.12: Functional simulation of Stride-TCAMs

4.5.2 Implementation of Priority Encoder in TCAM Cache

Priority Encoder in TCAM cache is implemented to obtain the index value in the matching bit vector with the highest priority. The increased number of rules stored in a TCAM results in a significant degradation in the achievable clock rate due to encoder functionality [37]. Therefore, TCAM size is limited to 32 rules. As seen in

Fig. 4.11, *bv_result(31:0)* ports of all *Stride TCAM Controller* units are connected to priority encoder in TCAM, named *POEnc-TCAM*.

In order to find highest priority match, *POEnc-TCAM* needs two inputs of rule information: <Rule ID><Rule Priority> pairs of the rules in TCAM and bit-wise ANDed bit vector match results from (39+64) *Stride TCAM Controller* units. As stated above, 64 of these *Stride TCAM Controller* blocks perform look up for IP Source and IP Destination fields with a single bit stride, and 39 of *Stride TCAM Controller* blocks perform lookup for other 13-fields with different header strides.

The implementation of *POEnc-TCAM* is designed as a state machine to make a separation between configuration and run-time encoding processes. Configuration of *POEnc-TCAM* is necessary because during a rule insertion to TCAM, *POEnc-TCAM* sorts the <Rule ID>s of the rules based on their priorities in descending order. Hence, there are two on-chip RAMs dedicated to 32 <Rule ID><Rule Priority> pairs in TCAM. The width of these RAMs is 9-bits, because both <Rule ID> and <Rule Priority> are represented as 9-bits, within the range (0,511). The depth of these RAMs is 32 words, which is the maximum number of rules that can be written to TCAM. Note that, there is no need to fill all 32 words in these RAMs, because the number of rules that are written to TCAM are determined by the number of popular rules and their dependent rules, as explained in section. 3.5.

Sorting of 32 rules takes a maximum of $32x32 = 1024$ clock cycles due to implementation of fully synchronous bubble sorting algorithm in FPGA. After rule sorting, run-time encoding phase begins. In this phase, after bit-wise ANDing of all *bv_result(31:0)* ports is performed to obtain final bit vector, highest priority match index is found. Total end to end latency of *POEnc-TCAM* is 1 clock cycle, which results in $t_{fast} = 3$ cycles for TCAM lookup.

Highest priority match is found by analyzing *HIGH* bits in the final bit-wise ANDed bit vector. Since, priorities are sorted in descending order, the index values of the highest priority rules are already known by *POEnc-TCAM*. Therefore, starting from highest index, a match result (HIGH bit) is checked inside bit vector. In case of a bit match (HIGH) at this highest index, all other bits except for this bit are set to logic LOW, and the output bit vector is sent from *POEnc-TCAM*. If bit value is LOW at

the highest index, then other indexes with lower priorities are searched to observe a match. Implementation details of *POEnc-TCAM* is demonstrated in Fig. 4.13.

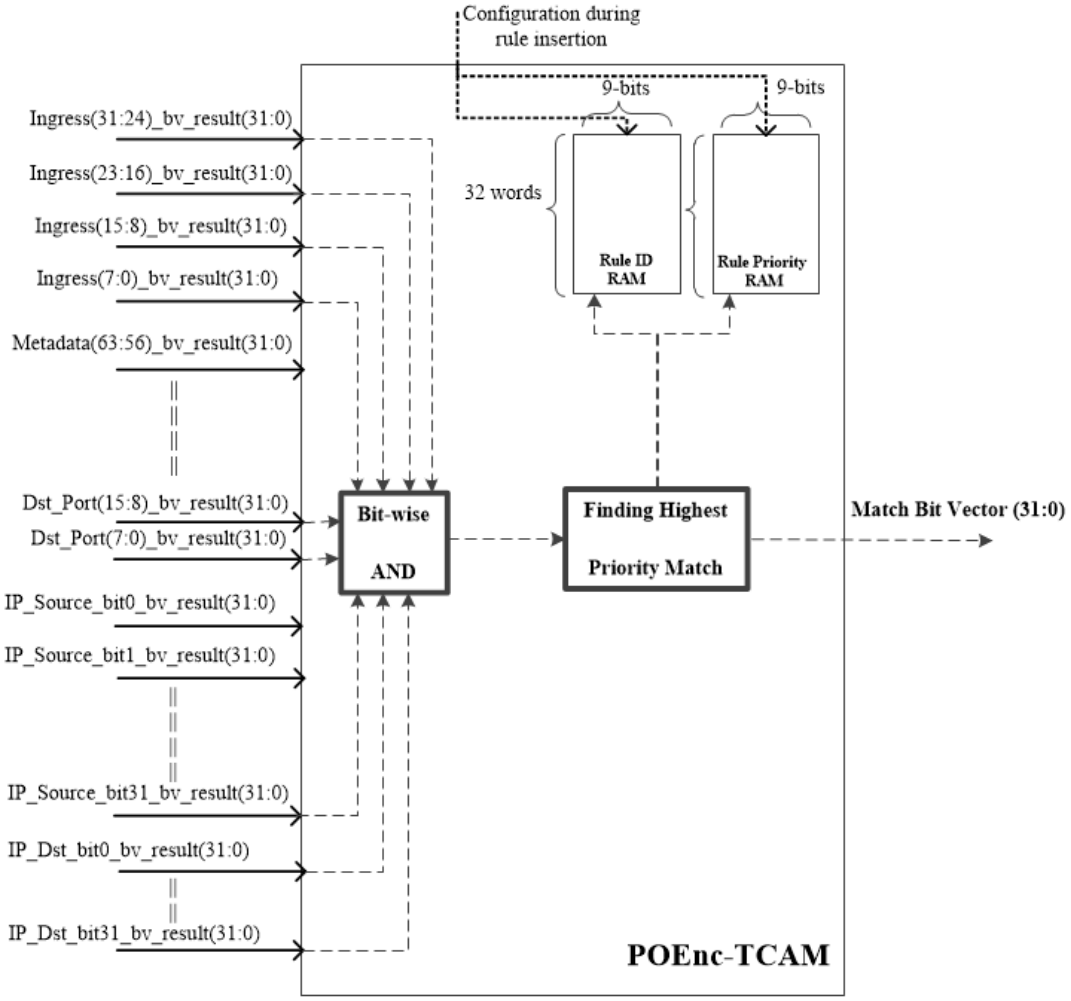

Figure 4.13: PO-Enc design in TCAM

Simulation waveform for *POEnc-TCAM* is given in Fig. 4.14. Note that, in Fig. 4.14, bit-wise ANDing is performed as a sequential clocked process in order to demonstrate the timings in a more clear way. This sequential process leads to an extra clock cycle delay. However, in current design of FASST, bit-wise ANDing is carried out using combinational gates. *BitWise_AND_valid* denotes the output of 'and' gate implementation inside *POEnc-TCAM*. This signal asserts HIGH one clock cycle after local bit vectors arrive to *POEnc-TCAM* due to sequential clocked process. After a single clock cycle, *POEnc-TCAM* provides the encoded bit vector result by setting *TCAM_match_valid* to HIGH. In Fig. 4.14, final bit vector result is shown

as *TCAM_Matched_Bit_Vector(31:0)*. Moreover, the associated <Rule ID> of the matched rules in TCAM is given as *TCAM_Matched_Rule_ID(8:0)*. For example, there are 4 match results for the rule with <Rule ID> =7. Mapping of bit vectors to <Rule ID>s is performed using a <Address><Rule ID> embedded memory. <Address> is actually the bit index in the final bit vector provided by *POEnc-TCAM*.

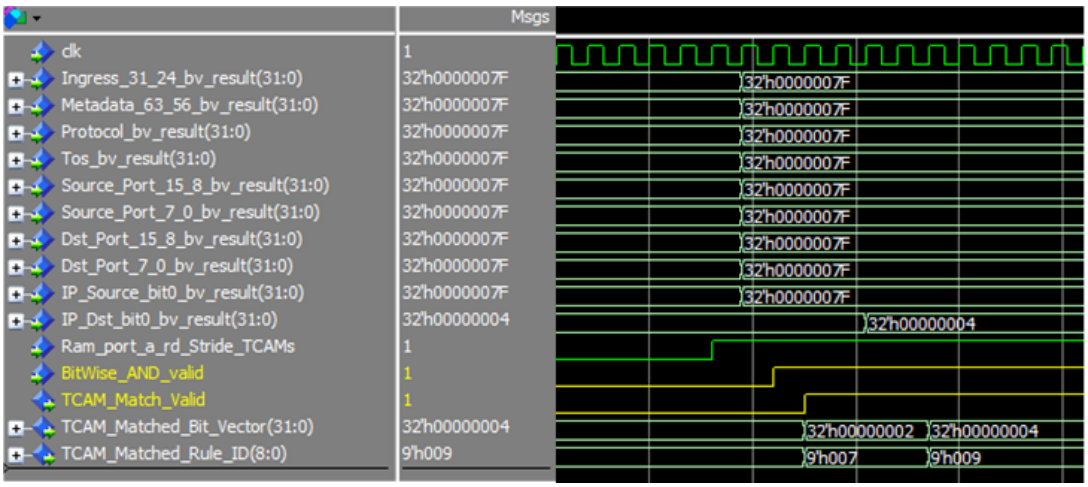

Figure 4.14: Functional simulation of PO-Enc in TCAM

4.6 Implementation Match Monitor (MM)

4.6.1 Locality Detector

In FASST, Locality Detector in Match Monitor (MM) checks whether the number of matches for a particular rule exceeds a predetermined threshold value Thr. If threshold value is exceeded, then, this rule is identified as a popular rule by MM and corresponding BVM address is written to RAM-1 as depicted in Fig. 4.1.

FPGA design of Locality Detector is given in Fig. 4.15. Since FASST stores 512 rules in BVM, each of these rules has to be analyzed to be identified as popular rules. Therefore, blocks named *windowed_rule_block* count the number of matches of each rule over a time window. Since locality information in a network traffic can change, *windowed_rule_block* updates the match results at two points in time window. Furthermore, the input to these blocks is BVM addresses of rule entries. For example, when *windowed_rule_block(2)* outputs a match count that exceeds threshold, it means

that entry F_j at BVM address=2 is a frequently accessed rule.

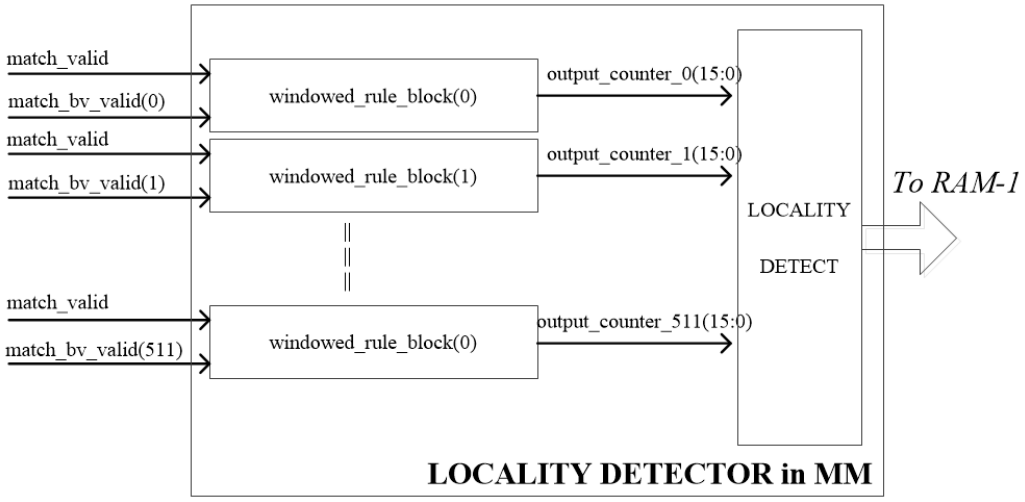

Figure 4.15: Internal architecture of Locality Detector

The duration of time window is expressed with respect to total match counts. A *time_window_counter* is used for this purpose. In other words, when a packet, P, matches a rule entry, then *time_window_counter* in each *windowed_rule_block* increments its value regardless of the BVM address of the matched rule. This matching is signaled with *match_valid* in Fig. 4.15. However, if the associated rule for *windowed_rule_block* has a match, then, particular counters for this rule increment their values to make a comparison with threshold level. The matching with a particular rule for each *windowed_rule_block* is signaled via *match_bv_valid*. The top level decoders designed to obtain this valid signal are not shown here. Basically these decoders takes 16-bit final bit vector and 32-bit group information register as described in Section. 3.3, and outputs a single valid signal for each rule at a time.

In *windowed_rule_block*, using only one counter to determine match counts can cause several inefficiencies. This is because, in FASST, sampling point to detect popular rules can be anywhere in time window. If the sampling point time and dynamic counter initialization overlap, then, it will be impossible to detect popular rules. This is because, dynamic counter is initialized to 0 to start a new time window at this time. Therefore, in *windowed_rule_block*, there are two implemented counters, *counter_1* and *counter_2*, to count the match numbers of the associated rule number. The incre-

mentation of these two counters is managed with a simple finite state machine (FSM) such that sampling point always observes a match count for the recent past.

Moreover, since *windowed_rule_block* has only one counter value as an output, the assignment of two counters to this output counter has to be managed in a controlled way. Hence, an example scenario for the assignment of output counter is illustrated in Fig. 4.16. For this scenario, time window duration is assumed to 2000 match counts. Hence upper limit on *time_window_counter* is 2000. Moreover, linear increase in match counts for the associated <BVM address> is assumed. Therefore, first counter and second counter increment their values linearly throughout the time window. The description of the assignments is given below:

- Only *counter_1*, which is first counter, is assigned to output counter value throughout the first time window.
- Second counter, *counter_2*, waits until the middle point of first time window, where *time_window_counter* is 1000. After that middle point, *counter_2* starts to increment its value.
- At the end of first time window, where *time_window_counter* reaches to 2000, *counter_1* initializes to 0. At that time, *counter_2* is assigned to output counter value. This assignment continues until the middle point of second time window.
- At the middle point of second time window, *counter_2* is initialized to 0, and *counter_1* is assigned to output counter value until the starting point of third time window. At this time, *counter_2* starts to count from 0 again.
- At the beginning of third time window (or at the end of second time window), *counter_1* is again initialized to 0, and starts to count from 0. At this time, *counter_2* is assigned to output counter value until middle of third time window.
- The assignment process proceeds similarly for each time window. In summary, except for the first time window, assignment order changes at starting and middle points of time windows. Therefore, any sampling point in time windows observes a traffic measurement over the recent past, where this recent past denotes 1000 match results in our case.

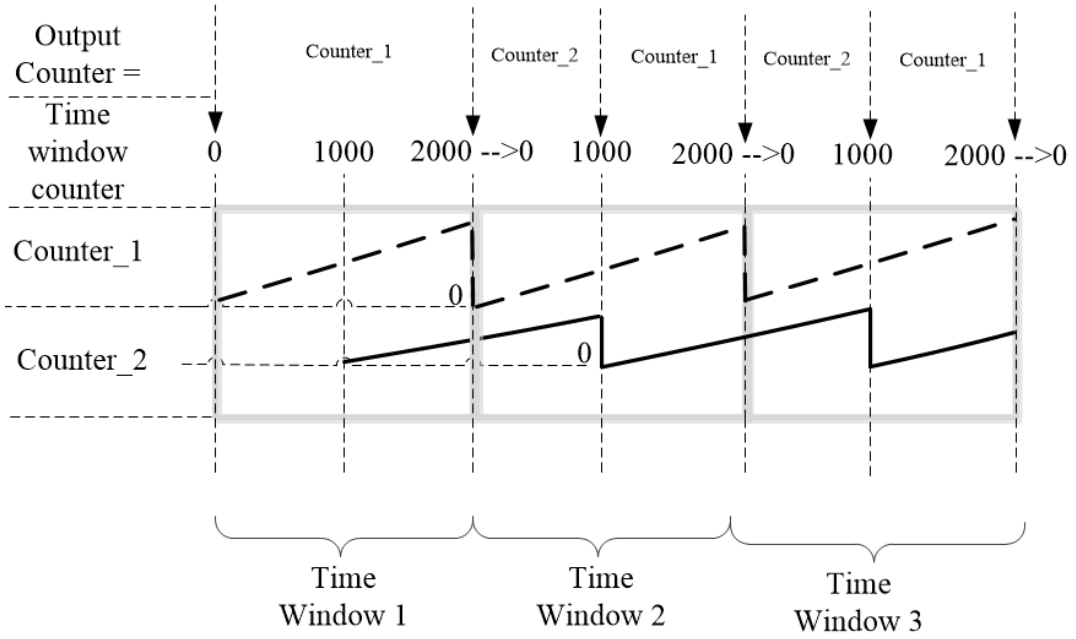

Figure 4.16: Output counter mapping in windowed rule block

A rule is identified as a popular rule when the match count exceeds the predetermined threshold level Thr. This count can be observed by analyzing the output counter value for each rule as seen in Fig. 4.16. *Locality Detect* checks each output counter in order, and writes the BVM address to RAM-1 at the associated location, if current value of output counter exceeds Thr. This checking process begins from BVM address=0. The match count for the rule F_j, with at BVM address=0, is provided by *windowed_rule_block(0)*.

