JANOG53公開資料

JANOG53 Meeting In Hakata

ハードウェアオフロード技術を使って高性能で省電力なUPFを作ろうとした話（モバイルトラフィック増大への対策）

KDDI株式会社

木本 瑞希、立和名 隼人、藪原 穣

2024年1月18日

Tomorrow, Together KDDI

おもしろいほうの未来へ。 au

アジェンダ

- 自己紹介
- はじめに
  - KDDIグループのカーボンニュートラルへの取り組み
  - モバイルトラフィックとUPF
  - HWオフロード技術
- UPFの試作
  - Step 1 UPFの機能をP4でどこまで実現できるのか
  - Step 2 Tofino X アーキテクチャでUPF機能を実装してみる
  - Step 3 試作したUPFの性能を測ってみる
- UPFを試作してみての考察
- まとめ
- ディスカッションしたいこと

自己紹介

自己紹介

KDDI株式会社 ソフトウェア開発部 にて モバイルコアシステムの内製開発に従事

木本　瑞希
Mizuki Kimoto
mz-kimoto@kddi.com
博多にはauショップ研修以来7年ぶり

立和名　隼人
Hayato Tachiwana
ha-tachiwana@kddi.com
牧のうどんの無限増殖する肉うどんが好き

藪原　穣
Osamu Yabuhara
os-yabuhara@kddi.com
2017年までNECでWDM装置開発/拡販(1年ブラジル駐在)
1年半前まで基幹伝送網の技術担当→現UPF開発PO
なので実はTS29.244よりG.709あたりが得意

はじめに

KDDIグループのカーボンニュートラルへの取組

KDDIは2030年度にCO_2排出量実質ゼロを目指す*

CDP DISCLOSURE INSIGHT ACTION A LIST 2022 CLIMATE
RE100 CLIMATE GROUP | CDP

* KDDIニュースリリース　https://news.kddi.com/kddi/corporate/newsrelease/2022/04/07/5984.html
* KDDIグループは2050年度カーボンニュートラル目標
* Scope1（直接排出）および Scope2（間接排出）が対象。Scope1 の削減には、各種オフセット手法も活用します。

