法政大学学術機関リポジトリ
HOSEI UNIVERSITY REPOSITORY

PDF issue: 2025-03-14

OpenFlow1.3 スイッチにおけるパケット分類の FPGA 実装に関する提案

SHIMIZU, Takeshi / 清水, 健志

(出版者 / Publisher)
法政大学大学院情報科学研究科

(雑誌名 / Journal or Publication Title)
法政大学大学院紀要. 情報科学研究科編 / 法政大学大学院紀要. 情報科学研究科編

(巻 / Volume)
12

(開始ページ / Start Page)
1

(終了ページ / End Page)
6

(発行年 / Year)
2017-03-31

(URL)
https://doi.org/10.15002/00014408

OpenFlow1.3 スイッチにおけるパケット分類の FPGA 実装に関する提案
A Proporsal to Implementing Packet Classification for OpenFlow1.3 Switch on FPGA

清水　健志 *
Takeshi Shimizu
法政大学大学院 情報科学研究科 情報科学専攻
Email: takeshi.shimizu.8s@stu.hosei.ac.jp

***Abstract*—OpenFlow is a unique concept to realize software-defined networking (SDN). It realizes the network flexibility by separating control plane and data plane. Data plane is implemented by OpenFlow switch. We discuss how to implement OpenFlow switch on FPGA (Field-programmable gate array). FPGA implementations of the OpenFlow switch have been researched. But their architectures are for only 5-tuple fields or OpenFlow1.0. OpenFlow is dramatically updated in short time, so we should design new architecture for newer OpenFlow specification. In this study, we propose how to implement the packet classification for OpenFlow1.3 switch on FPGA. Our idea is to use packet classification algorithm which contains a strideBV algorithm and a range bit vector encoding algorithm. We apply them to 13 fields which are required in implementations.**

1. 序論

近年ネットワークにおいて SDN(Software Defined Networking) という考えが盛んになっている。これはネットワーク構成を従来のようにハードウェア同士を物理的に接続して構築するのではなく、ソフトウェアでネットワークを動的に構築するものである。従来のネットワークはサーバやルータなどの機器を追加・除去する際、機器同士の接続をやり直したり個々の機器ごとに設定を行わなければならなかった。このような煩雑で時間がかかる作業をソフトウェアのプログラムで実現しようというものである。これによりネットワークが柔軟かつ簡単に管理できるようになり、急なネットワークの変更も対応できるようになる。

SDN が盛んになっている理由として、近年クラウド環境が台頭し、一つのサーバを複数のユーザが動的に利用したり、ネットワーク内を仮想サーバが動的に移動したりすることが短時間に大量に発生するようになったことがあげられる。クラウドのこの即応性からユーザやデータ量は増加すると考えられる。しかし、従来のネットワーク管理ではこのような動的なネットワーク管理が非常に困難である。そこでネットワークをソフトウェアで管理する SDN によって、ネットワークの変更を素早く行い、クラウドのネットワークに対応させる。

OpenFlow[1] は SDN を実現するための技術として考案された。OpenFlow の特徴として、従来のネットワーク機器では同一機器内に組み込まれていた経路制御機能（コントローラプレーン）とデータ転送機能（データプレーン）を分離したことにある。OpenFlow では経路制御を扱う OpenFlow コントローラとデータ転送を行う OpenFlow スイッチを OpenFlow プロトコルで通信することでソフトウェアでの経路制御を実現している。

コントローラはほとんどの場合ソフトウェアで実装され、Ryu[2] や Trema[3] といった多種多様なオープンソースのコントローラが使用されている。一方、OpenFlow スイッチの実装に関してはソフトウェア、ハードウェア両面から研究されてきた。ソフトウェアでの実装は Open vSwitch[4] や Lagopus[5] などがあり、それらを用いた研究もおこなわれている。ハードウェアでの実装は FPGA による実装が多くみられ、ほかには GPU を用いた実装もある。

これらの研究は OpenFlow の初期の規格である OpenFlow1.0[6] の仕様に基づいている。しかし OpenFlow1.0 にはいくつかの問題がある。

- OpenFlow1.0 はスイッチに搭載されている TCAM(Ternary Content Addressable Memory) を使用することを前提とした仕様である。しかしこの TCAM は消費電力が大きく、単位容量あたりの価格も高い。そのため、ユーザが必要な数のフローエントリを使用できない可能性がある。
- OpenFlow1.0 はフローエントリにないパケットをコントローラに毎回問い合わせるため、DDoS 攻撃に弱い。