Functional simulation result in Fig. 4.17 provides an example of writing popular BVM addresses to RAM-1. In this example, BVM addresses of the popular rules are 5, 9, 62 and 135. As observed, all output counter values for these BVM addresses exceed Thr, which is 100 match counts. Hence, *Locality Detect* writes the BVM addresses of these rules to on-chip RAM-1 through write enable signal *ram_1_wren*, address bus *ram_1_addr(8:0)*, and data bus *ram_1_data(9:0)*. Note that, after writing BVM address of 9, the match count for BVM address of 22 is observed as 51, which is below the threshold level Thr. Hence, *Locality Detect* unit skips this address and proceeds to write BVM address of 62. Moreover, address of RAM-1 is incremented by 8 bytes for each popular rule due to subparts of 8-words in RAM-1. For example, BVM addresses of 5, 9 and 62 are written to addresses of 1, 9, and 17 at RAM-1. The

remaining addresses are filled with dependent BVM addresses of rules.

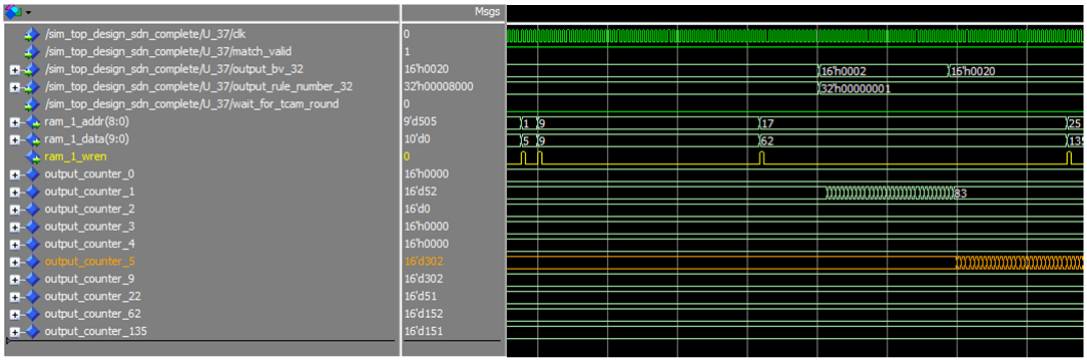

Figure 4.17: Functional simulation of locality detection

4.6.2 NIOS II Soft Processor System-on-Chip (SoC) Design

The function of processor block in MM is to determine the dependent BVM addresses of the popular rules written to RAM-1. For FPGA implementation, this processor is designed using a System-on-Chip (SoC) integration tool named *Qsys*, which is specific to FPGA vendor *Altera*. This SoC tool provides a flexible and efficient environment to connect different intellectual property (IP) functions and subsystems. Moreover, interconnection logic is automatically generated by this integration. Therefore, complex network on chip (NoC) architectures are designed and verified using *Qsys* in many fields.

Processor unit in MM performs a variety of algorithmic functions at run time such as rule dependency graph generation and depth first searching as described in section. 3.5. Using a hardware description language (HDL) for these functions is redundant and time-consuming. For this reason, a soft processor, called Nios II IP, is implemented using *Qsys*. Nios II IP is 32-bit RISC soft processor core architecture designed specifically for the FPGA families of Altera. This processor architecture is named as soft processor core due to the fact that it can fit anywhere in FPGA programmable logic. Similar to conventional processor cores, Nios II IP provides support for embedded computing applications using high level languages, command shells, and memory management units .

Nios II IP provides only a soft processor core in FASST. However, as described in Fig. 4.1, a SoC is implemented in MM processor unit to make connections for on chip RAMs. As a result, custom peripherals are connected to Nios II soft core in order to communicate with external RAMs. These custom peripherals are specifically designed to FASST regarding timing constraints and on chip RAM size. The interface diagram of processor SoC, consisting of Nios II IP and custom peripherals, is given in Fig. 4.18.

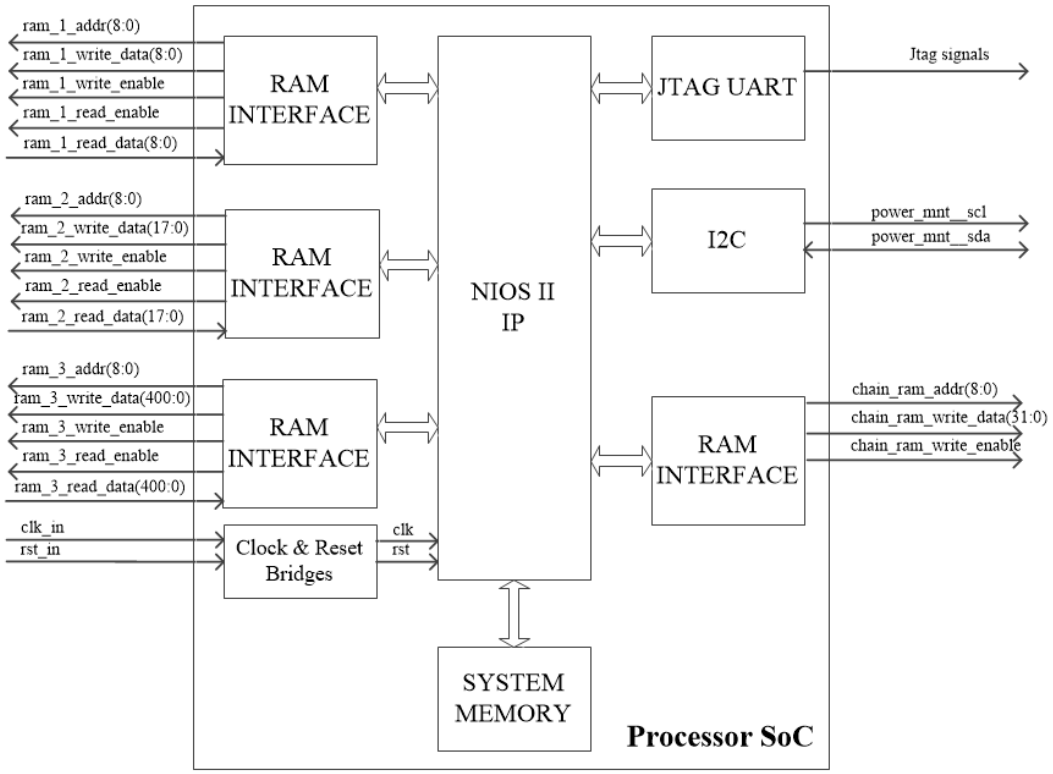

Figure 4.18: SoC design of Nios II in MM

The custom interfaces, called *RAM Interface*, provide a memory mapped access between Nios II IP soft processor core and external on chip RAMs. Each *RAM Interface* has a number of parameters such as number of pipeline transfers, required read latency and required write latency. As a result, 4 *RAM Interface* units are utilized and configured in order to communicate with RAM-1, RAM-2, RAM-3 and chain_RAM. Moreover, clock and reset signals generated outside MM is sent to Nios II IP through *Clock&Reset Bridges* in order to make an isolation. Nios II IP has direct connection to system memory, which is used to store data and instructions. Regarding FASST

algorithms, the size of system memory is set to 256 Kb. Processor SoC also has a custom interface called *JTAG UART*. This interface is added for debug purposes of the FASST architecture on run time. *JTAG UART* provides a convenient way to communicate with host PC's console in order to display real time test results.

Since FASST is a stand-alone hardware architecture that is implemented on FPGA, power consumption is monitored at run time. For this reason, there is a custom interface called *I2C*, which means Inter-Integrated Circuit. I2C is a common protocol used in industry in order to communicate with a variety of devices including sensors, actuators over a serial interface. There two signals utilized in I2C protocol: clock signal, named as *scl*, and bidirectional data signal, named as *sda*. Since our targeted development board includes power monitor devices that are compatible with I2C communication protocol, Nios II IP continuously reads power measurements and reports them to host PC. The signals used for I2C communication are *power_mnt_scl* and *power_mnt_sda*.

In order to generate rule dependency graph, and determine dependent rules, Algorithm 1 and Algorithm 2 are run in Nios II IP. According to the results, $78Kb$ system memory is used for all these algorithms, and the remaining $256 - 78 = 178Kb$ is free for stack and heap memory.

The connection diagram of processor SoC in *Qsys*, excluding I2C interface, is given in Fig. 4.19. All interfaces are connected to each other through Altera specific memory mapped protocol called *Avalon Memory Mapped Interface*. Nios II IP is the only master in SoC that sends instructions to other custom interfaces.

4.6.3 Implementation of TCAM Cache Interface (TCAM Writer Block)

In FASST, TCAM Cache Interface is responsible for writing the popular rules and their dependent rules to TCAM at run-time. This block has 5 ram interfaces to perform rule insertion process as illustrated in Fig. 4.20. FPGA implementation of TCAM Cache Interface is designed as a series of sequential processes using finite state machines. The process sequence, starting from the completion of writing popular BVM addresses to RAM-1, is explained in detailed below:

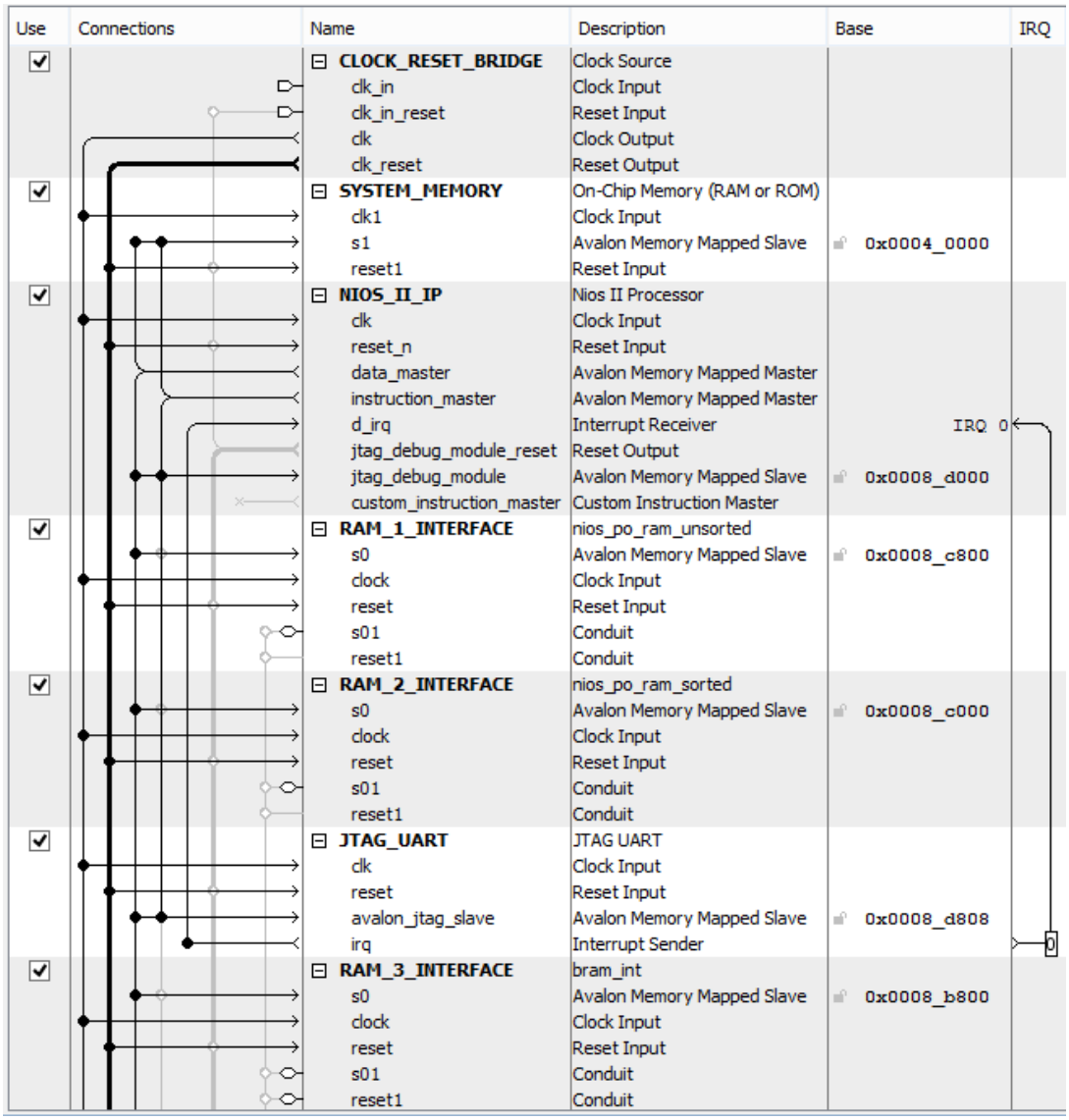

Figure 4.19: System contents in MM Processor in Qsys

- TCAM Cache Interface polls bit 0 at $Address_0$ of RAM-1 in order to determine whether completion of writing popular BVM addresses is completed by *Locality Detect*. A HIGH value at bit 0 means a completion of writing.

- Following the completion of writing popular BVM addresses, TCAM Cache Interface cleans bit 0 at $Address_0$ of RAM-1 and waits for Nios II IP to identify dependent rules of BVM addresses for each popular rule. This waiting process is done by polling bit 0 at $Address_{511}$ of RAM-1. Note that rule dependency graph is already generated by Nios II IP following the insertion of all 511 rules to BVM. Hence, it only performs depth-first searching on the graph to find

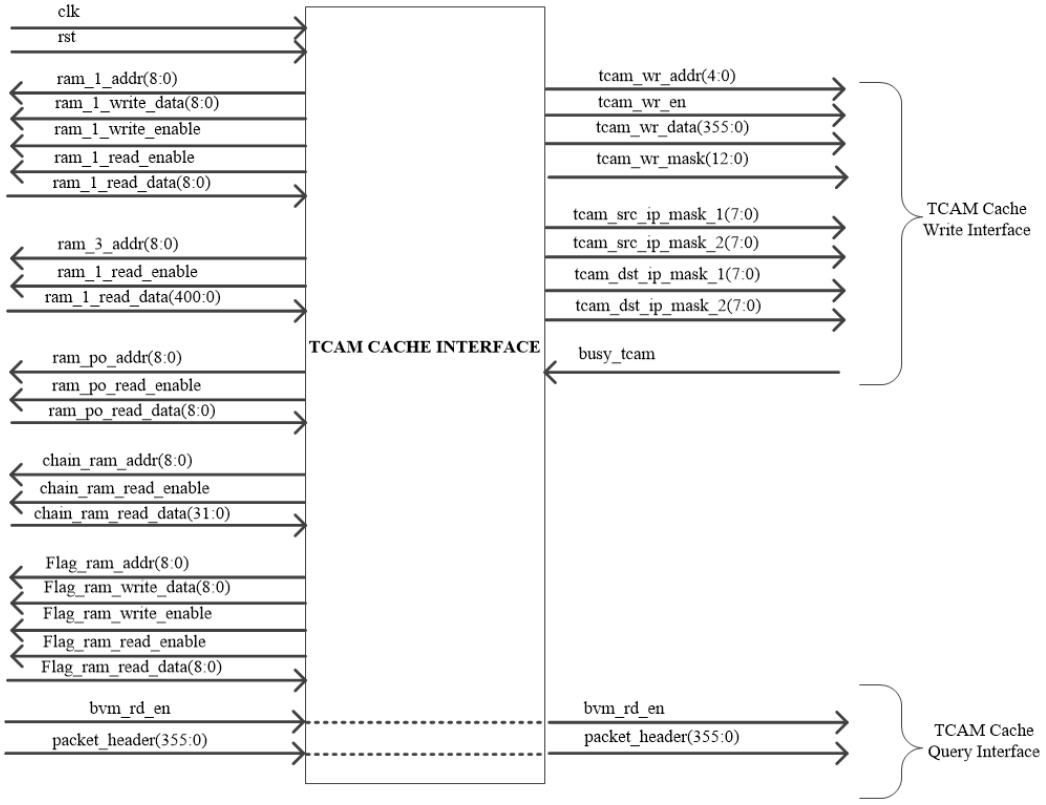

Figure 4.20: Interface diagram of TCAM Cache Interface

dependencies at this time.

- After that, when bit 0 at $Address_{511}$ of RAM-1 asserts HIGH, TCAM Cache Interface cleans this bit, and starts to read BVM addresses from RAM-1 starting from $Address_1$. At this time, it searches *Flag_RAM* to figure out whether this rule is written to TCAM before. This is done by analyzing flag bits in *Flag_RAM*. For each rule in BVM, there is a unique flag bit in *Flag_RAM*, which indicates the status of rules in TCAM. If flag bit for rule entry F_j is HIGH, it means that F_j is written to TCAM before. Moreover, if F_j is written, it means that all dependent rules are also written. Hence, TCAM Cache Interface skips this rule and its dependent rules and it reads the next popular rule, if it exists, at $Address_9$ of RAM-1.

- Searching process in *Flag_RAM* lasts only single clock cycle, because BVM addresses are used as address bits in *Flag_RAM*.

- If the current popular rule at $Address_1$ is not written to TCAM before, then

TCAM Cache Interface reads the number of dependent rules of this current popular rule from *chain_RAM*. This information is needed in order not to make an overflow in TCAM. In other words, if the number of dependent rules of the current popular rule exceeds the number of available empty locations in TCAM at some time, then, this current rule and its dependent rules are skipped and they are not stored. In order to preserve locality information, popular rules must be stored in TCAM together with all their dependent rules.

- If the number of dependent rules is less than the number of available slots in TCAM, then current popular rule and its dependent rules are written through *TCAM Cache Write Interface*. At this time, *Flag_RAM* is again accessed for each dependent rule. This is because, in depth first search, some rules can be visited more than once, and a labeling is necessary to avoid overwriting.

- Before reading next popular rule and its dependent rules, TCAM Cache Interface waits for busy signal, *busy_tcam*, to deassert LOW. This signal is driven by TCAM internal logic. Since the rules in TCAM can have wildcard match fields, this waiting time can take up to 256 clock cycles due to using 8 bit header strides.