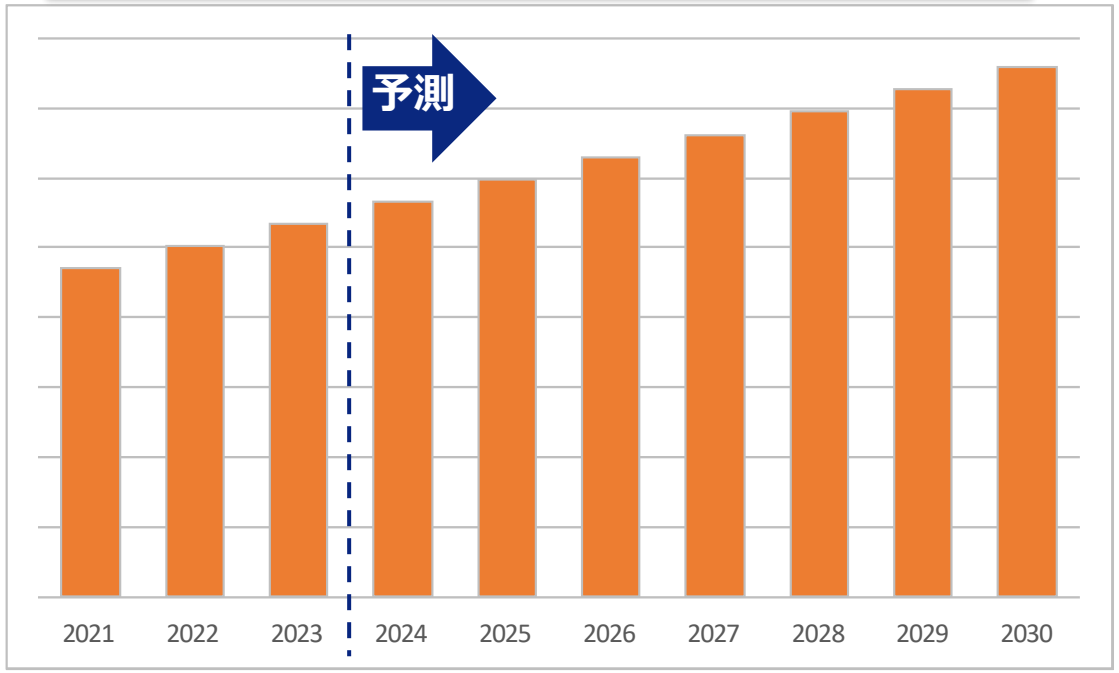
Tomorrow, Together KDDI　おもしろいほうの未来へ。au

モバイルトラフィック量増大と電力消費量/CO_2排出量の増加

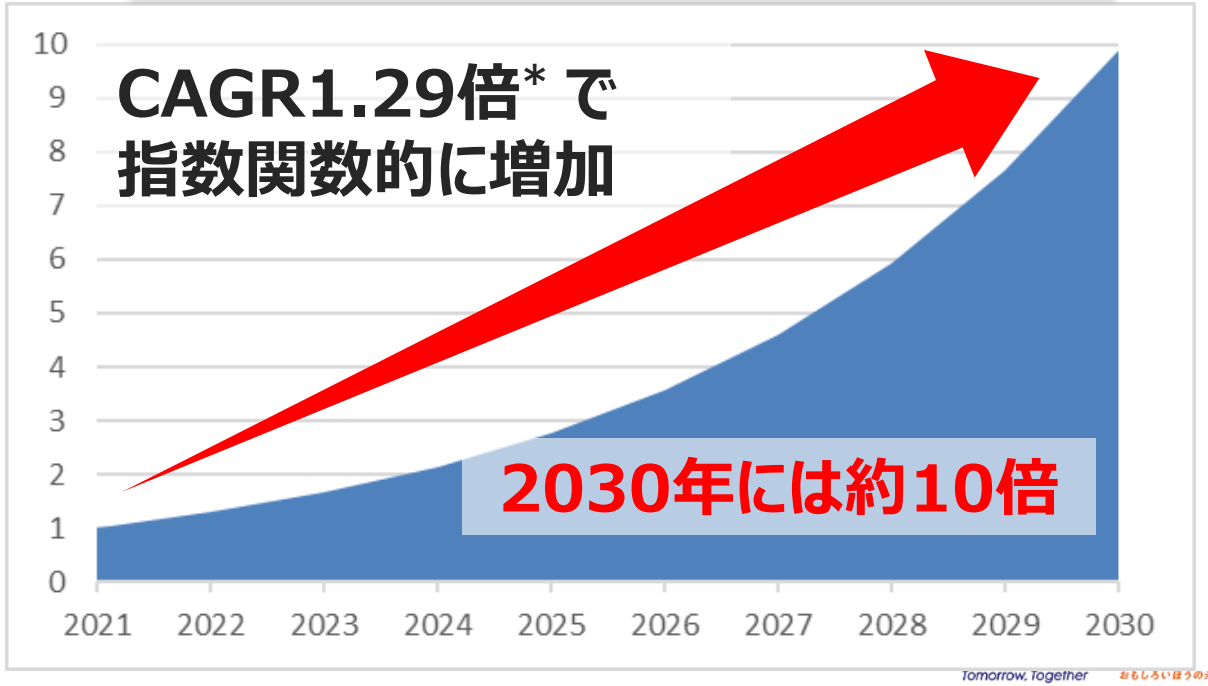
2030年には今の10倍の設備が必要　電力消費量・CO_2排出量も10倍！

- **トラフィック量は指数関数的に増加する傾向**
- CPUの1コア当たりの性能向上はほぼ横ばいで、トラフィック需要に追い付かなくなる可能性あり
- モバイルコア増設にはより多くのサーバが必要になる
  - ➡ **サーバ増設に伴う電力消費量、CO_2排出量の増加が課題**