以上の問題から、OpenFlow1.0 での運用は限界に達しつつある。
OpenFlow は ONF(Open Networking Foundation)[7] によって仕様の変更が頻繁に行われている。OpenFlow スイッチの実装に関する研究はほとんどが OpenFlow1.0 を対象にしている。しかし最近では OpenFlow1.3 に対応したスイッチの商品が増えてきており、FPGA 実装のほうも 1.3 に対応しなければならなくなった。1.0 と 1.3 では仕様が大きく異なるため、それに合わせたアルゴリズムに変更しなければならない。

本研究では OpenFlow スイッチの機能のうち、パケットのヘッダとフローテーブルとのマッチング部分に着目する。パケット分類に関する研究は昔から様々な研究がなされているが、OpenFlow スイッチのパケットマッチングに関する研究は限られている。そこで本研究は OpenFlow1.3 におけるパケットマッチングを考える。

2. 研究背景

序章でも述べた通り、OpenFlow スイッチをどのように実装するかについてさまざまな議論がなされている。ネットワーク機器を提供しているベンダーは OpenFlow に対応したスイッチを販売している。そのためネットワーク管理者はネットワーク構築の際、このようなスイッチを購入して OpenFlow ネットワークを構築する。

そんな中 OpenFlow スイッチを FPGA に実装するという研究が行われてきた。OpenFlow スイッチを FPGA に実装する理由として以下があげられる。

1) ネットワークを複数のベンダーが混合するマルチベンダーにしやすい

* Supervisor: Prof. Yamin Li

2) 将来 OpenFlow のバージョンが上がり、仕様が変更になっても対応できる

3) SRAM にフローテーブルを実装することができる

まずは 1) について説明する。ネットワークは様々な機器を接続することで構築するが、その機器を販売しているベンダーごとに機器の扱い方が異なる。例えば命令コマンドが異なっていたり、同じ性質の機器でも機能や挙動が異なっていたりする。そのため異なるベンダーの機器を使用してネットワークを構築すると予期せぬトラブルに見舞われることが多くなる。またベンダーからのサポートもほかのベンダーの機器がネットワークに存在すると十分なサポートを受けられない可能性がある。そのためネットワークは同じベンダーの機器のみで構成するのが望ましいとされている。
これは OpenFlow スイッチについても同様である。ネットワーク管理者はすでにあるネットワークを OpenFlow で制御しようと考えたときに、ネットワーク内の機器と同じベンダーの OpenFlow スイッチを購入することになる。したがって OpenFlow を導入しているにもかかわらずネットワークは単一のベンダーに依存することになり、セキュリティホールが発見された場合その影響はネットワーク全体に及ぶことになる。

しかし FPGA による実装であればどのベンダーにも対応した OpenFlow スイッチを実装することが可能である。FPGA はユーザが自由に回路設計を行うことができるので、各ベンダーに対応した回路を設計することが可能だからである。したがって FPGA に実装した OpenFlow スイッチをネットワークに導入すれば、マルチベンダで構成されるネットワークを構築することが可能である。

マルチベンダの利点として、各ベンダーの得意な部分だけを採用することでより効率的なネットワークを構築できることである。OpenFlow スイッチを FPGA に実装することでそれが可能になる。

次に 2) について説明する。OpenFlow は OpenFlow1.0 からバージョンアップを繰り返しており、そのたびに仕様の変更が行われている。OpenFlow スイッチをベンダーから購入した場合、バージョンアップがあった場合スイッチを買いなおす必要がある。一方、OpenFlow スイッチを FPGA に実装した場合、バージョンアップがあっても FPGA ボードは買いなおす必要がなく、回路の書き換えだけで対応することができる。

最後に 3) について説明する。通常のネットワーク機器では TCAM にパケットマッチングする際のルールを書き込む。しかし、序章で示した通り TCAM は容量単価が高いためユーザが必要とするルール数を確保できない可能性がある。また消費電力も大きいため、データセンターで TCAM を多用すると運用コストが大きくなってしまう。これは OpenFlow スイッチでも同様である。