- The produce of writing rules to TCAM continues until either TCAM is full or there are no popular rules left in RAM-1.

- Before new locality information, TCAM Cache Interface passes read enable signal *bvm_rd_en*, and packet headers, *packet_header(355:0)* to TCAM in order to perform rule query. This is because, BVM and TCAM perform lookup process in parallel in query phase. When locality information is updated in FASST, TCAM Cache Interface cleans *Flag_RAM*, and polls bit 0 of the data at the $Address_0$ of RAM-1 again.

CHAPTER 5

PERFORMANCE EVALUATION OF FASST

The overall design FASST is functionally verified at the end of design procedure. For this purpose, all functional blocks and Nios II SoC are simulated using QuestaSim, which is an advanced simulator for verification. Moreover, since FASST is implemented in real hardware using Altera FPGA development kit, hardware tests are also carried out in order to indicate that overall design works under targeted clock frequencies to achieve the desired throughput and latencies. Most of the hardware based SDN classification algorithms proposed in previous works are only implemented in FPGA and the results are presented by estimation [8], [15]. However, in our work, run-time debugging tools on real hardware are used.

The obtained results are used in order to calculate the throughput and average latency of packets in rule query phase. These two parameters are strictly dependent on clock rate in FPGA and cache hit rate in TCAM as described in chapter. 3 and chapter 4. Therefore, while carrying out static (functional) and dynamic (hardware) tests, the network characteristics including BVM rule set $\mathcal{R}$ and TCAM rule set $\mathcal{F}$ is extremely important to verify the overall design at each phase. For this purpose, sample traffic traces and SDN rules are generated and sent to FASST. While generating these traffic traces and SDN rules, packet formats in Table. 4.1 and Table. 4.2 are taken into account. MATLAB environment is utilized to generate these synthetic data traces. After that, traces are sent to FPGA via serial interface to load Upper Input FIFO and Lower Input FIFO as stated in section. 4.1.

After generating synthetic data traces, FASST is analyzed at two operation phases. Throughout the chapter. 5, these phases are:

- No rule caching phase, where *cache_hit_rate = 0%*
- Partially rule caching phase, where *cache_hit_rate = x%*, $0 \leq x \leq 100$

At each phase, we evaluate average packet latencies and throughput values for static (functional) and dynamic (hardware) tests.

Furthermore, this thesis provides the results of power consumption at run time. Since SRAM-based TCAMs are power consuming architectures, as cache hit rate increases, power rates are expected to increase in FASST. In order to verify the power consumption of FASST, real-time results are compared with the results of Altera Early Power Estimator (EPE) Tool. This tool (EPE) gives the ability to estimate power consumption of designs without actual implementation on real hardware with a good accuracy. Moreover, power values are provided at different clock values in order to indicate the scalability of FASST. Power consumption, latency and throughput analysis of FASST with previous work are also given in this chapter.

After all functional and hardware tests are carried out, FPGA resource utilization is given for different parameters. For this purpose, resource utilization of FASST for $|\mathcal{R}|$=512 and $|\mathcal{R}|$=128 is provided in order to observe the scalability of FASST with respect to rule set size.

5.1 Synthetic SDN Flow Table with 512 Rules

A synthetic flow table is generated using MATLAB. The rules in this flow table must satisfy certain characteristics to prove the functionality of FASST, which can be listed below:

- ***Req.1*:** Some rules in BVM must have wildcard match fields including ingress (31:0), metadata (63:0), source mac (47:0), destination mac (47:0), eth_type (15:0), vlan_id (11:4), vlan_po (2:0), mpls_label (19:0), mpls_po (2:0), protocol (7:0), tos (5:0), source port (15:0) and destination port (15:0). This is required to verify the rule insertion and rule query processes for wildcard match entries in BVM and TCAM.

- ***Req.2*:** Some rules in BVM must have prefix lengths which are different from 0 for Source IP(31:0) and Destination IP(31:0). This is required to verify the rule insertion and rule query processes for IP fields having different prefix lengths.

- ***Req.3*:** Some rules must have containment relationships regarding Source IP and Destination IP. The definition of containment in FASST is given such that two rules have containment relationship if all header fields, excluding Source IP or Destination IP, are same. However, by the arrangement of prefix lengths of IP fields, a packet, P matches both of the rules. This is required to verify the dependency graph with respect to IP fields.

- ***Req.4*:** Some rules must have partial overlaps between each other. Two rules have partial overlaps if some of the header fields are different between these rules; but, by a proper arrangement of mask bits, a packet P matches both of the rules. This is required to verify the dependency graph with respect to header fields excluding IP fields.

- ***Req.5*:** All 512 rules must have different <Rule ID> and <Rule Priority> values within the range *(0,511)* in order to verify overall design.

- ***Req.6*:** Some rules must have exactly same header fields, where all fields are unmasked and prefix lengths are 0. This is required to verify priority based query in FASST.

Based on rule characteristics stated above, an example rule set is generated for 512 rules. Consider that, rules are named as R_i, $0 \leq i \leq 511$. R_i is the rule located at address i in BVM. Moreover, Rule IDs are given to rules such that; if $RuleID_i$ is the rule ID of R_i, then $RuleID_i$ =i. Regarding priorities, if $Priority_i$ is the priority of R_i, then $Priority_i = i + 256$ for $0 \leq i \leq 255$, and $Priority_i = i - 256$ for $256 \leq i \leq 511$. Moreover, R_i has higher priority than R_j if $i < j$.

The configuration of Rule IDs and Rule Priorities are illustrated in Table. 5.1. The reason of giving distinct values of Rule IDs and Rule Priorities is to satisfy *Req.5*.

Most of the rules have distinct header patterns in this synthetic flow table. However, in order to satisfy *Req.6* stated above, exactly same header bits are assigned to some

Table 5.1: Rule IDs and Rule Priorities for synthetic flow table with 512 rules

Rule Number	**Rule ID**	**Rule Priority**
R_0	0	256
R_1	1	257
R_2	2	258
.	.	.
R_{255}	255	511
R_{256}	256	0
R_{257}	257	1
.	.	.
R_{511}	511	255

rules. All fields for these rules are unmasked and prefix lengths of IP fields are set to 0. Therefore, according to our synthetic flow table, R_i and R_{i-1} have exactly same header bits for $i = 16k+15, 0 \leq k \leq 31$. For example; pairs of (R_{14}, R_{15}), (R_{30}, R_{31}), ..., (R_{510}, R_{511}) have exactly same header contents. Hence, according to priorities in Table. 5.1, it can be deduced that R_{15} is dependent on R_{14}, R_{31} is dependent on R_{30} and similarly R_{511} is dependent on R_{510}.

Furthermore, in order to support *Req.1*, and *Req.4* together, the header and mask bits for 4 rules among 512 rules are specially assigned. Table. 5.2 demonstrates the header bits for these 4 rules, which are R_{24}, R_{338}, R_{467} and R_{490}. As seen in Table. 5.2, R_{24}, R_{338}, R_{467} and R_{490} are dependent on each other. If we consider priorities, a packet P that matches all 4 rules will result in a final match result of R_{338}, where Rule Priority of R_{338} is the lowest value among them, which is 82.

Lastly, in order to satisfy *Req.2*, and *Req.3* together, IP fields of 3 rules are arranged such that a containment relationship can be observed among these rules. These rules are R_{67}, R_{98}, and R_{263}. Table. 5.3 shows the Source IP and Destination IP field contents of these 3 rules, together with prefix lengths. Note that, other fields of these 3 rules have the same header content. Proper arrangement of prefix lengths for both IP fields enables 3 rules to have dependency relation between each other. However, since R_{263} has highest priority, a packet P that matches all 3 rules will result in a final

Table 5.2: Rules with partial overlaps using mask bits

Header Field	R_{24}	R_{338}	R_{467}	R_{490}
Ingress	*	01:23:45:67	01:23:45:67	01:23:45:67
Metadata	01:23:45:67: 89:AB:CD:EF	01:23:45:67: 89:AB:CD:EF	01:23:45:67: 89:AB:CD:EF	01:23:45:67: 89:AB:CD:EF
Source MAC	01:23:45:67:89:AB	01:23:45:67:89:AB	01:23:45:67:89:AB	01:23:45:67:89:AB
Dest MAC	01:23:45:67:89:AB	01:23:45:67:89:AB	*	01:23:45:67:89:AB
Eth Type	F1:23	F1:23	F1:23	F1:23
Vlan ID	0:12	0:12	0:12	*
Vlan Po	2	2	2	2
MPLS Lbl	0:12:34	0:12:34	0:12:34	0:12:34
MPLS Po	2	2	2	2
Src IP	01:23:45:67	01:23:45:67	01:23:45:67	01:23:45:67
Dst IP	01:23:45:67	01:23:45:67	01:23:45:67	01:23:45:67
Protocol	01	01	01	01
ToS	09	09	09	09
Src Port	01:23	*	01:23	01:23
Dst Port	01:23	*	01:23	01:23
Src IP Prefix	0	0	0	0
Dst IP Prefix	0	0	0	0

match result of R_{263}.

Table 5.3: Rules with containment relation using prefix lengths in IP fields

Header Field	R_{67}	R_{98}	R_{263}
Src IP	0B:5B:05:11	05:5B:05:1E	0B:5B:05:11
Dst IP	0D:5B:05:10	0D:5B:05:10	0D:5B:0A:EE
Src IP Prefix	0	4	0
Dst IP Prefix	0	0	12

The remaining rule entries have distinct header fields, except for rules in Table. 5.2, Table. 5.3 and rules with same header fields, $R_i : R_{i-1}, i = 16k + 15, 0 \leq k \leq 31$. Hence, there is no dependency relation among them.

This example flow table is generated by MATLAB source code, and written to Upper Input FIFO via RS232 serial interface. After that Packet Parser (PP) reads all 512 rules from FIFO and writes to BVM in order.

5.2 Synthetic Traffic Trace using Flow Table with 512 Rules

After defining a flow table of 512 rules for BVM, it is required to generate a sample traffic trace in order to verify rule query phase, temporal locality detection and cache hit rates. As described in section. 4.2, network characteristics are stored in Lower Input FIFO with a specific packet format in FASST. Hence, a series of packets with 15-tuple SDN headers is generated and sent to Lower Input FIFO. While generating these packets, we assume a line utilization of 100%, which makes BVM 2D pipeline always full. This is because, maximum throughput is achieved when line utilization is 100% at the targeted clock rate. Moreover, in order to verify the timing constraints at high clock rates, 100% line utilization is required.

Moreover, packet headers for this traffic trace are generated by considering flow table in BVM. In other words, matching packet headers are same with the matched rule headers located in BVM. This approach is carried in order to make the verification process in rule query phase in a more controlled and efficient way. Table. 5.4 illustrates generated series of packets, together with the order and the packet numbers. For example, 30 packets having the same header fields with R_1 are sent to FASST for rule query phase. Right after this, 100 packets having the same headers with R_5 are sent. Therefore, we expect 30 final match results of R_1, and 100 final match results of R_5 from BVM in order.

In order to verify the priority-based query, we also send packets having the same contents with at least two rules in BVM. For example, for *Packet Sequence =13* and *Packet Sequence =15*, packets having same header contents with R_{127} are sent. Since R_{127} has a dependency on R_{126} as stated in section. 5.1, then the expected final match

Table 5.4: Sample traffic trace with 100% line utilization

Sequence	Packet Number	Rule of Same Content	Expected BV Result(s)	Expected Final BV result
1	30	R_1	R_1	R_1
2	100	R_5	R_5	R_5
3	20	R_1	R_1	R_1
4	200	R_5	R_5	R_5
5	90	R_9	R_9	R_9
6	50	R_{22}	R_{22}	R_{22}
7	210	R_9	R_9	R_9
8	10	R_{65}	R_{65}	R_{65}
9	150	R_{24}	R_{24}, R_{338}, R_{467}, R_{490}	R_{338}
10	40	R_{65}	R_{65}	R_{65}
11	150	R_{24}	R_{24}, R_{338}, R_{467}, R_{490}	R_{338}
12	300	R_{98}	R_{67} R_{98}, R_{263}	R_{263}
13	30	R_{127}	R_{126}, R_{127}	R_{126}
14	50	R_{200}	R_{200}	R_{200}
15	20	R_{127}	R_{126}, R_{127}	R_{126}
16	50	R_{63}	R_{62}, R_{63}	R_{62}
17	150	R_{135}	R_{135}	R_{135}
18	100	R_{62}	R_{62}, R_{63}	R_{62}
19	30	R_{244}	R_{244}	R_{244}
20	10	R_{245}	R_{245}	R_{245}
21	10	R_{304}	R_{304}	R_{304}
22	10	R_{339}	R_{339}	R_{339}
23	10	R_{452}	R_{452}	R_{452}
24	10	R_{502}	R_{502}	R_{502}
25	20	R_{244}	R_{244}	R_{244}
26	150	R_{276}	R_{276}	R_{276}
	2000			

result must be for R_{126}.

Moreover, in order to verify overlap conditions for the rules, some packets have same

header fields with particular rules as in the case of *Packet Sequence =9* and *Packet Sequence =11*. Note that, R_{24}, R_{338}, R_{467}, and R_{490} have dependencies among each other with overlap condition. Hence, when 150 packets are sent with same header content with R_{24}, the final match result must be R_{338} due to lowest priority. The same procedure is applied for containment relations in IP fields for *Packet Sequence =12*. When a whole sequence is completed in Table. 5.4, packets are continued to be sent by PP starting from beginning where *Packet Sequence =1*. As a result, 100% line utilization is achieved.

5.3 Design Parameters Used in Performance Evaluation

While performing overall verification of FASST, some design parameters should be set at synthesis level. First of all, for 512 rules, overall design in FPGA is run at 200 MHz clock rate. This clock rate is the maximum value for FASST hardware architecture for the target FPGA after a number of iterations. Moreover, time window size, W_s, is set to 20000. In other words, *time_window_counter* in *windowed_rule_block* stated in section. 4.6.1 counts up to 20000 at the end of each time window. After that it is initialized to 0. The selection of W_s is strictly dependent on the sample network traffic. In order to achieve increasing cache hit rates at run time, W_s should be long enough to catch the localities for the recent past. Our sample network traffic consists of about 2000 match counts for different SDN packets at each round. After 2000 match counts, same series of packets arrives to FASST for another round of look up process due to 100% line utilization. Therefore; W_s = 20000, which is ten times of match counts in each round, is long enough to catch different cache hit rates inside each time window.

In order to identify a rule as a popular rule, threshold level Thr is set as 100. Hence, if the value of match count of a particular rule is higher than or equal to 100 at the sampling time, then this rule is identified as popular rule and corresponding BVM address is written to RAM-1 by *Locality Detector*. The selection of threshold level Thr can be changed at pre-synthesis level before FPGA fitter operation, and it exactly depends on the incoming network traffic. In other words, the value of Thr should be low enough to determine elephant or large volume flows inside the run-time network

traffic. However, if it is too low, then TCAM Cache will be filled with unnecessary rule entries with short-term durations. Similarly, if Thr is set to a very high value, no popular rules will be detected inside a time shifting window at the sampling points. Our sample network traffic consists of SDN packets whose durations are between 10-300 match counts, and the sequence number in each round is 26 as seen in Table. 5.4. Therefore, the value of Thr is determined as 100 match counts using a discrete uniform distribution for each round. In other words; when a uniform distribution is considered, the average match count for each sequence will be about 12. Therefore, about a ten times higher value of the average number of match counts for Thr is an optimal value. This is because, as stated in[9], 97% of the matches in a SDN traffic are to 10% of the rules.

The sampling points in FASST are chosen to be the points that *time_window_counter* value is integer multiples of $1900k$, $k > 0$. For example, *Locality Detect* takes first locality samples when *time_window_counter* reaches to 1900, and second locality samples when *time_window_counter* reaches to 3800. The selection is of sampling points is made by considering the size of time window, which is 20000 match counts. In order to observe increasing cache hit rates in each time window, the sampling points are set to lower value than time window size.

As observed from Table. 5.4, one round for traffic trace consists of 2000 match counts, which is very close to sampling point time. Hence, at first round where first sampling occurs in time window, some of the rules are expected to be identified as popular rules.

5.4 Hardware Tests of Overall Design for 512 Rules

The results of FASST hardware tests are carried out by using *Altera Signal Tap II Embedded Logic Analyzer* tool, which helps debugging of FASST by probing the state of internal signals in FPGA at run-time.

Since *Altera Signal Tap II Embedded Logic Analyzer* consumes limited embedded memories in FPGA, it is not possible to probe all signals at all time instants. For this purpose, hardware test results are given in 2 phases: no-caching phase where

Figure 5.1: Match results with no-caching phase

cache_hit_rate = 0% and partially rule caching phase where *cache_hit_rate = x%*, $0 \leq x \leq 100$.

In Fig. 5.1, matched Rule IDs of BVM for the sample traffic trace are illustrated. These signals are probed in hardware before the first sampling point in time window. Therefore, *cache_hit_rate* is equal to 0% in this case and no popular rules are observed yet. Furthermore, *tcam_output_valid* is 0, which is the output valid signal of FASST TCAM.

The signals *rule_id_out(8:0)* and *match_valid_out* indicate matched Rule IDs provided by BVM in rule query phase. These match results are observed to be completely compatible with the sample traffic trace given in Table. 5.4. Moreover, there is no blanking intervals between match results due to 100% line utilization. As seen in Fig. 5.1, output match results with *rule_id_out =276* are followed by results with *rule_id_out =1* immediately. The packet numbers are also equal to the input packet numbers, For example, match count for *rule_id_out =5* is observed as 100.

The match result with *rule_id_out =338* verifies the functionality of overlap conditions. Note that, input packet content for this match is same with the content of R_{24}. However, since R_{24} is dependent on R_{338}, and R_{338} has highest priority, final match result becomes R_{338} with *rule_id_out =338*.

Since TCAM does not have any rules to perform lookup at this time, average latencies

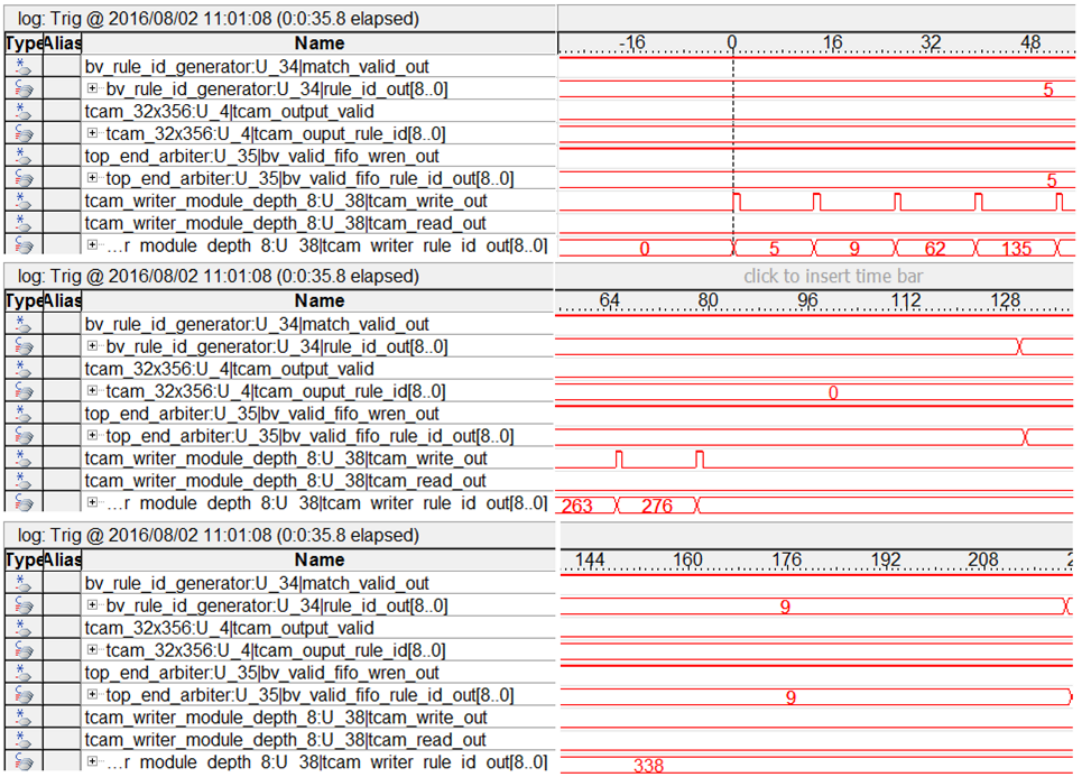

Figure 5.2: First round at locality detection for 7 rules

of the packets are equal to deterministic end-to-end latency of BVM, which is 80 clock cycles. Since our clock frequency is 200 MHz, average latency value for all packets is 400 ns.

Fig. 5.2 shows the data samples at the end of first locality detection round. At the end of first sampling, where total match counts are 1900, match counts for rules R_5, R_9, R_{62}, R_{135}, R_{263}, R_{276}, and R_{338} are 300, 300, 150, 150, 300, 150 and 300, respectively. These values are higher than or equal to determined Thr value at synthesis level. Hence, TCAM Cache Interface writes these popular rules to TCAM using write enable signal *tcam_write_out*. Moreover, Rule IDs of popular rules are written to TCAM using *tcam_writer_rule_id_out(8:0)*. Due to limited on chip resources in FPGA; only write enable and Rule ID signals are sampled. As a result, a total of 7 rules are written to TCAM at this time. Furthermore, as observed from Fig. 5.2, BVM continues to generate match results for the sample traffic trace. Therefore, writing rules to TCAM is completely independent of rule query phase in BVM.

Note that, since the matched rules have highest priorities among their dependent rules,

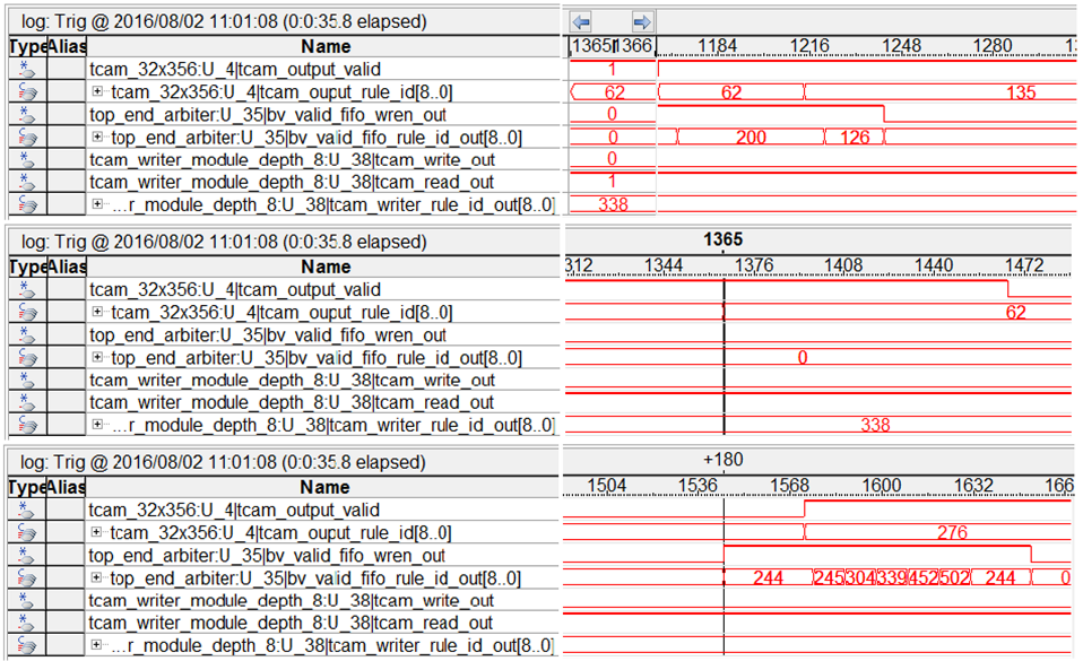

Figure 5.3: TCAM and BVM parallel lookup after first round of locality detection

no other dependent rules are written to TCAM. In other words, R_{24} is not written to cache although R_{338} is written. This is because BVM already generates match results for R_{338}, not for R_{24}.

The situation after writing 7 rules to TCAM is depicted in Fig. 5.3. TCAM output signals, *tcam_output_valid* and *tcam_output_rule_id(8:0)* start to assert after this point. In Fig. 5.3, *bv_valid_fifo_wren_out* and *bv_valid_fifo_rule_id_out(8:0)* denote the outputs of BVM Match Controller in MA explained in section. 3.6. As observed, this controller does not generate Rule IDs which are already provided from TCAM. For example, there is no match result for *bv_valid_fifo_rule_id_out(8:0) =135*, or *bv_valid_fifo_rule_id_out(8:0)= 62*. On the other hand, for the rules that are not stored in TCAM, BVM Match Controller in MA continues to generate Rule IDs in rule query phase. For example, R_{244} and R_{245} are not stored in TCAM and match results of *bv_valid_fifo_rule_id_out(8:0) =244* or *bv_valid_fifo_rule_id_out(8:0) =245* are still provided by BVM Match Controller.

Since 7 rules are written to TCAM for a faster lookup process, the average latencies for the packets that match 7 rules will be far less than the packets which do not match. For example, the clock cycle duration between the first match result of R_{62}

in *Sequence =18* and the first match result of R_{244} is about 180 clock cycles. Note that match result for R_{62} is generated by TCAM with *tcam_output_rule_id(8:0)*, and match result for R_{244} is generated by 2D BVM with *bv_valid_fifo_rule_id_out(8:0) =244*. If no rules are cached in TCAM, then the first match result for R_{62} must arise 100 clock cycles before the first match result of R_{244} when BVM pipeline is full. However, first match result of R_{62} is observed about 80 clock cycles before that point. This time difference is mainly results from $t_{fast} = 3$ cycles lookup of TCAM. TCAM provides Rule ID of R_{62} about 80 clock cycles before BVM.

In Table. 5.4, there are 18 packets which have exactly different header contents. Since 7 of these 18 rules are written to TCAM, *cache_hit_rate* can be calculated with equation. 5.1. Therefore, for first round of locality detection, *cache_hit_rate* is found as *1650/2000* $\approx 82.5\%$ for parallel processing of BVM and TCAM. In order to calculate the overall average latency for the traffic trace, equation. 5.2 can be used.

$$Cache\,Hit\,Rate = \frac{Total\,BV\,match\,results\,by\,TCAMCache}{Total\,BV\,match\,results\,by\,BVM} \tag{5.1}$$

$$Overall\,Average\,Latency = \frac{\sum_{i=1}^{n} Packet\_Count_{\mathrm{i}} \times Average\_Latency_{\mathrm{i}}}{TotalPacketCount} \tag{5.2}$$

$AverageLatency_i$ in Equation. 5.2 is strictly dependent on cache hit rate. For the first round where *cache_hit_rate* $\approx 82.5\%$, overall average latency is found by ≈ 16 clock cycles, which is 80 ns. If all packets are classified using only BVM, then overall average latency will be 80 clock cycles, which is 400 ns. Therefore, for *cache_hit_rate* $\approx 82.5\%$, FASST achieves a reduction of 80.00% in overall average latency in ns for this traffic trace.

At this point, a relation between *cache_hit_rate* and *reduction in overall average latency* can be deduced. As observed above, two values, 82.5% and 80.00%, are very close to each other. This relation mainly caused from 3 clock cycles lookup of TCAM. This is because TCAM lookup provides a constant 3 in equations, which has small effect on overall latency equation. For example, assume that, there are *A* packets, whose match results are produced by BVM Controller in MA, and there are *B*

packets, whose match results are produced by TCAM. Hence, all of *A* packets observe an average latency of 80 clock cycles due to BVM, and all of *B* packets observe an average latency of 3 clock cycles due to TCAM. For this case, *cache_hit_rate* and overall average latency in terms of clock cycles will be as in Equation. 5.3, and Equation. 5.4, respectively. Hence, the reduction in overall average latency will as in Equation. 5.5 in terms of clock cycles. The results in equation. 5.3 and equation. 5.5 are almost same if we ignore 77 and 80. Therefore, as the *cache_hit_rate* increases, the overall average latency decreases at same percentage.

$$Cache\,Hit\,Rate = \frac{B}{A+B} \tag{5.3}$$

$$Overall\,Average\,Latency = \frac{80A+3B}{A+B}\,in\,clock\,cycles \tag{5.4}$$

$$Reduction\,in\,Latency = \frac{80 - \frac{80A+3B}{A+B}}{80} = \frac{77B}{80(A+B)}\,in\,clock\,cycles \tag{5.5}$$

After giving data samples at the end of first round where *time_window_counter* is 1900, Fig. 5.4 shows the rules written to TCAM at second round of locality detection. At this second round, 6 new rules are added to TCAM in addition to previous 7 rules. This is due to the fact that, when *time_window_counter* reaches to 3800, the sequence of packets in Table. 5.4 is almost sent twice. Hence, match counts for each rule increase. In other words, total match counts for R_1, R_5, R_9, R_{22}, R_{62}, R_{65}, R_{126}, R_{135}, R_{200}, R_{244}, R_{263}, and R_{338} will be 100, 600, 600, 100, 300, 100, 100, 300, 100, 100, 600, 300, and 600, respectively. As observed, all total match counts are higher than or equal to Thr. Moreover, as seen in Fig. 5.4, the old rules are rewritten to TCAM at second round, because TCAM content is cleared by TCAM Cache Interface at the beginning of each locality detection. Note that, TCAM match output valid signal, which is *tcam_output_valid*, is 0, which indicates that TCAM does not perform classification during rule insertion. Actually, just before rule insertion phase, TCAM is empty.

The situation after writing these 13 rules to TCAM is depicted in Fig. 5.5. As observed, match results for 13 rules out of 18 rules are generated by TCAM. BVM

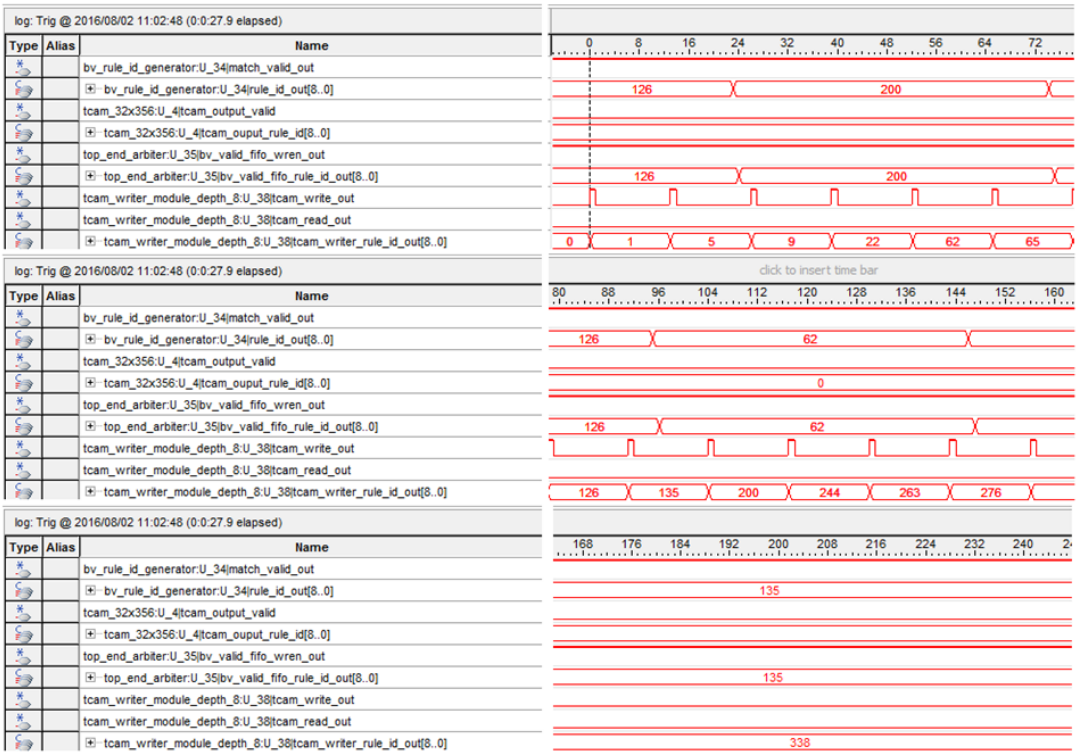

Figure 5.4: Second round at locality detection for 13 rules

Match Controller in MA only generates the match results for the rules that are not stored in TCAM, which areR_{245}, R_{304}, R_{339}, R_{452}, and R_{502}. Due to fast clock lookup process in TCAM, match result of R_{276} is produced before the match result of R_{245}

Using equation. 5.1, *cache_hit_rate* for the second round of temporal locality is found as *1950/2000 =97.5%*. Similarly, using equation. 5.2, for this *cache_hit_rate*, overall average latency is found as $\approx$5 clock cycles, which is 25 ns. Therefore, reduction in overall average latency is $\approx$93.75%.

For synthetic SDN flow table in section 5.1 and sample traffic trace in section. 5.2, Fig. 5.6 demonstrates the relation between *cache_hit_rate* and *overall_average_latency* in FASST. Moreover, during query operation, achieved throughput is always 200 Mega Packet Per Second (MPPS) due to the 200 MHz clock rate in FASST.

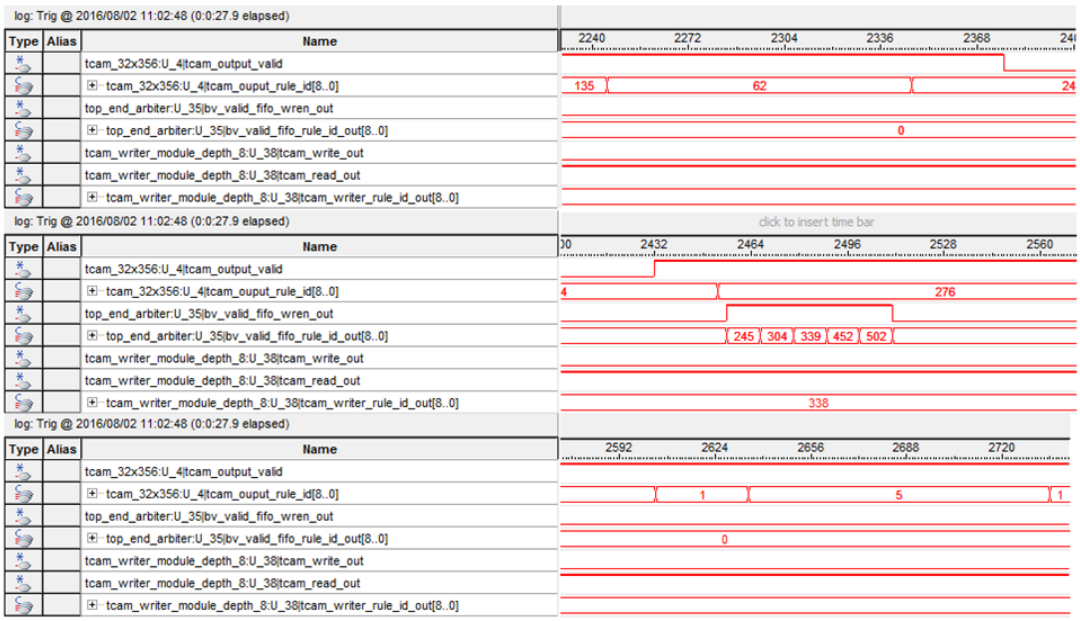

Figure 5.5: TCAM and BVM parallel lookup after second round of locality detection

5.5 Monitoring Consumed Power for 512 Rules

The hardware platform, where FASST architecture is implemented, has Octal Digital Power Supply Monitor Devices, named LTC2978 [54], in order to monitor voltages and currents of the components on board. Each LTC2978 has an I2C communication interface used for configuration and data transfer. As explained in section 4.6.2, I2C custom interfaces in processor SoC are utilized for this purpose. Since configuration of LTC2978 devices is not in the scope of this thesis, it is not given.