FPGA ボードには SRAM が搭載されており、そこにフローテーブルを書き込めば TCAM より安く、より多くのフローエントリを確保することができるのである。

3. OpenFlow の概要

3.1. OpenFlow のアーキテクチャ

OpenFlow ではスイッチに入ってきたパケットの処理方法をフローエントリというもので定義している。フローエントリはヘッダフィールド、カウンタ、アクションからなる。ヘッダフィールドはこのフローエントリにマッチするための条件を記述し、アクションはマッチしたパケットの処理を記述する。カウンタはマッチ回数を記録する。

フローエントリはスイッチ内にまとめられてフローテーブルとして保存される。入力パケットが来たら、スイッチはパケットのヘッダを解析し、どのフローテーブルのヘッダフィールドとマッチするか検索する。マッチしたフローエントリがあった場合、そのフローエントリに書かれているアクションに従って入力パケットを処理する。アクションには次のようなものがある。

- Forward:パケットを指定ポートから送出
- Drop:パケットを破棄
- Modify-Field:パケットのヘッダの書き換え

例えば Forward の場合、パケットを指定されたポートに送出するほかにも受信した物理ポート以外のすべてのポートに送出したり、コントローラに送出することもできる。Modify-Field は IP アドレスや MAC アドレスなどの書き換えを行うことができる。このような機能によりスイッチはルータやファイヤウォール、ロードバランサなどの枠割を担うことができる。入力パケットがどのフローエントリにもマッチしなかった場合、コントローラに該当パケットをどうするか問い合わせる。コントローラは問い合わせが来たらそのパケットにマッチするフローエントリを作成し、スイッチにフローテーブルの更新を問い合わせる。

3.2. OpenFlow1.3 の概要

OpenFlow1.3[8] では 1.0 で採用されていた固定長のヘッダフィールドが廃止され、可変長の TLV 構造を持つ OXM(OpenFlow eXtensible Match) が採用されている。OpenFlow1.3 における OXM マッチフィールドは 40 個ある。そのうち実装が必須と定められているフィールドは 13 個である。表 1 にそのフィールドと概要を示す。

表 1. OpenFlow1.3 の実装が必須となる OXM マッチフィールド

フィールド	ビット長	内容
OXM_OF_IN_PORT	32	入力ポート
OXM_OF_ETH_DST	48	受信先 MAC アドレス
OXM_OF_ETH_SRC	48	送信元 MAC アドレス
OXM_OF_ETH_TYPE	16	イーサネットフレームタイプ
OXM_OF_ IP_PROTO	8	IP プロトコル番号
OXM_OF_IPV4_SRC	32	送信元 IPv4 アドレス
OXM_OF_IPV4_DST	32	受信先 IPv4 アドレス
OXM_OF_IPV6_SRC	128	送信元 IPv6 アドレス
OXM_OF_IPV6_DST	128	受信先 IPv6 アドレス
OXM_OF_TCP_SRC	16	送信元 TCP ポート番号
OXM_OF_TCP_DST	16	受信先 TCP ポート番号
OXM_OF_UDP_SRC	16	送信元 UDP ポート番号
OXM_OF_UDP_DST	16	受信先 UDP ポート番号

またこれらフィールドにはそれぞれ前提条件 (pre-requisite) が定められている。これはそのマッチフィールドを解析する際に前提条件に書かれた条件を満たすパケットだけマッチフィールドを使用するものである。例

えば IPV4_SRC、すなわち IPv4 送信元のマッチフィールドを使用したければ ETH_TYPE が 0x0800 である必要がある。表 1 に示したフィールドに対する前提条件を表 2 に示す。

表 2. OXM フィールドの先行条件

フィールド	先行条件
OXM_OF_IN_PORT	なし
OXM_OF_ETH_DST	なし
OXM_OF_ETH_SRC	なし
OXM_OF_ETH_TYPE	なし
OXM_OF_IP_PROTO	ETH_TYPE=0x0800 or 0x86dd
OXM_OF_IPV4_SRC	ETH_TYPE=0x0800
OXM_OF_IPV4_DST	ETH_TYPE=0x0800
OXM_OF_IPV6_SRC	ETH_TYPE=0x86dd
OXM_OF_IPV6_DST	ETH_TYPE=0x86dd
OXM_OF_TCP_SRC	IP_PROTO=6
OXM_OF_TCP_DST	IP_PROTO=6
OXM_OF_UDP_SRC	IP_PROTO=17
OXM_OF_UDP_DST	IP_PROTO=17