While analyzing power consumption of FASST, the only required information is the current drawn by the core voltage of FPGA. FPGAs have many voltage rails such as core voltages, transceiver voltages and I/O voltages. However, since FASST is a purely on-chip architecture on FPGA without using any transceivers and I/Os, core voltage, which is 0.85 V, is the primary focus in this power consumption. When we read the voltage value across the current sense resistor connected to the core voltage power rail of FPGA, we obtain the current value for this power rail. After that, multiplying the core voltage and current gives us the power consumption of FPGA in *Watts*.

LTC2978 provides the voltage values over the current sense resistor as L11 data for-

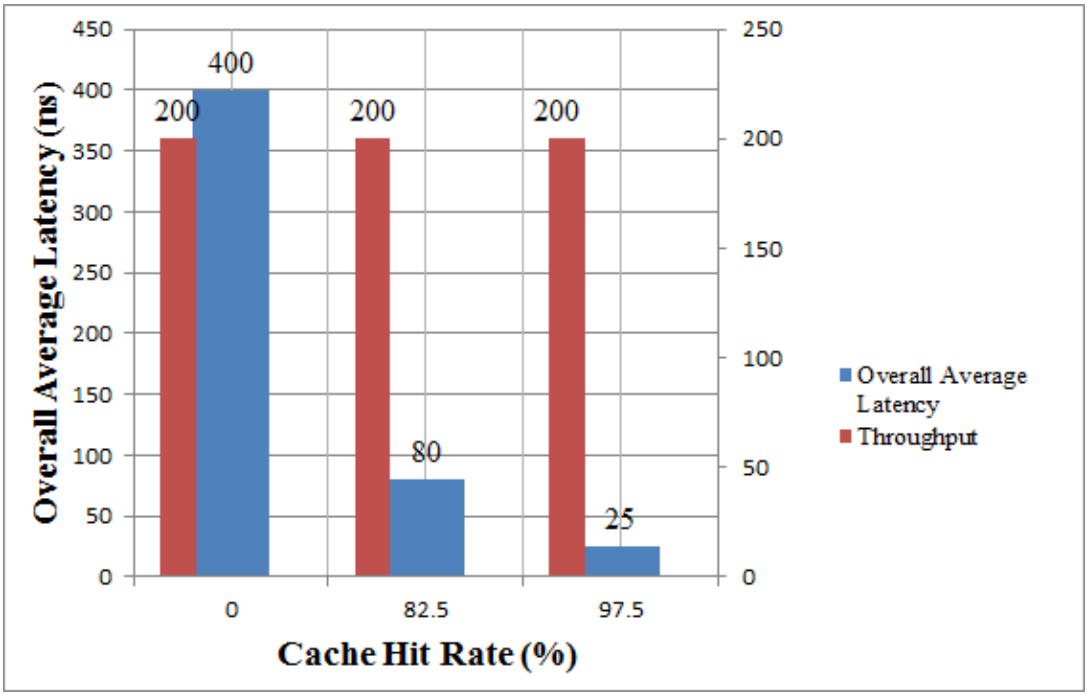

Figure 5.6: Cache Hit Rate(%) vs. Average Latency and Throughput

mat, which is a format defined by *Linear Technologies*. This format consists of a total of 16-bits, which encapsulates 5-bit signed exponent and 11-bit signed mantissa [54]. L11 format is mainly utilized in order to provide current values in high precision and high dynamic range. Therefore, Nios II IP soft processor core reads current value in L11 format and converts it to floating values with the help of high level programming language.

Current values are read at three different phases. One of them is the idle condition where there is no rule query. For this case, BVM and TCAM has no functionality for classification of packets. Since majority of power is consumed by these modules, the current drawn by FPGA core is quite small at this idle condition, which is *1.29 A*. Hence, total power consumed for this phase equals to $0.85V \times 1.29A = 1.1$ *Watts*.

Second phase at monitoring power consumption occurs when *cache_hit_rate* equals to 0%. At this phase, only BVM performs lookup process for the packets at 200 MHz. Since line utilization is 100% in our traffic trace, two dimensional pipeline in BVM is always full, and all *Stride BVM* units do read and write transactions using data associative memories. Hence, due to 100% utilization of on-chip ram blocks in BVM at 200 Mhz, the current value read from LTC2978 is about *6.18 A*, which gives

the power value of *5.253 Watts*.

Third phase of power monitoring is carried out when both BVM and TCAM perform lookup in parallel. Current values at third phase are sampled after the point of second locality detection in Fig. 5.5. At this time, there are 13 rules are stored in TCAM. The current value for this case is calculated as *6.20 A*, which is very close to the value in second phase. Therefore total power consumed by FPGA core with parallel operation BVM-TCAM at 200 MHz is *5.27 Watts*. Obtaining close values between second phase and third phase is mainly caused from the fact that TCAM size is relatively small (32 rules) compared to the size of BVM (512 rules). Moreover, since utilization percentage of on-chip RAMs inside *Stride BVM* is 100%, BVM dominates the power consumption. Table. 5.5 summaries all power values for three phases.

Table 5.5: Power consumption of FASST for 512 rules at different phases

Phase	**Description**	**Current(A)**	**Power(W)**
1	Idle Phase: BVM and TCAM does not perform classification	1.29	1.1
2	Only BVM performs lookup for packets. TCAM Cache is empty	6.18	5.253
3	Both BVM and TCAM Cache perform lookup. There are 13 rules stored in TCAM Cache.	6.20	5.27

5.6 Calculating Power Consumption Using Early Power Estimator (EPE)

Altera Early Power Estimator (EPE) tool provides estimation of power consumption in complete FPGA designs without actual implementation on real hardware. Therefore, a comparison between the power values presented in section. 5.5, and the output of this tool is made in order to verify the power consumption rates for a certain accuracy.

To determine the power consumption, Altera Early Power Estimator Tool uses the targeted FPGA family information, total resource utilization, toggle rates, clock fre-

quencies of logic units and on-chip memories *M20K*, and percentages of write and read modes for embedded memories with respect to each other [55]. For 512 rules and 200 MHz clock rate, this information is entered to EPE tool.

Total power consumption is observed to be 5.502 W as seen in Table. 5.6, which is very close to the power value, 5.27 W, presented in section. 5.5. Note that, this power consumption is for parallel processing of BVM and TCAM at 200 MHz after the second round of locality detection. Moreover, power consumption at idle state, 1.1 W, is similar to P_{Static}, which is 1.156 W. Since FASST does not utilize any transceiver (XCVR,PCS), hard IP (HIP), Digital Signal Processor (DSP) block, or I/O in FPGA, the associated power values for these components are 0. Logic power, which is 2.039 W and RAM power, which is 2.306 W, are the sources of total consumed power.

Table 5.6: Power consumption using EPE for parallel processing of BVM and TCAM

INPUT PARAMETERS	
Family	Stratix V
Device	5SGXEA7N
Package	F40
Temperature Grade	Commercial -2L/-3/-4 (0.85V)
Power Characteristics	Typical
Power Model Status	FINAL
THERMAL POWER (W)	
Logic	2.039
RAM	2.306
DSP	0.000
I/O	0.000
XCVR	0.000
PCS and HIP	0.000
P_{static}	1.156
TOTAL FPGA	5.502

Details of parameters used to calculate logic and RAM power consumptions are pre-

sented in Table. 5.7. For logic power, the number of combinational look up table (LUT) and the number of flip flops (FF) are entered. Moreover, clock frequency is set as 200 MHz. Average fan out value is set as 1 due to the fact that BVM consumes most of the logic inside FASST. Since *Stride BVM* and *POEnc* units are connected in a two dimensional pipelined fashion in BVM , each block only communicates with single block. This results in a fan out value of 1 for all functional blocks. Toggle rate is assumed as 12.5%, which is the typical value for most logic inside FPGA as presented in [55]. This toggle rate can be considered as the toggle percentage of 16-bit counter.

For RAM power, total number of RAM blocks, which are actually the number of on-chip embedded blocks called *M20K* to generate on chip RAMs such as RAM-1, RAM-2 and RAM-3, are given as input parameter to EPE. Moreover, RAM Mode is selected as true dual port memory, which means that memory has two independent read and write interfaces that provide simultaneous transfers. The selection of dual port RAM Mode is due to the fact that *Stride BVM RAM* blocks in section. 4.4.1 and *Stride TCAM RAM* blocks in section. 4.5.1 have two ports, namely *PORT_A* and *PORT_B*. In Table. 5.7 *PORT_A* denotes write interface and *PORT_B* denotes read interface in EPE, which is a reverse order given in Fig. 4.7 and Fig. 4.11. For *PORT_A*, percentage of write mode versus read mode is set as 1%, which is a very small value. This is because, FASST is in rule insertion phase only at the beginning of the operation. Overall architecture mostly performs rule query. Moreover, for *PORT B*, percentages of read mode, which are *R/W%* and *Enable%*, is set as 100%. This is because line utilization in rule query phase is 100% for the sample traffic trace. Lastly, clock frequency and toggle rates for RAM blocks are again set as 200 MHz, and 12.5%, respectively.

5.7 Static Functional Simulation of Overall Design for 512 Rules

Simulation results for the sample traffic trace given in Table. 5.4 are exactly compatible with the hardware results presented in section. 5.4. These results are illustrated in Fig. 5.7. In addition to Rule IDs of the written rules, output counter values are also

Table 5.7: Details of Logic and RAM Power in FASST

LOGIC POWER						
# Combinational LUT	#FFs	Clock Freq. (MHZ)	Toggle (%)	Avg. Fan Out	Est. LUT Utilization (%)	**Total Power (W)**
331474	349578	200	12.5%	1	70.6%	**2.039**
RAM POWER						
# of RAM Blocks	RAM Mode	Clock Freq. (MHZ)	Toggle (%)	Port A Write (%)	Port B Enable and R/W (&)	**Total Power (W)**
1740	True Dual Port	200	12.5%	1%	100%	**2.306**

provided in simulation result. As observed, for the second round of locality detection, output counter values for Rule IDs of 245, 304, 339, 452 and 502 are below threshold level Thr. Hence rules with these Rule IDs are not stored in TCAM at second round of locality detection.

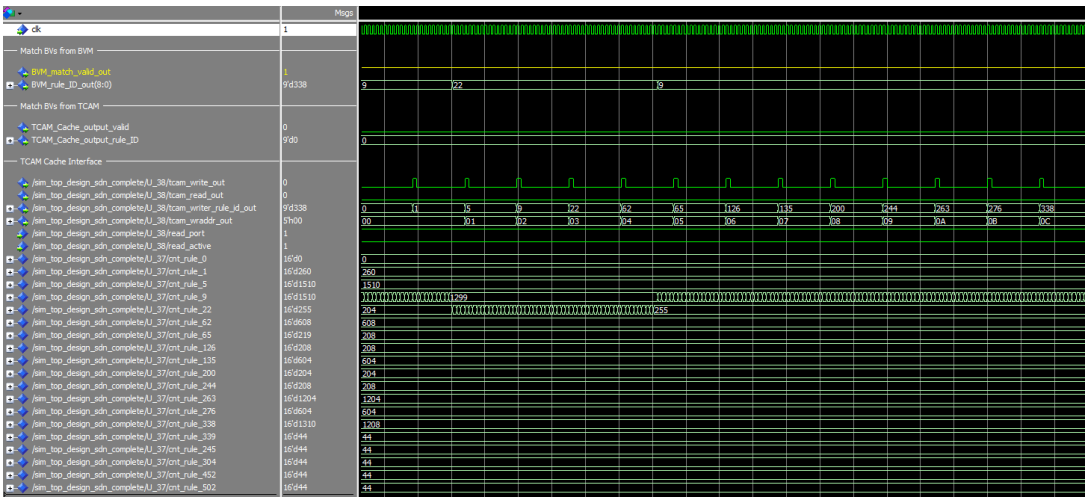

Figure 5.7: Functional simulation of FASST during second round on locality detection

The functional simulation waveform after writing popular rules to TCAM is presented in Fig. 5.8. This is the situation where BVM and TCAM perform lookup in parallel, *cache_hit_rate* equals to $\approx 97.5\%$ and reduction in overall average latency is $\approx 96.25\%$. As seen in Fig. 5.8, read enable signal, *tcam_read_out*, toggles peri-

odically in rule query phase. The period of this toggling depends on the sampling window size, $1900k$, in FASST. This is due to the fact, FASST TCAM is updated dynamically. In other words, whenever a sampling occurs inside the sliding time window, new popular rules are detected by *Locality Detect* unit and new dependent rules are computed by Nios II IP soft processor core. Therefore, at the beginning of each new locality round, TCAM Cache Interface terminates the rule query phase by deasserting *tcam_read_out* to LOW. After that, it cleans TCAM, writes the new rules and continues query process by asserting *tcam_read_out* to HIGH again. In Fig. 5.8, matched rule IDs provided by TCAM can also be seen.

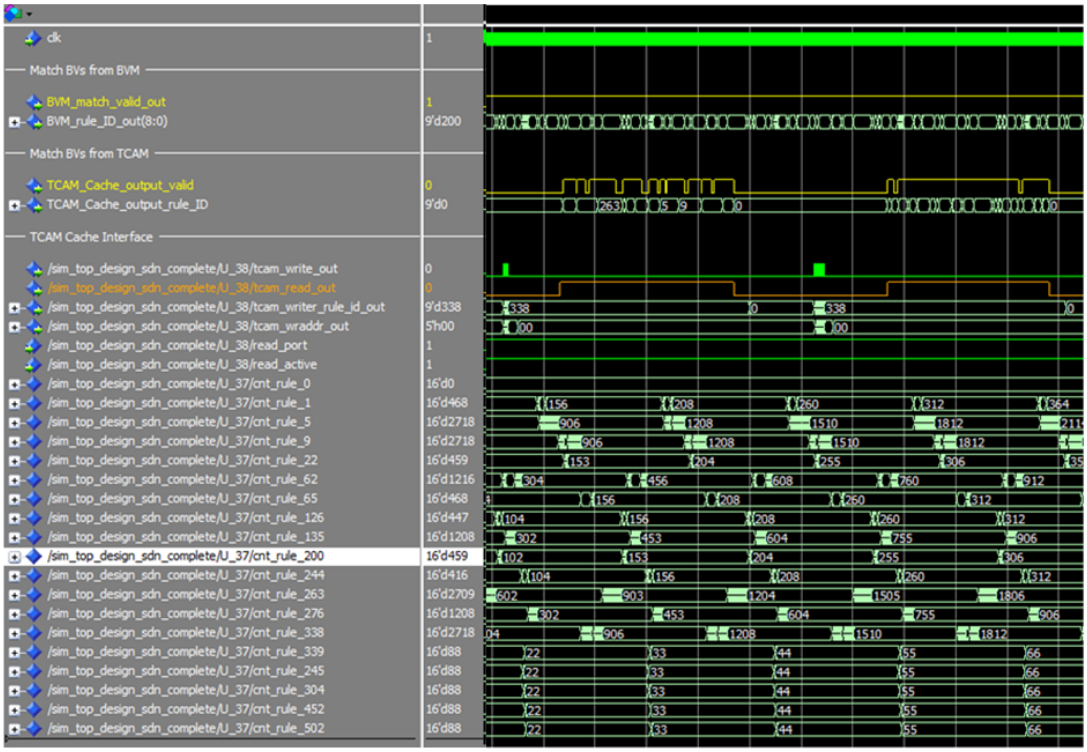

Figure 5.8: Functional simulation of parallel processing of BVM and TCAM after second round

5.8 Synthetic Flow Table for 128 Rules, Traffic Sample and Hardware Tests

After verifying the functionality of FASST for 512 15-tuple SDN rules, another sample of flow table with consists of 128 15-tuple rules is generated. Moreover, a similar traffic trace is again provided for this rule set. The purpose of using a smaller rule set size is both seeing the scalability of FASST with respect to rule set size, and verifying

the process of writing popular rules having depth of 2.

For the flow table of 512 rules and generated traffic trace, all popular rules written to TCAM have a depth of 1. In other words, for that scenario, packets arriving to FASST for classification already match the rules having the highest priorities in flow table. As a result, there is no need to take precautions to store other lower priority dependent rules. For example, packets that match R_{67}, R_{98} and R_{263} result in a final bit vector match of R_{263}, which has higher priority than R_{67} and R_{98}. Therefore, lower priority rules R_{67} and R_{98} are not stored in TCAM.