4. パケット分類のアルゴリズム

本章では、この研究に用いるパケット分類のアルゴリズムについて説明する。

4.1. パケット分類の構図

ハードウェアにおけるパケット分類には大きく分けて 2 種類の方法がある。Desition-tree と Decomposition である。Desition-tree はルールをある規則に従って再帰的に分類していくアルゴリズムである。一方、Decomposition は各フィールドを独立に検査しマッチングしたルールを抽出、その後すべてのフィールドでマッチしたルールが選ばれるアルゴリズムである。Desition-tree をもとにしたパケット分類アルゴリズムは総じてメモリ消費量が大きく、メモリが限られている FPGA には適さない。したがって本研究では Decomposition をもとにしたアルゴリズムを採用する。
Decomposition にも 2 つのアプローチがある。Prefix search と Range search である。Prefix search はフィールドの一部のビット列がルールと同じであればマッチするアルゴリズムである。Range search はマッチする範囲を指定しておき、ヘッダの値がこの範囲内にあればマッチするアルゴリズムである。本研究ではフィールドによってこの 2 つを使い分ける。次節から本研究で使用するアルゴリズムを示す。

4.2. Field-Split Bit Vector(FSBV) Algorithm

FSBV[9] は BV アルゴリズムを改良し、メモリ消費量を抑えた Decomposition ベースのアルゴリズムである。ルール数を N、フィールド内のビット数を W とする。i 番目のルール $R_i(i = 0, 1, \cdots, N-1)$ を W 個の 1 ビットの値

$$R_i = T_{i,W-1}, T_{i,W-2}, \cdots, T_{i,0}(i = 0, 1, \cdots, N-1)$$

に分割する。N ビットの bit vector を $2W$ 個用意し,bit vector V_i を、

$$V_i = B_{i,N-1}, B_{i,N-2}, \cdots, B_{i,0}(i = 0, 1, \cdots, 2N-1)$$

と定義する。ルール N_i に対し、$B_{i,j}$ の値によって以下のように B_i の値を定める。

$$\begin{cases} B_{2j,i} = 1 \\ B_{2j+1,i} = 1 \end{cases} \quad (T_{i,j} = *)$$

$$\begin{cases} B_{2j,i} = 0 \\ B_{2j+1,i} = 1 \end{cases} \quad (T_{i,j} = 1)$$

$$\begin{cases} B_{2j,i} = 1 \\ B_{2j+1,i} = 0 \end{cases} \quad (T_{i,j} = 0)$$

この式の意味するところは、i 番目のルールの j ビット目の値によって $B_{2j,i}$ と $B_{2j+1,i}$ の値が決まるということである。V_{2j} は各ルールの j ビットが 0 であるかどうかを示し、V_{2j+1} は j ビットが 1 であるかどうかを示す。$T_{i,j}$ がアスタリスク、すなわちドントケアの場合は $B_{2j,i}$ と $B_{2j+1,i}$ の両方に 1 を格納する。これをすべてのビット、すべてのルールに対して行い、bit vector V_i を完成させる。
パケットが来たら対応するフィールドを抽出し、W ビットのフィールドを W 個の 1 ビット

$$P_{W-1}, P_{W-2}, \cdots, P_0$$

に分割する。各 $P_i(i = W-1, W-2, \cdots, 0)$ に対して bit vector V_p を以下のように選択する。

$$V_p = \begin{cases} V_{2i} & (P_i = 0) \\ V_{2i+1} & (P_i = 1) \end{cases}$$

すべての P_i に対して V_p を求めたら、すべての V_p の and をとる。その結果値が 1 となるビット $B_{i,j}$ がこのパケットにマッチするルールになる。このアルゴリズムでは実行時間が $O(W)$、メモリ消費量が $2WN = O(WN)$ となる。

4.3. StrideBV

StrideBV[10] は FSBV を改良した Decomposition ベースのアルゴリズムである。FSBV がフィールドを W 個の 1 ビットに分割するのに対し、StrideBV は任意の値 k を用意し、$\frac{W}{k}$ 個の k ビットのサブフィールド

$$s_{\frac{W}{k}-1}, s_{\frac{W}{k}-2}, \cdots, s_0$$

に分割する。各サブフィールドは k ビット分の幅があるため各サブフィールドは 2^k 個の Nbit vector を持つ。各ルールは対応するサブフィールド s_i 部分のビットと比較し、対応するビット部分を 1 にしそれ以外を 0 とする。例えば、$k = 4$ のとき、bit vector は 0000 から 1111 までの 16 個を保持する。この時入力パケットのフィールドの値が 0101 の場合、0101 に該当する bit vector を取得