In order to verify that rules are written to TCAM with correct functionality in terms of rule depth, flow table with 128 rules and a corresponding traffic trace are generated. Note that, FASST BVM is designed as a modular architecture, which provides flexibility in terms of rule set size. In other words, by simply removing *Rule BVM* blocks from top design, a new architecture that supports a small rule set siz can be created. Moreover, since FASST also has a flexible architecture that supports any combination of header fields, only 3 header fields are used for classification in this flow table: ToS (5:0), Protocol (7:0) and IP Destination (31:0). In other words, mask bits of all remaining header fields are set to '1', which means ANY value for these fields.

Synthetic Flow Table with 128 rules is constructed as a similar way with the construction of flow table for 512 rules. Table. 5.8 shows <Rule ID><Rule Priority> pairs of the rules using the same convention. Different from the flow table of 512 rules, Rule Priorities are also given in ascending order in Table. 5.8.

Header bits of R_{13} and R_{14} are specially assigned to create a partial overlap between these two rules. Table. 5.9 shows the contents of these two rule entries stored in BVM. Regarding priority fields, R_{14} depends on R_{13}. Note that, there are other dependencies for other rules; but, the details are not given for the sake of avoiding repetitive information.

A traffic sample is generated to verify the functionality of writing popular rules with

Table 5.8: Rule IDs and Rule Priorities for synthetic flow table with 128 rules

Rule Number	**Rule ID**	**Rule Priority**
R_0	0	0
R_1	1	1
R_2	2	2
.	.	.
R_{62}	62	62
R_{63}	63	63
R_{64}	64	64
.	.	.
R_{127}	127	127

Table 5.9: Partial overlaps of two tules in Flow Table of 128 rules

Header Field	R_{13}	R_{14}
Protocol	*	05
ToS	02	*
Dst IP	00:00:00:1f	00:00:00:1f
Dst IP Prefix	0	0

depth 2. The sequence of packets consists of many different header contents. However some of the packets have Protocol value of *0x05*, Tos value of *0x01* and Dst IP value of *00:00:00:1f*. As expected, these packets must match only R_{14} in flow table. Moreover, the packet count for this specific content is arranged such that R_{14} will be identified as a popular rule in every rounds of locality detection. Therefore, R_{14} and the rules it has dependent on must be stored in TCAM Cache. At this point, according to functionality of FASST,R_{13} must also be written to TCAM. If we assume that only R_{14} is stored in TCAM, late arriving packets with Protocol value of *0x05*, ToS value of *0x02* and Dst IP value of *00:00:00:1f* result in a TCAM match with R_{14} instead of R_{13}, which is a malfunction in classification due to priority-based classification. Even though BVM presents the correct rule ID for these packets, there will be two match results for a single packet, because BVM Match Controller in Match Arbiter Unit cannot determine a late duplicate match for R_{13}, which is provided by TCAM.

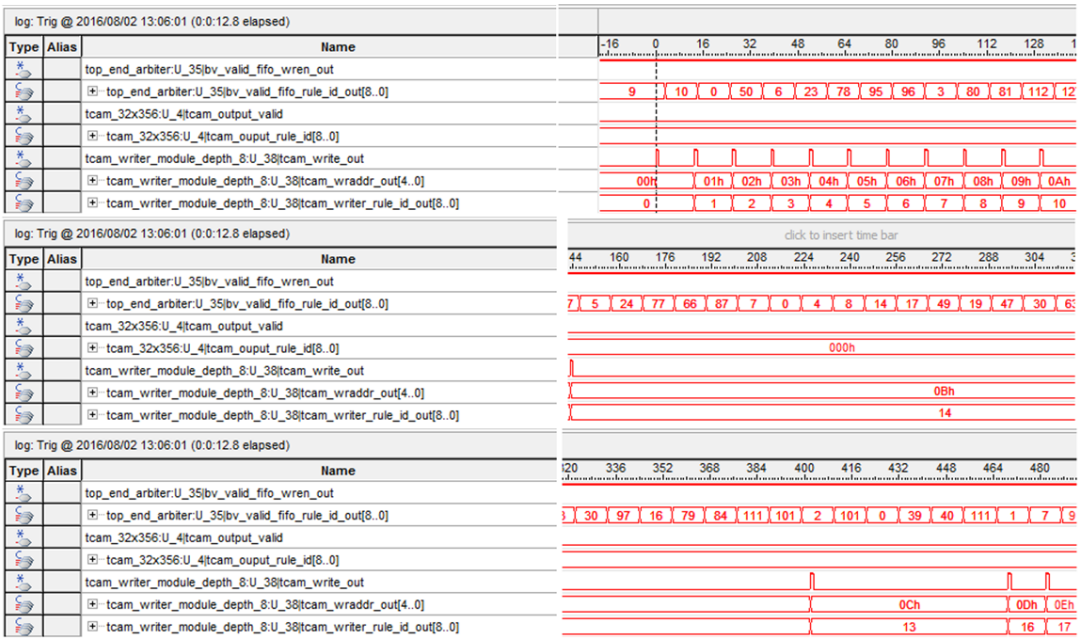

Figure 5.9: TCAM write sequence for popular rule with depth 2

Hardware test results that present writing popular rules to TCAM at some time of locality detection are shown in Fig. 5.9. TCAM Cache Interface writes the popular rules to TCAM such as R_0, R_1, and R_2. The Rule IDs of the popular rules are arranged in the sample traffic trace. As observed, R_{13} is also written TCAM Cache after R_{14} is stored, even though there are no match results for R_{13} for this sample traffic trace. This preserves the correct functionality of FASST lookup sequence. Moreover, the order of ascending Rule IDs while writing TCAM is not applicable for R_{13} and R_{14}. Furthermore, the latency of writing R_{14} to cache is quite higher than writing other popular rules. This is due to the fact that TCAM Cache Interface unit carries out additional checks for the popular rules with rule depths rather than 1.

Run time rule query in hardware is illustrated in Fig. 5.10. As seen, *tcam_output_valid* and *tcam_output_rule_id(8:0)* denote the match outputs for TCAM Cache. Similarly, *bv_valid_fifo_wren_out* denotes the match result for BVM Match Controller in Match Arbiter Unit with rule ID port shown as *bv_valid_fifo_rule_id_out(8:0)*.

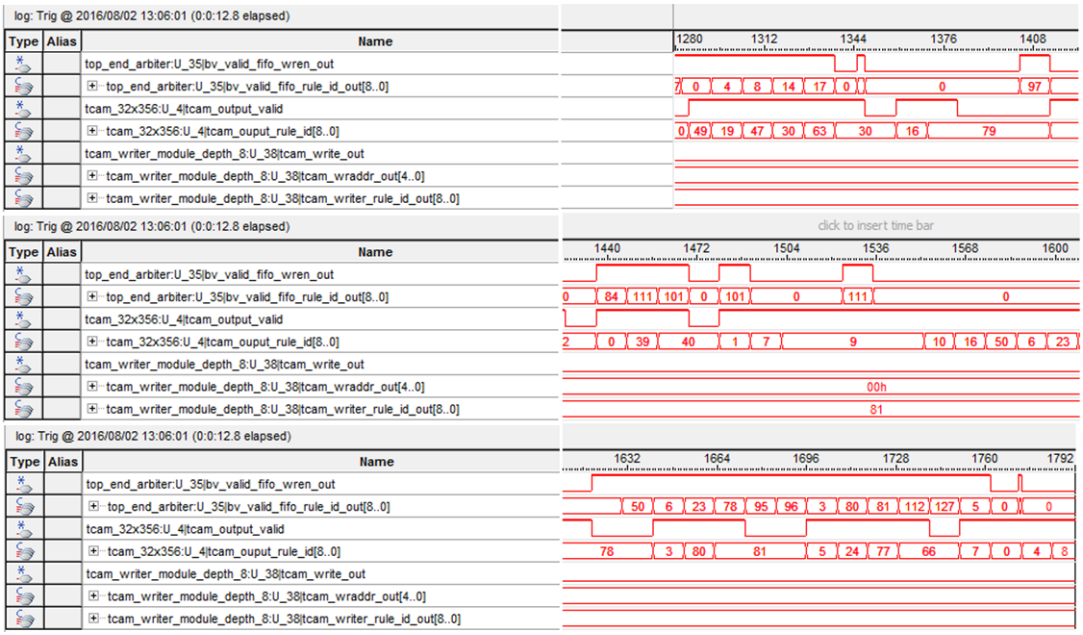

Figure 5.10: Parallel processing of BVM and TCAM for 128 Rules

5.9 Power Consumption With Respect to Clock Rate and Rule Size

Compilation time for the FPGA implementation of overall design for 512 rules is observed to be very long compared to 128 rules case. As a result, scalability analysis of power with respect to different clock rates is carried out with the design that stores 128 rules in FASST. Table. 5.10 shows the consumed power for three clock rates of 100 MHz, 200 MHz, and 250 MHz. Moreover, while monitoring power values, three phases are considered, which are idle phase, only BVM processing phase and parallel processing phase of BVM and TCAM Cache, similar to phase 1, phase 2 and phase 3 given in Table. 5.5. In phase 3, where BVM and TCAM performs parallel lookup, *Cache_hit_rate* is observed to be *71%* at maximum. However, cache hit rate has no effect on the consumed power due to the that TCAM Cache obviously performs classification even for the packets that are not stored in cache. That means that all units inside TCAM Cache operates at the clock rate for all incoming packets whether they match a rule entry inside cache or not, which is an independent process from *Cache_hit_rate*. *Cache_hit_rate* only causes a significant reduction in average latency. The reason of close values of power consumption in idle phases is because of the fact that this power is actually the static power being consumed by the device with a 'zero' frequency just after programming FPGA. In other words, this power con-

sumption cannot be considered of a dynamic power component. However, different clock rates costs different RTL designs on FPGA, which induces minor differences on static power.

Table 5.10: Power consumption for different clock rates for 128 Rules

Clock Rate: 100 MHz			
Phase	**Description**	**Current(A)**	**Power(W)**
1	Idle Phase: BVM and TCAM does not perform classification	1.14	0.969
2	Only BVM performs lookup for packets. TCAM Cache is empty	1.81	1.539
3	Both BVM and TCAM Cache perform lookup. Cache Hit Rate is at 71%5 at maximum for the traffic trace of 128 rules.	1.87	1.59
Clock Rate: 200 MHz			
Phase	**Description**	**Current(A)**	**Power(W)**
1	Idle Phase: BVM and TCAM does not perform classification	1.16	0.991
2	Only BVM performs lookup for packets. TCAM Cache is empty	2.66	2.261
3	Both BVM and TCAM Cache perform lookup. Cache Hit Rate is at 71%5 at maximum for the traffic trace of 128 rules.	2.75	2.338
Clock Rate: 250 MHz			
Phase	**Description**	**Current(A)**	**Power(W)**
1	Idle Phase: BVM and TCAM does not perform classification	1.17	1.002
2	Only BVM performs lookup for packets. TCAM Cache is empty	3.05	2.6
3	Both BVM and TCAM Cache perform lookup. Cache Hit Rate is at 71%5 at maximum for the traffic trace of 128 rules.	3.14	2.67

The comparison of power consumption between FASST designs that stores 512 rules and 128 rules is illustrated in Fig. 5.11. As observed, consumed power values during idle phase (phase 1) are very close to each other for each case due to static power

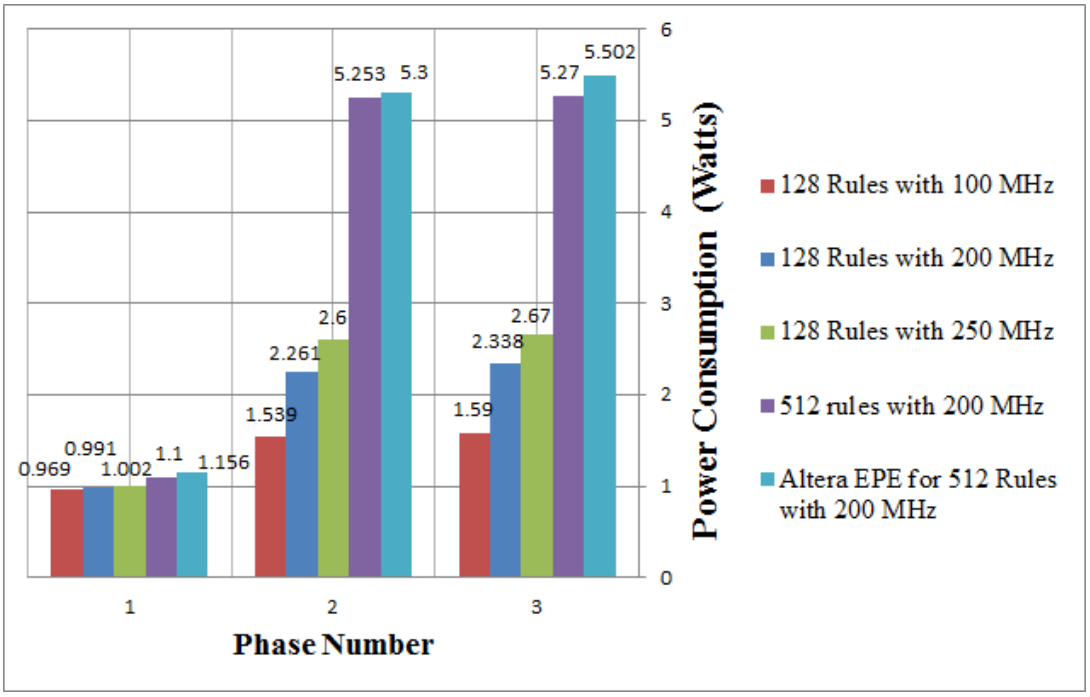

Figure 5.11: Power consumption for rule set size and clock rates

description. However, for phase 2 and phase 3, power values shows variations. Moreover, the difference between the power values in phase 2 and phase 3 for 128 rules case is higher than 512 rules case. This is mainly resulted from the cache size ratio to the overall rule set size. Since TCAM cache only stores 32 SDN rules with 15-tuple headers, the ratio of 32/512 is quite smaller than 32/128. Furthermore, Altera EPE power consumption results are given for 512 rules for FASST design in Fig. 5.11. As seen, our monitored power consumption and Altera EPE results are almost same for 512 rules in BVM, which verifies our power values.

5.10 Scalability of SRAM-based TCAM Design

In FASST, TCAM is designed and implemented considering on-chip resources such as embedded memory bits and logic gates compared to native TCAMs. In some SDN applications where higher cache rates are desired, TCAM size is needed to be increased. Therefore, scalability performance of TCAM Cache has a significant effect in our hardware architecture FASST. Fig. 5.12 shows the scalability of embedded memory blocks (M20K) and logic gates in our TCAM design with respect to increas-

ing number of rules. As observed, memory blocks increase sublinearly for different TCAM rules. This is mainly due to fact that the number of data-associate memories used in each *Stride TCAM* increases with discrete numbers, where each memory block can store 20Kbits. On the other hand, logic gate consumption increases at high rates. This results from the concurrent lookup operation of all *Stride TCAM* units inside TCAM design. This is because, as rule size increases, logic interface in each *Stride TCAM* becomes more complex to meet the timing requirements at the targeted clock rate.

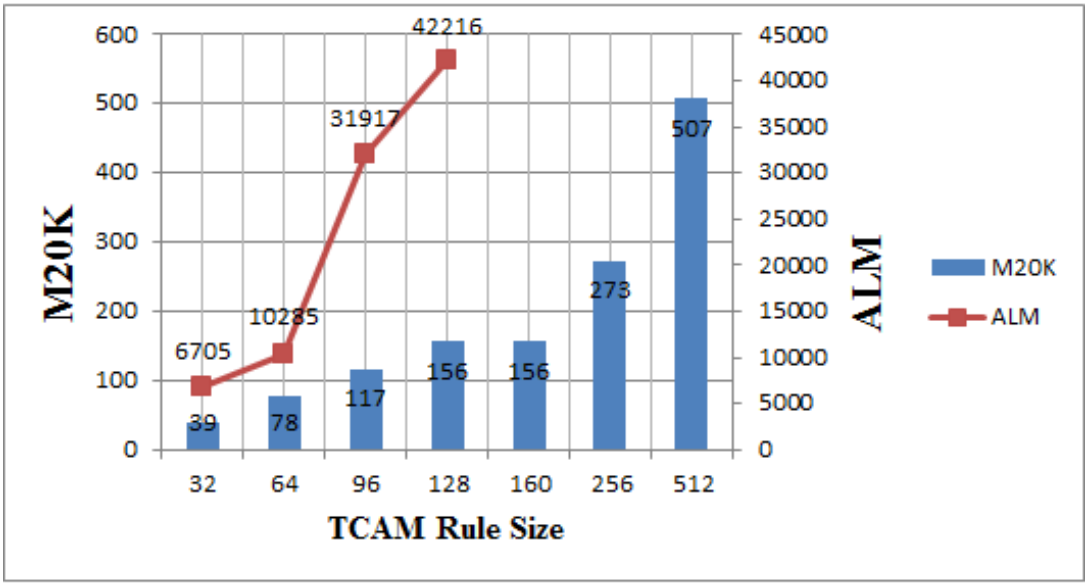

Figure 5.12: Scalability of embedded memory blocks (M20K) and logic gates in TCAM design

Fig. 5.13 shows the scalability of power consumption in TCAM design with respect to increasing rule size. As seen, most of the total power is consumed by RAM blocks instead of logic gates. Since RAM blocks usage (M20K) increases sublinearly with increasing rule size, power consumption does not increase rapidly, which leads to a satisfied scalability.

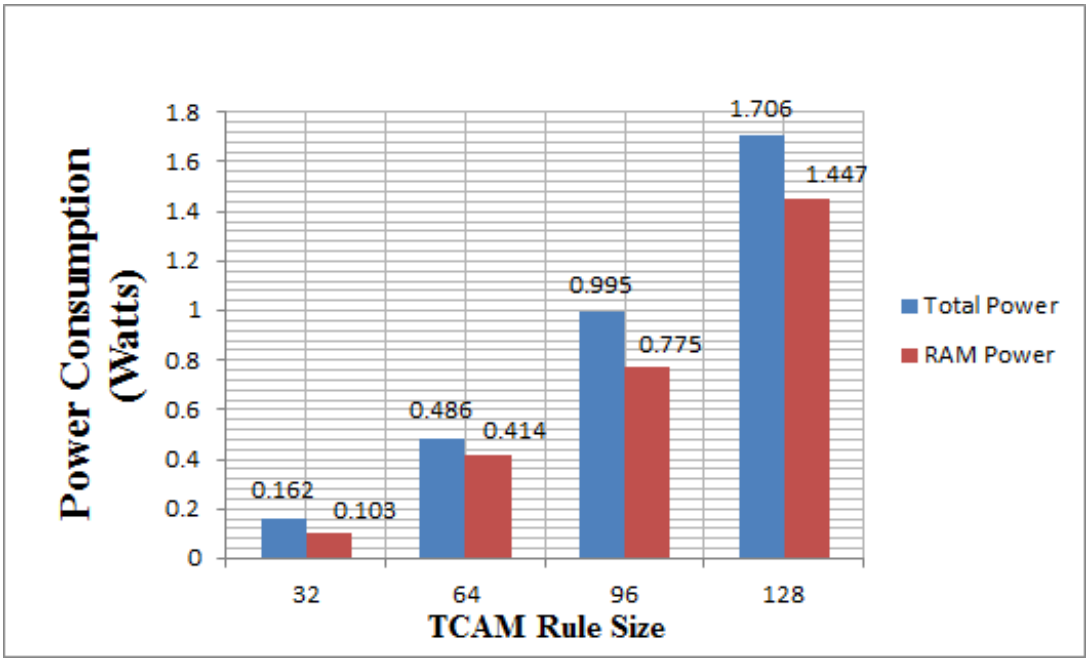

Figure 5.13: Scalability of power consumption in TCAM design

5.11 Scalability of Clock Rate, Latency and Resource Consumption of FASST with Rule Set Size

Clock rate and latency scalability performance of FASST hardware architecture with respect to increasing rule set size is given in Fig. 5.14. With increasing rule set size, FASST does not suffer from clock rate deterioration due to two dimensional pipeline architecture in BVM design. For example, even though rule size doubles at each phase, the clock rate does not decrease sharply. Blue columns and green columns in Fig. 5.14 show the BVM pipeline latency and the achieved average packet latency for 97.5% cache hit rate, respectively. The reason of the increase in BVM pipeline latency with different rule set size is due to the increase in the number of pipeline stages in BVM for fixed number of rule-set divisions (n=32). At high cache hit rates, overall average packet latencies are very low and they are very close to each other. This is because, at high cache hit rates, TCAM cache design with $t_{fast} = 3$ cycles, dominates the overall average latency.

Resource consumption scalability of FASST in terms of logic gates (ALM) and memory bits (Kbits) with respect to increasing rule set size is given in Fig. 5.15. Even though rule size doubles at each time from 64 to 512, memory bit consumption

Figure 5.14: Clock rate and latency scalability of FASST with increasing rule size

increases sub-linearly. On the other hand, logic gate consumption increases more rapidly. This is due to the fact that, as rule size increases, more memory blocks are used in each *Stride BVM* and the increase in these memory blocks is discrete due to more flow entries. However, since number of rules in BVM is doubled at each phase, the logic interface between *Stride BVM Controller* and *Stride BVM RAM* can nearly meet the timing requirements at the targeted clock rate. Therefore, in order to achieve the clock rate operation, more logic gates are utilized such as inserting more FFs for these interfaces. However, compared to other classification engines such as cross-producting of field labels, the logic gate consumption does not increase exponentially in FASST.

5.12 Comparison of Power, Latency and Throughput with Recent Work

Our proposed design FASST is compared with the recent study that performs packet classification with similar approach. For a fair classification, the method in dynamically updatable pipelined bit vector algorithm [8] is hardware based and implemented on FPGA. Moreover, since power consumption, throughput and latency are strictly dependent of rule set size and header width, the comparison is made for 1024 15-tuple OpenFlow rule entries. Dynamically updatable pipelined bit vector algorithm

Figure 5.15: Logic gate and memory bit scalability with increasing rule size

is implemented in a Xilinx Virtex 7 FPGA and the test results are given for 1024 rules. Hence, no scaling operation is made. The main result of performing comparison with 1024 is that, FASST is a pure hardware architecture and implemented in Altera Stratix V FPGA, which is a completely different hardware platform from other platform, Xilinx Virtex 7. Since power consumption scaling operations of dynamically updatable pipelined bit vector algorithm in [8] require Xilinx Power analyzer tools that we are not familiar with, our approach FASST is linearly scaled up to 1024 rules using Altera Early Power estimator tool.

The comparison of power consumption and throughput is shown in Table. 5.11. We observe that:

- FASST power consumption is higher than other pipeline implementation. In dynamically updatable pipelined bit vector algorithm, distributed-RAM blocks in Xilinx FPGAs are used in implementation. By using distributed-RAMs, routing delays between data associative elements for BV algorithm and logic gates are decreased leading a decrease in power consumption. However, in FASST, memory block RAMs in FPGA are used instead of logic gates in order to im-

plement data associative functional elements. This is a complete different implementation technique from other study. This is due to the fact that there is a high number of embedded (on chip) memory blocks that can be used to store 1024 SDN rules in Altera Stratix V FPGAs, compared to logic gates. Morever, FASST has a variety of management and control blocks due to utilization of an extra TCAM Cache, such as locality detection units. Such functions do not exist in other implementation. Therefore, logic gates are separated for other functions in FASST design. Logic gate term in Altera FPGAs corresponds distributed RAM term in Xilinx FPGAs.

- Power consumption can vary for different hardware platforms due to SoC design inside FPGAs. In other words, there are different number of hard IPs, embedded transceiver blocks or DSP blocks for Altera and Xilinx series FPGAs. This variety leads to different values of static power consumption.

- FASST can achieve a throughput value of 200 MPPS. Dynamically updatable pipelined bit vector algorithm achieves a throughput value of 324 MPPS if the clock rate in implementation is 324 MHz. However, resource utilization is given in [8] such that 95% of distributed RAMs are consumed for the compilation at 324 MHz. This high ratio of utilization can cause static timing errors at slightly different operation conditions.

Table 5.11: Power consumption and throughput comparison

	Power Consumption (Watts)	**Throughput (MPPS)**
FASST	8.3	200
Dynamically Updatable BVM	5.2	324

The latency comparison of FASST is given in Fig. 5.16. The latency of FASST is given by considering a cache hit rate of 0%, 82.5% and 97.5%. For cache hit rate with 0%, the comparison is carried out with dynamically updatable BV approach, where there is no parallel TCAM cache effect in FASST. Moreover, for higher cache hit rates, there is a shard decrease in FASST overall latency as shown in right part of in Fig. 5.16. Therefore;

- In [8], stride size s and bit vector length n is taken as 4 and 8, respectively, which is the optimum configuration for this study in terms of clock rate and power. As a result, since 1024 rules are stored with 356-bits headers, latency is given as 1024/4 + 356/8 = 217 clock cycles.

- Since FASST is a completely modular and pipelined architecture, latency can be scaled by adding extra Rule-BVMs. For 512 rules, each horizontal latency inside a Rule-BVM is 48 clock cycles, where stride size s is configurable with 3, 4, 6 and 8 bits. Moreover, each Rule-BVM has two internal pipelines, where each pipeline stores 16 rules. As a result, bit vector length is taken as 16. Therefore, for 512 rules; 48 + (512/16) = 80 clock cycles latency is achieved. If we scale this modular architecture to 1024 rules, then 32 Rule-BVMs are used instead of 16 Rule-BVMs. Moreover, each of these 32 Rule-BVM units again stores 32 rules with internal two dimensional horizontal pipelines. As a result, end to end latency will be 48 + (1024/16) =112 clock cycles. For *cache_hit_rate* values calculated based on hardware test results, an average latency of 5.7 clock cycles is achieved when *cache_hit_rate* equals to 97.5%.

- For networks with strong locality, FASST supports very low latency compared to other FPGA-based SDN classification implementation. Since other architecture does not provide a parallel caching mechanism, end to end latency of [8] is deterministic and independent of *cache_hit_rate*.

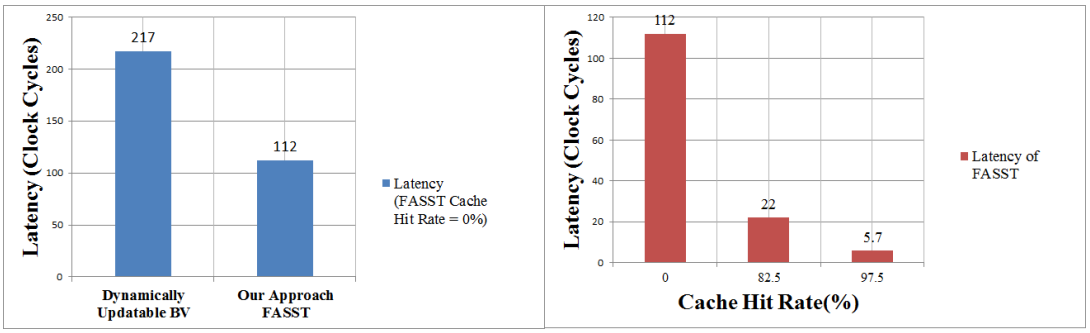

Figure 5.16: Latency comparison

5.13 FPGA Resource Utilization of FASST

Overall architecture of FASST is implemented on Altera Stratix-V FPGA, whose device number is (5SGXEA7N2F40C2). Altera FPGAs consists of basic building blocks known as Adaptive Logic Modules (ALMs) that can be configured to implemented logic functions, register functions and arithmetic functions [56]. Each ALM consists of a 6-input Look up Tables (LUT), 2 adders and 4 registers. Moreover, there are embedded memory blocks named M20K that can be used for on-chip RAMs. FASST mainly uses these two building blocks, ALM and M20K, to implement the required functionality. Table . 5.12 shows the utilization of FPGA resources for 512 and 128 rules, in terms of ALMs, registers, block memory bits and Digital Signal Processor (DSP) Blocks. As observed, resource consumption in FASST is linearly scalable with rule set size *N*.

Table 5.12: FPGA resource utilization for 512 and 128 rules

	FASST 512 Rules (15-tuple)	FASST 128 Rules (15-tuple)
ALM	165,737 (71%)	51,694 (22%)
Registers	349578	107368
Block Memory Bits	9,333,248 (18%)	5,736,960 (18%)
M20K	1740 (67.9%)	660 (25.7%)
DSP	2 ($< 1\%$)	2 ($< 1\%$)

For the case of storing 512 rules, resource utilization for the main units inside FASST is given in Table. 5.13. Nios II SoC design utilizes all of the DSP blocks, which are 2, during generation of dependency graph. Almost all of the block memory bits inside Nios II SoC are used for processor program memory. Moreover, most of the resources are utilized by BVM due to high number of SDN rule set. TCAM Cache, on the other hand, consumes quite low resources with respect to BVM because of storing a small set of rule entries.

Depending on these rule consumption values, FASST hardware architecture consumes most of ALM resources. On the other hand, utilization of block memory bits

Table 5.13: FPGA resource utilization of main blocks

	BVM	TCAM	Nios II SoC
ALM	117376(51%)	6705 (3%)	1395($< 1\%$)
Registers	285584	10177	1753
Block Memory Bits	6,160,384 (12%)	319,488 (0.61%)	2,143,232(4%)
DSP	0	0	2($< 1\%$)

is around 18%. Hence, scaling of FASST to support 1024 rules can be accomplished by applying some optimization techniques during designing RTL.

ALM and registers correspond to logic gate consumption in Altera Stratix V FPGA, and block memory bits correspond to embedded memory blocks. Each embedded memory block can store up to 20Kbits. Moreover, using discrete memory blocks results in an overhead usage of memory in term of bits. This is the main reason of the fact that the utilization of M20K blocks is higher than the utilization of total memory block memory bits in terms of percentages.

CHAPTER 6

TEST ENVIRONMENT OF FASST HARDWARE ARCHITECTURE

FASST is designed and implemented on Altera Stratix-V, which is a high performance FPGA in Altera FPGA families. For this purpose, Altera Signal Integrity (SI) Development Board is used. On this board, there is one Altera Stratix V FPGA, memory devices such as 128 MB flash memory, high speed serial interfaces, power monitor devices (LTC2978), temperature sensors, general I/O pins and clock generators [57]. We utilized Stratix V FPGA to implement our architecture, power monitor devices to measure current drawn by FASST at different operation phases, and some of I/O pins to connect the board to a PC in order generate syntetic rule set, and traffic traces.

Moreover, a serial interface add-on board is specifically designed for our architecture in order test and verification steps. On this designed board, there is RS-232 transceiver, voltage regulators and level shifters. Using this board, we can make a connection to a host PC through RS-232 interface. Therefore, synthetic rule sets and traffic traces can be sent to development board through this interface. Furthermore, run time power values are received from the receive channel of RS-232 interface.

Fig. 6.1 shows Altera SI Development board with the add-on interface board connected to it. As seen, the connection to host PC is made from add-on interface board. The purpose of using voltage regulators and levels shifter on this board is that all I/O pins of Stratix V FPGA is 2.5V, which causes an incompability with 5V RS-232 transmit and receive channels. Therefore, converting 5V signals to 2.5V is performed using these components.

A detailed illustration of the designed RS-232 Interface board is given in Fig. 6.2.

All synthetic rule sets and traffic traces described in section. 5.1 and section. 5.2 are generated in Matlab Environment. After that, MATLAB serial interface library is used to send all these traces to development kit. Similarly, run-time power values are read using serial interface library in MATLAB. RS-232 connection properties for the testing purposes is given in Table. 6.1.

Table 6.1: RS-232 connection parameters

Baudrate	**Data bits**	**Stop Bit**	**Parity**	**Buffer Size**
115200	8	1	None	1024 Bytes

The reason of using a slow interface such RS-232 is that there is no available high speed interfaces in order to test FASST at line rate. In other words, we tested and verified FASST at 200 MHz clock rate with 100% line utilization. Moreover, since each packet header is 356 bits, then 71.2 Gbps line rate is required to test our hardware architecture. As a result, using a slower interface, RS-232, we firstly load our traffic trace to lower INPUT FIFO as described in section. 4.1. After that, we send a query command through RS-232 interface to start the rule query phase inside FASST. Since Stratix-V FPGAs can provide high bandwidth due to abundant parallelism, a line rate operation is achieved.