する。入力パケットとマッチングする際、サブフィールドと対応するビット幅を比較し、マッチした bit vector を取得する。$k = 1$ のとき FSBV と同じになる。このアルゴリズムでは実行時間は $O(W/k)$、メモリ消費量は $\Theta(2^k \times N \times \frac{W}{k})$ である。k の値によって実行時間とメモリ消費量は変化し、この 2 つは反比例の関係にある。そのため実行時間とメモリ消費量のバランスを考えて k の値を定めるのが重要となる。

4.4. Range Bit Vector Encording(RBVE) Algorithm

RBVE[11] アルゴリズムは StrideBV を基にした range search アルゴリズムである。このアルゴリズムではマッチング範囲を 16 ビットと固定している。16 ビットのマッチング範囲を $[LB, UB]$ と定義する。LB は範囲の下限、UB は範囲の上限である。この範囲を d ビットのサブ範囲に分割する。ただし $d = 1, 2, 4, 8$ のいずれかとする。これにより $s(= 16/d)$ 個の d ビットサブ範囲ができ、s ステージのパイプラインとして実装することができる。例えば $d = 4$ のとき、$[LB, UB]$ を $[LB_i, UB_i](i = 1, 2, 3, 4)$ に分割することができる。入力フィールドを A とし、これも $A_i(i = 1, 2, 3, 4)$ に分割する。そして $i = 1$ から順番に $[LB_i, UB_i]$ と A_i の関係を調べていく。その関係は以下のように考えられる。

- $A_i < LB_i$
- $A_i = LB_i < UB_i$
- $LB_i < A_i < UB_i$
- $LB_i < A_i = UB_i$
- $UB_i < A_i$

もし $LB_1 < A_1 < UB_1$ であった場合は $LB < A < UB$ であるといえるので、A は範囲 $[LB, UB]$ にマッチしたということになる。しかし $LB_2 < A_2 < UB_2$ であるからといって $LB < A < UB$ であるとは限らない。このほかにも $LB_1 < UB_1 = A_1, A_1 = LB_1 < UB_1$ といった結果が考えられるが、これらはいずれもそれだけでマッチしたとは言えず、この後のステージの結果による。つまりステージの結果によってそのあとのステージの結果が変わってくる。そのため各ステージごとにそのステージの関係はどういったものであるかを表した信号が必要である。その信号はステージによってビット数と意味を変える。その信号について表 4.4 に各値の意味を示す。なお、first stage は一番最初のステージ、last stage は一番最後のステージ、middle stage は first stage と last stage の間にあるステージを指す。

表 3. RBVE における FIRST STAGE の信号

ビットの値	意味
111	無条件でマッチ
001	この後のステージで $A_i > LB_i$ であればマッチ
100	この後のステージで $A_i < UB_i$ であればマッチ
010	この後のステージで $LB_i < A_i < UB_i$ であればマッチ
000	無条件でミスマッチ

例えば first stage において 111 であった場合、$LB_1 < A_1 < UB_1$ であるので、この後のステージの結果にかかわらずマッチである。001 の場合、この後のステージの結果がすべて $A_i > LB_i$ であった場合にマッチし、それ以外の場合はミスマッチとなる。同様に 100 の場合、

表 4. RBVE における MIDDLE STAGE の信号

条件	ビットの値
$A_i = UB_i$	$c_3 = 1$
$A_i < UB_i$	$c_2 = 1$
$A_i = LB_i$	$c_1 = 1$
$A_i > LB_i$	$c_0 = 1$

表 5. RBVE における LAST STAGE の信号

条件	ビットの値
$A_s \leq UB_s$	$f_1 = 1$
$LB_s \leq A_s$	$f_0 = 1$

この後のステージの結果がすべて $A_i < UB_i$ であった場合にマッチし、それ以外の場合はミスマッチとなる。出力信号のビット数はステージによって異なる。$d = 4$ のとき、$i = 1$ のステージでは 3 ビット、$i = 2, i = 3$ では 4 ビット、$i = 4$ では 2 ビットとなる。各ステージで生成した出力信号を総合してマッチするかどうか決定する。
RBVE は middle stage と final stage に 5 入力 LUT(Look Up Table) を使用している。これはそのステージでのマッチング結果と前ステージからの出力信号を入力として次のステージの出力信号を生成するためのものである。middle stage には 4 個の LUT($e1, e2, e3, e4$)、last stage には 1 個の LUT($e5$) がある。