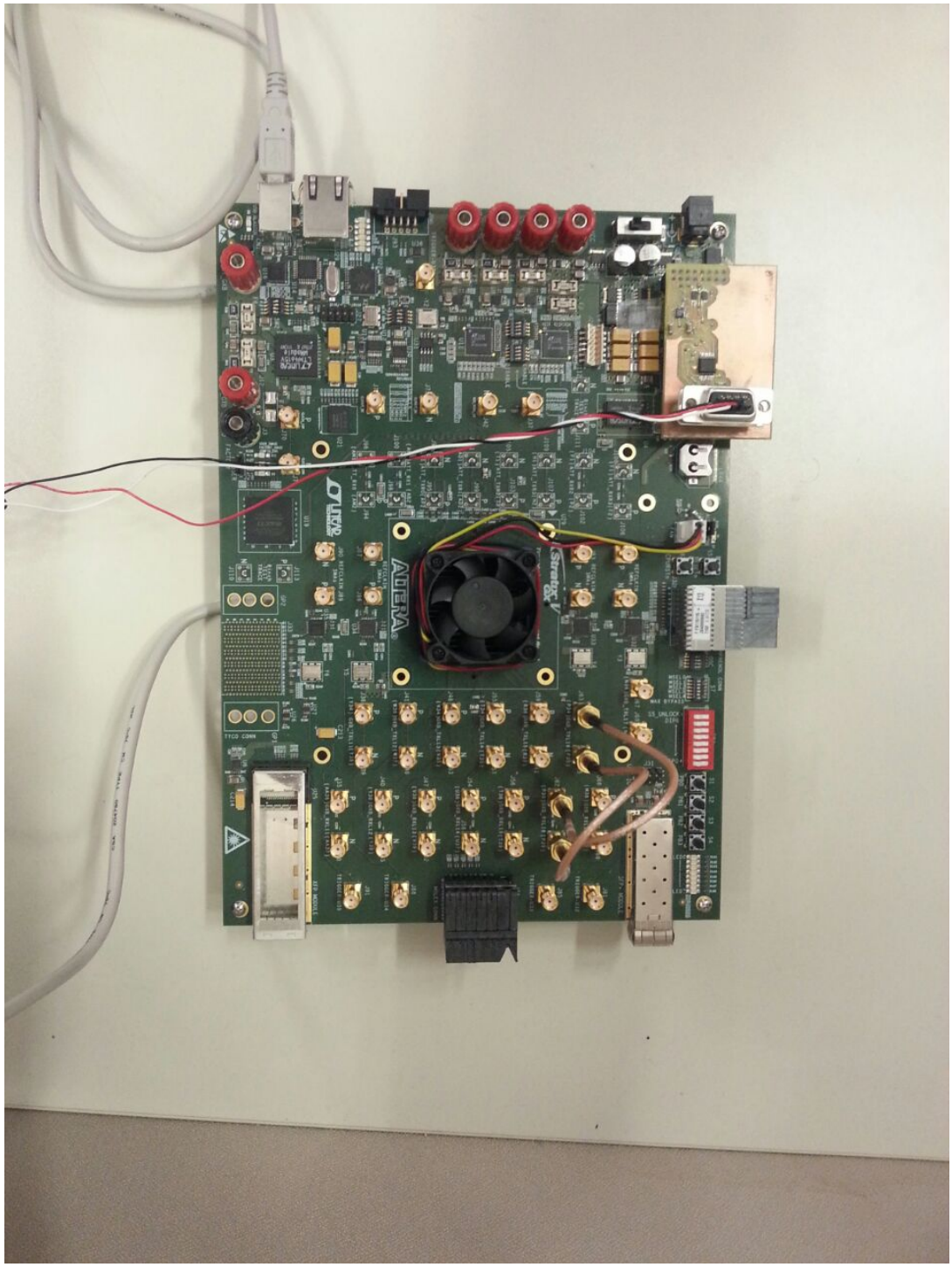

Figure 6.1: Altera SI development kit and Add-on interface board

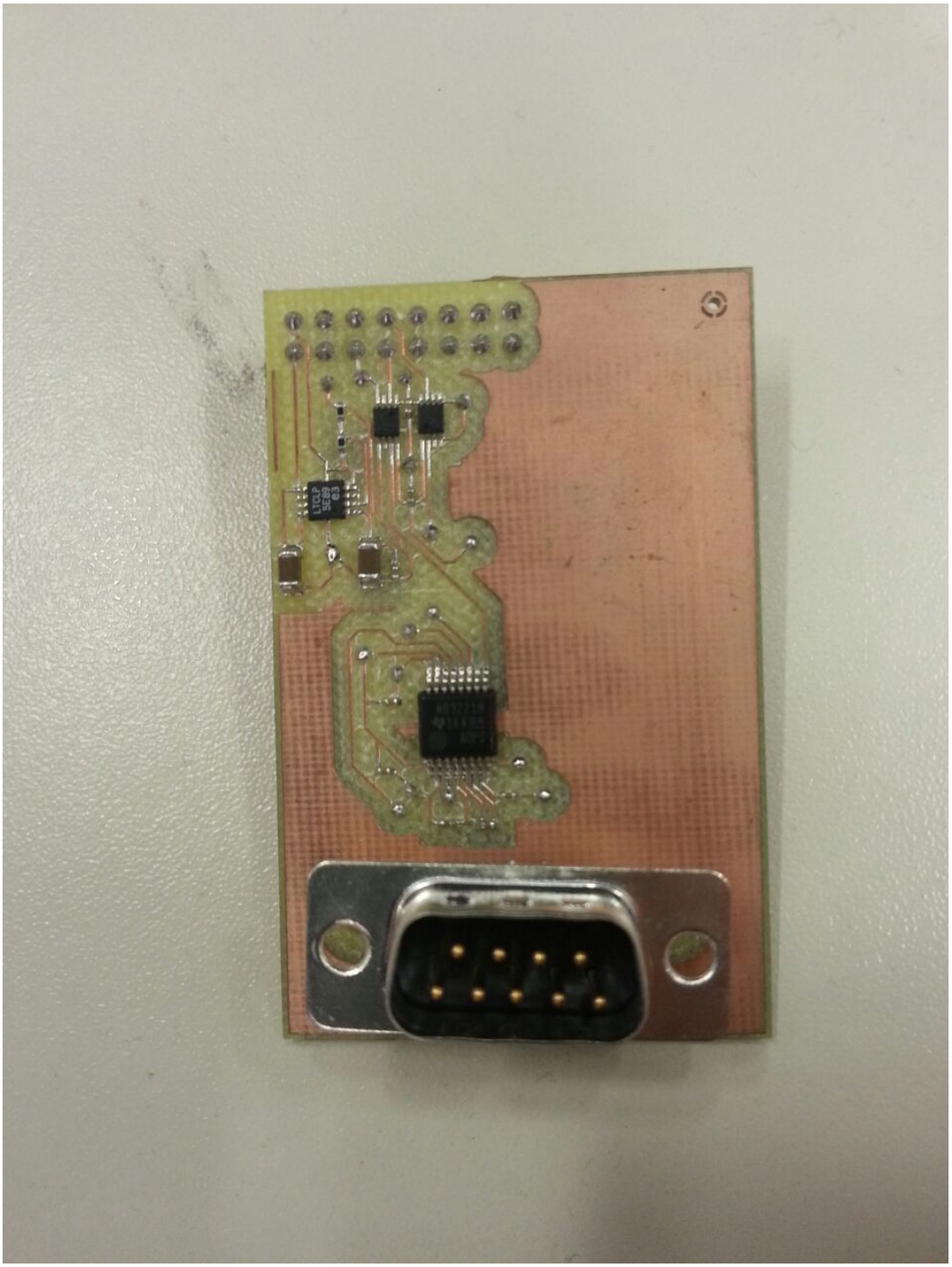

Figure 6.2: RS-232 Add-on interface board

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this work, Fast Scalable SDN Table (FASST), a complete hardware architecture for SDN flow table look up is proposed. The complete design of FASST at gate level detail together with full scale implementation on Altera Stratix V development board and performance evaluation results are presented.

FASST combines the throughput advantage of pipelined design with ultra low latency lookup time of TCAM by exploiting the network traffic locality. To this end, the frequently matched rules are dynamically determined and stored in the TCAM to decease the average look-up latency. The entire rule set is stored in a Bit Vector Machine with large pipeline latency. Hence, required control modules to run these two engines concurrently are also integrated in FASST.

In order to determine popular rules, our work exploits temporal locality dynamically with a shifting time window. Proposed architecture stores the frequently used rules together with their dependent rules. This is achieved by computing direct or indirect rule dependencies with an algorithm that is appropriate to run on the soft processor and generate an acyclic graph. Match results of frequently used rules and their dependent rules are provided in 15 ns due to fast lookup in TCAM Cache. For a network traffic that shows strong locality, match results are mostly provided by TCAM Cache and FASST achieves very low average latency. Hardware test results demonstrate that for an example of synthetic traffic trace, when cache hit rate is observed as 97.5%, the overall latency is decreased by 93.75%.

Based on real-time hardware tests, FASST achieves 200 MPPS throughput at run time

for 15-tuple headers with 512-rules flow table. If 356 bits in each packet header is considered, 71.2 Gbps throughput is achieved using BVM in FASST. For this throughput and rule set size, BVM consumes about 5.253 W in FASST if TCAM does not perform any lookup. When cache hit rate is observed as 97.5% for sample traffic trace, parallel processing of BVM and TCAM leads to a power consumption of 5.27 W. Moreover, if FASST stores 128 rules, $|\mathcal{R}|$=128, power consumption is monitored as 2.338 W for the parallel processing of BVM and TCAM at the same cache hit rate of 97.5% and clock rate of 200 MHz.

Compared to recent works that deploys hardware based classification for SDN environment, FASST presents real-time power monitoring instead of estimation techniques. It is observed that, consumed power is linearly scalable with respect to increasing rule size and clock rates.

The future work of FASST includes several developments regarding both BVM, TCAM Cache and SoC design. First of all, current FASST architecture does not support dynamic rule insertions during query operation. Instantaneous rule insertion and rule query operations will be supported at the future version of FASST by managing dual port on chip RAMs. Moreover, flow table size will be increased by optimizing embedded memory blocks (M20K) in our hardware platform. Using off-chip SRAMs or SDRAMs can be another option to store rule set at the expense of increasing latency. Furthermore, hardware test will be carried out using real-world network traffic in order to observe the cache hit rates and average latencies achieved by FASST at run-time. Furthermore, in order to design a overall SDN switch, FASST will be reorganized to preserve OpenFlow semantics by designing a memory mapped software agent interface in order to communicate a SDN controller outside.

Lastly, sequential processing of BVM and TCAM can be considered for scalability issues. Assume that, packets arriving to FASST are firstly sent to TCAM in order find a match. In case of TCAM match with $t_{slow} = 3$ cycles, these packets are routed to output without using BVM. For the packets that do not produce a positive match in TCAM, BVM can perform with $t_{slow} = 80$ cycles. For a network with 10 Gbps, if %90 of the packets produce a positive match in TCAM, then there will be no need to use a high latency classification engine such as BVM. However, for %10 of the

packets, the average latencies are further increases. The focus of this thesis is to decrease the high latencies introduced by high throughput two dimensional pipeline engines as in [8]. However, for the network applications where scalability is the primary concern, then our proposed architecture can be reconfigured for sequential processing of BVM and TCAM.

REFERENCES

[1] P. Veríssimo C. Rothenberg S. Azodolmolky D. Kreutz, F. Ramos and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103:14–76, January 2013.

[2] N. Mckeown. How sdn will shape networking. `http://www.comsnets.org/archive/2014/doc/NickMcKeownsSlides.pdf`. Accessed on October 2011.

[3] Algo Speed High Frequency Trading Solution. `http://www.cisco.com/c/en/us/solutions/industries/financial-services/financial-markets/algo-speed-solution.html`.

[4] Pica8. Sdn system performance. `http://pica8.org/blogs/?p=201`. Accessed on October 2012.

[5] D. Taylor E. Spitznagel and J. Turner. Packet classification using extended tcams. page 120, 4th November 2003.

[6] Y. R Qu and V.K. Prasanna. High-performance and dynamically updatable packet classification engine on fpga. *IEEE Transactions on Parallel and Distributed Systems*, 27:197–209, January 2016.

[7] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Infinite cacheflow in software-defined networks. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 175–180, 2014.

[8] OpenFlow archive. Openflow switch specification version 1.1.0 (wire protocol 0x02). `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.1.0.pdf#page=24`. Accessed on December 2011.

[9] E. Reinecke. Mapping the future of software-defined networking. http://goo.gl/fQCvRF. Accessed on 2014.

[10] I. Yokneam. Ezchip announces openflow 1.1 implementations on its np-4 100-gigabit network processor. http://www.ezchip.com/pr110713.htm. Accessed on 2011.

[11] NoviFlow. Noviswitch 1248 high performance openflow switch. http://205.236.122.20/gestion/NoviSwitch1248Datasheet.pdf. Accessed on 2013.

[12] G. A. Covington G. Appenzeller J. Naous, D. Erickson and N. McKeown. Implementing an openflow switch on the netfpga platform. *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 1–9, 2008.

[13] P. T. Congdon, P. Mohapatra, M. Farrens, and V. Akella. Simultaneously reducing latency and power consumption in openflow switches. *IEEE/ACM Transactions on Networking*, 22(3):1007–1020, June 2014.

[14] W. Jiang and V. K. Prasanna. Field-split parallel architecturefor high performance multi-match packet classification using fpgas. pages 188–196, August 2009.

[15] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. Devoflow: Scaling flow management for high-performance networks. *SIGCOMM Comput. Commun. Rev.*, 41(4):254–265, August 2011.

[16] S. Banerjee and K. Kannan. Tag-in-tag: Efficient flow table management in sdn switches. In *10th International Conference on Network and Service Management (CNSM) and Workshop*, pages 109–117, Nov 2014.

[17] S. Zhou, S. Zhao, and V. K. Prasanna. 400 gbps energy-efficient multi-field packet classification on fpga. In *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, pages 1–6, Dec 2014.

[18] Y. Kanizo, D. Hay, and I. Keslassy. Palette: Distributing tables in software-defined networks. In *INFOCOM, 2013 Proceedings IEEE*, pages 545–549, April 2013.

[19] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

[20] Pankaj Gupta and Nick McKeown. Packet classification on multiple fields. *SIGCOMM Comput. Commun. Rev.*, 29(4):147–160, August 1999.

[21] A. Lara, A. Kolasani, and B. Ramamurthy. Network innovation using openflow: A survey. *IEEE Communications Surveys Tutorials*, 16(1):493–512, First 2014.

[22] OpenFlow Switch Specification. Openflow switch specification version 1.3.0 (wire protocol 0x02). `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf#page=40`. Accessed on December 2012.

[23] P. Gupta and N. McKeown. Algorithms for packet classification. *Netwrk. Mag. of Global Internetwkg.*, 15(2):24–32, March 2001.

[24] Paul Francis Tsuchiya and Paul F. Tsuchiya. A search algorithm for table entries with non-contiguous wildcarding, 1991.

[25] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '98, pages 203–214, New York, NY, USA, 1998. ACM.

[26] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '98, pages 203–214, New York, NY, USA, 1998. ACM.

[27] T. Ganegedara and V. K. Prasanna. Stridebv: Single chip 400g+ packet classification. In *2012 IEEE 13th International Conference on High Performance Switching and Routing*, pages 1–6, June 2012.

[28] D. Yuan, X. Yang, X. Shi, B. Tang, and Y. Liu. Multi-protocol query structure for sdn switch based on parallel bloom filter. In *2014 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 206–211, Oct 2014.

[29] J. Aguilar-Saborit, P. Trancoso, V. Muntes-Mulero, and J. L. Larriba-Pey. Dynamic count filters. *SIGMOD Rec.*, 35(1):26–32, March 2006.

[30] Michael Mitzenmacher. Compressed bloom filters. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, PODC '01, pages 144–150, New York, NY, USA, 2001. ACM.

[31] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet classification using multidimensional cutting. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '03, pages 213–224, New York, NY, USA, 2003. ACM.

[32] D. E. Taylor and J. S. Turner. Scalable packet classification using distributed crossproducting of field labels. In *INFOCOM, 2005 Proceedings IEEE*, pages 269–280, 2005.

[33] F. Yu, R. H. Katz, and T. V. Lakshman. Efficient multimatch packet classification and lookup with tcam. *IEEE Micro*, 25(1):50–59, Jan 2005.

[34] Karthik Lakshminarayanan, Anand Rangarajan, and Srinivasan Venkatachary. Algorithms for advanced packet classification with ternary cams. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '05, pages 193–204, New York, NY, USA, 2005. ACM.

[35] F. Zane, Girija Narlikar, and A. Basu. Coolcams: power-efficient tcams for forwarding engines. In *INFOCOM 2003. Twenty-Second Annual Joint Conference*

of the IEEE Computer and Communications. IEEE Societies, volume 1, pages 42–52 vol.1, March 2003.

[36] Weirong Jiang. Scalable ternary content addressable memory implementation using fpgas. In *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '13, pages 71–82, Piscataway, NJ, USA, 2013. IEEE Press.

[37] J. Fong, X. Wang, Y. Qi, J. Li, and W. Jiang. Parasplit: A scalable architecture on fpga for terabit packet classification. In *2012 IEEE 20th Annual Symposium on High-Performance Interconnects*, pages 1–8, Aug 2012.

[38] Haoyu Song and John W. Lockwood. Efficient packet classification for network intrusion detection using fpga. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, FPGA '05, pages 238–245, New York, NY, USA, 2005. ACM.

[39] P. Gupta and N. McKeown. Classifying packets with hierarchical intelligent cuttings. *IEEE Micro*, 20(1):34–41, Jan 2000.

[40] T. J. Lin, W. Zhang, and N. K. Jha. A fine-grain dynamically reconfigurable architecture aimed at reducing the fpga-asic gaps. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(12):2607–2620, Dec 2014.

[41] W. Jiang and V. K. Prasanna. Scalable packet classification on fpga. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(9):1668–1680, Sept 2012.

[42] V.K. Prasanna S. Zhou, W. Jiang. A flexible and scalable high-performance openflow switch on heterogeneous soc platforms. volume 27, pages 1–8, December 2014.

[43] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 19–19, Berkeley, CA, USA, 2012. USENIX Association.

[44] Ramana Rao Kompella, Kirill Levchenko, Alex C. Snoeren, and George Varghese. Every microsecond counts: Tracking fine-grain latencies with a lossy difference aggregator. *SIGCOMM Comput. Commun. Rev.*, 39(4):255–266, August 2009.

[45] Thorsten Brants, Ashok C. Popat, Peng Xu, Franz J. Och, Jeffrey Dean, and Google Inc. Large language models in machine translation. In *In EMNLP*, pages 858–867, 2007.

[46] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. It's time for low latency. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 11–11, Berkeley, CA, USA, 2011. USENIX Association.

[47] Nadi Sarrar, Steve Uhlig, Anja Feldmann, Rob Sherwood, and Xin Huang. Leveraging zipf's law for traffic offloading. *SIGCOMM Comput. Commun. Rev.*, 42(1):16–22, January 2012.

[48] Willibald Doeringer, Günter Karjoth, and Mehdi Nassehi. Routing on longest-matching prefixes. *IEEE/ACM Trans. Netw.*, 4(1):86–97, February 1996.

[49] REANZZ. `http://reannz.co.nz/`.

[50] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 9–9, 2012.

[51] Jon C. R. Bennett, Craig Partridge, and Nicholas Shectman. Packet reordering is not pathological network behavior. *IEEE/ACM Trans. Netw.*, 7(6):789–798, December 1999.

[52] Vern Paxson. Automated packet trace analysis of tcp implementations. *SIGCOMM Comput. Commun. Rev.*, 27(4):167–179, October 1997.

[53] Piet Van Mieghem Xiaoming Zhou. *Passive and Active Network Measurement*, volume 3015. Springer Berlin Heidelberg, 2004.

[54] LTC2978 Octal Digital Power Supply Manager with EEPROM. http://www.linear.com/product/LTC2978.

[55] Altera Early Power Estimator Tool. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_epe.pdf.

[56] Stratix V Device Handbook. https://www.altera.com/en_US/pdfs/literature/hb/stratix-v/stx5_core.pdf.

[57] Stratix SI Development Kit. https://www.altera.com/products/boards_and_kits/dev-kits/altera/kit-transceiver-si-stratix-v.html.