5. 提案手法

本節では OpenFlow1.3 のパケット識別における提案手法を説明する。まず、各フィールドがどのようにマッチングするかを説明し、その後全体の説明に入る。

5.1. 各フィールドに対してのマッチング方法

3 節で研究対象となるフィールドを示したが、フィールドによってマッチングのアルゴリズムは異なる。フィールドによってルールに書かれる傾向が異なるためである。各フィールドに対するマッチング方法を表 6 に示す。RBVE はマッチする範囲を指定してマッチングを行うアルゴリズムであるため、どの通信においても使用する範囲が決まっているフィールドに適用するのが望ましい。TCP と UDP のポート番号は 85% 以上の通信で 0 から 1023 までの番号が使われている。そのため、TCP/UDP 関連のフィールドは RBVE を用いるのが望ましい。その他のフィールドはパケットによって値は様々で、OXM TLV である一定の範囲に定まることが難しいと考えられるので StrideBV を用いる。

パラメータ k の値は $k = 4$ を基本とする。これは [12]、[11] において $k = 4$ のときに実行時間とメモリ消費量のバランスが最も良いとされているからである。IPv6 のフィールドのみ $k = 8$ となるのは、$k = 4$ だと IPv6 におけるパイプラインの段数が多すぎるからである。提案手法全体のアーキテクチャを考えたとき、1 つのフィールドに 32 ものパイプライン処理を行うとそこがボトルネックとなる。そのため k の値を増やす必要があるが、

FPGA ボード上にある RAM にも限りがあるため、そのバランスが取れた値にしなければならない。そこで調べたところ、$k = 8$ のときにパイプライン段数が 16、1 ルールあたりのメモリ消費量が 4096 ビットとなり最もバランスが取れている値となった。

表 6. マッチング方法

マッチフィールド	マッチング方法
OXM_OF_IN_PORT	StrideBV($k = 4$)
OXM_OF_ETH_DST	StrideBV($k = 4$)
OXM_OF_ETH_SRC	StrideBV($k = 4$)
OXM_OF_ETH_TYPE	StrideBV($k = 4$)
OXM_OF_IP_PROTO	StrideBV($k = 4$)
OXM_OF_IPV4_SRC	StrideBV($k = 4$)
OXM_OF_IPV4_DST	StrideBV($k = 4$)
OXM_OF_IPV6_SRC	StrideBV($k = 8$)
OXM_OF_IPV6_DST	StrideBV($k = 8$)
OXM_OF_TCP_SRC	RBVE($k = 4$)
OXM_OF_TCP_DST	RBVE($k = 4$)
OXM_OF_UDP_SRC	RBVE($k = 4$)
OXM_OF_UDP_DST	RBVE($k = 4$)

5.2. 並列アーキテクチャ

提案手法のアーキテクチャは複数のフィールドのマッチングを並列に処理する。各フィールドがそれぞれのアルゴリズムでパケットマッチングを行う。また各アルゴリズムをパイプライン化し、一度の複数のパケットを処理できるようにする。

3.2 節で示したように OpenFlow1.3 のマッチフィールドには前提条件がある。そのため前提条件を満たしたフィールドのみマッチングを行う。使用しないフィールドのマッチングモジュールは、パイプラインの整合性を保つためダミーデータを入力に入れる。PPE においてそのデータは破棄される。

IPv4 と IPv6 に関するマッチフィールドは ETH_TYPE の値によってどちらを使用するか決まる。ETH_TYPE の値が 0x0800 であれば IPv4、0x86dd であれば IPv6 が選択される。

同様に TCP と UDP 関連のフィールドは、IP_PROTO の値が 6 であれば TCP、17 であれば UDP が選択される。

各フィールドはマッチング処理を行った後そのフィールドでマッチしたルールを示した bit vector を出力する。各 bit vector は Pipelined Priority Encorder に送られ、最終処理を行う。

5.3. StrideBV のパイプラインアーキテクチャ

StrideBV のパイプラインアーキテクチャ[10] を図 1 に示す。各ステージごとに前ステージの bit vector の結果と入力フィールドを入力とする。入力フィールドは該当する stride 部分をメモリ番地としてメモリアクセスし、stride 部分の bit vector を得る。この 2 つの bit vector を and 演算することによってこの stride でマッチするルールを抽出する。

5.4. RBVE のパイプラインアーキテクチャ

RBVE のパイプラインアーキテクチャ[11] を図 2 に示す。各ステージごとに入力フィールドに対して出力信号を決定する。最初のステージを除くすべてのステージでは出力信号と前ステージの出力信号を入力とした LUT(Look Up Table) で範囲のどこに位置しているか判定する。最後のステージではこのルールが入力フィールドにマッチするかどうかを調べる 1 ビットの信号を出力する。

このアーキテクチャでは 1 個のパイプラインにつき 1 個のルールのみ対応している。[12] ではルールごとに専用のパイプラインを用意するアーキテクチャを提案している。本研究でもこれに倣い、ルールごとに専用の RBVE パイプラインアーキテクチャを用意する。すべてのパイプラインの結果が得られたら出力である 1 ビットの信号を集めて bit vector を作成する。

5.5. Pipelined Priority Encorder

Pipelined Priority Encorder(PPE) はすべてのフィールドのマッチングを終えた後に使用するモジュールである。PPE はすべてのフィールドのマッチング結果である bit vector を集め、それらをすべて and 演算することで最終的なマッチングを行う。PPE から出力された bit vector で 1 になっているビットがすべてのフィールドにおいて入力パケットにマッチしたルールとなる。

6. 実装結果

今回は Altera 社製の FPGA である EP4CE115F29C7（114480 LEs）が搭載されている FPGA ボード DE2-115 での実装を想定している。シミュレータは ModelSim を使いこのアルゴリズムのスループットやメモリ消費量などを計測する必要がある。

今回の実験では TCP/UDP モジュールの実装を除いた環境でシミュレートした。その結果を 6 に示す。

表 7. シミュレーション結果 (N=32)

使用 LE ｓ	11226
使用レジスタ	6219
使用メモリ	281600

使用 LE ｓは全 LEs のうちの 10 %ほどとなり、回路規模は問題ないと思われる。

しかし、ここから想定できるメモリ使用量は表 12 のとおりである。

表 8. 想定するメモリ使用量

ルール数 N	メモリ使用量
64	563200
128	1126400
256	2252800
512	4505600
1024	9011200
DE2-115 最大値	3981312

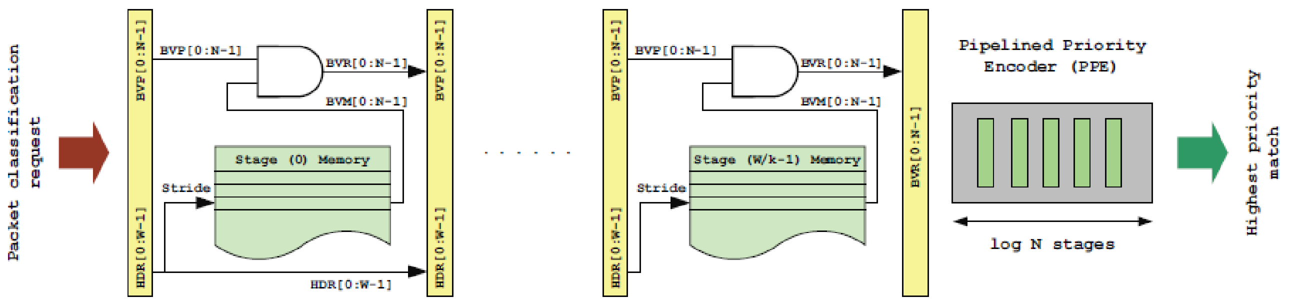

図 1. StrideBV のパイプラインアーキテクチャ[10]

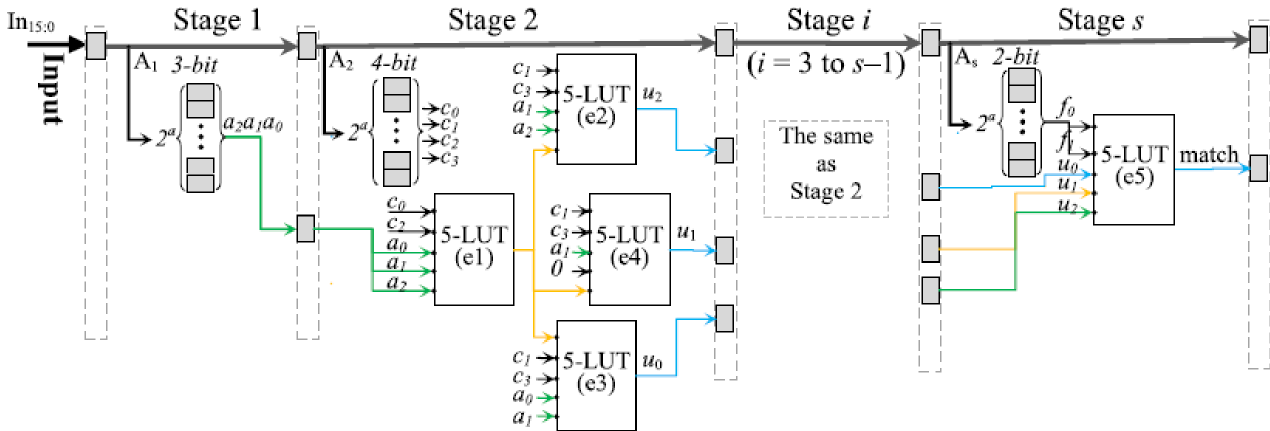

図 2. RBVE のパイプラインアーキテクチャ[11]

この通り DE2-115 では 256 のルール数でしか保持できないことが予想される。そのため今後はいかにメモリ消費量を抑えるかが課題となる。

7. まとめと今後の課題

本研究では OpenFlow1.3 で必須とされている 13 フィールドに対するパケット識別アルゴリズムを示した。今後このアルゴリズムをシミュレートしてスループットやメモリ消費量を見積もる必要がある。またこのアルゴリズムはパケットの分類だけを見ており、入ってきたパケットからヘッダ情報を抽出する部分やマッチしたルールに従ってパケットを処理する部分については触れられていない。そのため実際にこのアルゴリズムを使用して OpenFlow のネットワークを構築する際、それらも考慮する必要がある。パケット分類の点で見れば、40 フィールドすべてに対するパケット識別のアルゴリズムを考案し、いかにして性能を出すことを考える必要がある。その後、パケット識別だけでなくスイッチ全体の実装も考える必要がある。
スイッチ全体の実装においては以下の点を考慮する必要がある。

- コントローラと通信する際の OpenFlow プロトコルをどのように実現するか
- フローテーブルの更新をどのように行うか
- コントローラからの命令をどのように処理するか

そのため、スイッチの実装方法に関してはかなり複雑となり、今後様々なアプローチが登場するものと思われる。

参考文献

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “OpenFlow: Enabling innovation in campus networks,” SIGCOMM Comput. Commun. Rev., 38(2):6974, 2008.

[2] Ryu SDN Framework. https://osrg.github.io/ryu/index.html.

[3] Trema GitHub Pages. https://trema.github.io/trema/.

[4] Open vSwitch. http://openvswitch.org/.

[5] Lagopus switch. http://www.lagopus.org/.

[6] OpenFlow Switch Specification Version 1.0.0. http://archive.openflow.org/documents/openflow-spec-v1.0.0.pdf.

[7] Open Networking Foundation. https://www.opennetworking.org/.

[8] OpenFlow Switch Specification Version 1.3.0. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf.

[9] W. Jiang and V. K. Prasanna, “Field-Split Parallel Architecture for High Performance Multi-Match Packet Classification Using FPGAs,” In Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures (SPAA ’09). ACM, New York, NY, USA, 188-196.

[10] T. Ganegedara and V. K. Prasanna, ”StrideBV: Single chip 400G+ packet classification,” 2012 IEEE 13th International Conference on High Performance Switching and Routing, Belgrade, 2012, pp. 1-6.

[11] Y. K. Chang and C. S. Hsueh, ”Range-Enhanced Packet Classification Design on FPGA,” in IEEE Transactions on Emerging Topics in Computing, vol. 4, no. 2, pp. 214-224, April-June 2016.

[12] T. Ganegedara, W. Jiang and V. K. Prasanna, ”A Scalable and Modular Architecture for High-Performance Packet Classification,” in IEEE Transactions on Parallel and Distributed Systems, vol. 25, no. 5, pp. 1135-1144, May 2014.