モバイル端末の稼働数推移



トラフィック需要予測



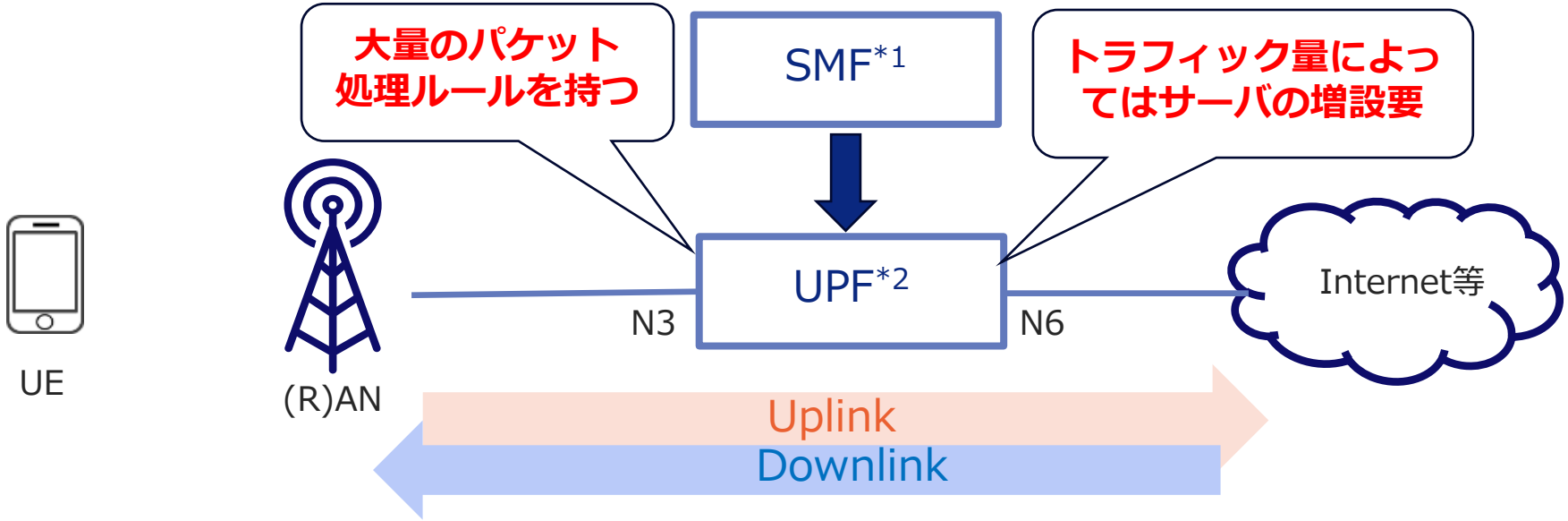
* 我が国のインターネットにおけるトラヒックの集計・試算（2021年5月時点の集計結果の公表）
https://www.soumu.go.jp/menu_news/s-news/01kiban04_02000189.html

UPFと電力消費量の関係

ユーザトラフィックを処理するため、需要に応じてUPF増設が必要
UPFの台数が増えることで、モバイルシステムの電力消費量が増大する

■5GCにおけるUPFの役割

- ユーザデータ（Internetトラフィック(Web/動画等）、音声通話など）を転送する
- モバイルシステム特有のカプセリングプロトコル(GTPv1)に対応し、課金処理、帯域制御、優先制御、パケットの一時的な保持などを行う
- セッション(≒端末)単位に、SMFが指示するパケット処理ルールを記憶する
- 収容するセッション数は、数十万～数百万規模。数Gbps～数百Gbpsのトラフィックを処理



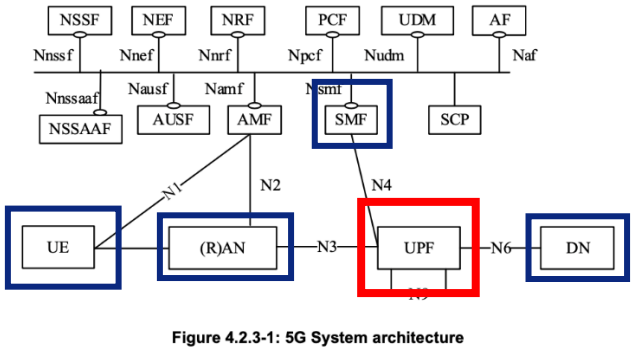


Figure 4.2.3-1: 5G System architecture

*1 Session Management Function
5Gコアノードの一つで、他のノードと連携し、UPFとパケット処理に関するルールをやりとりする

*2 User Plane Function
5Gコアノードの一つで、ユーザデータの送受信を制御し、トラフィックのルーティングなどを行う

*3 ETSI TS 123 501 v16.7.0

HWオフロード技術を活用したネットワーキングデバイスの登場

FPGA、プログラマブルASIC (Tofino*1) を搭載したWhite Box SwitchやSmartNICなど、P4言語を使用したロジック開発が可能なデバイスが登場

*1 インテル® Tofino™　https://www.intel.co.jp/content/www/jp/ja/products/details/network-io/intelligent-fabric-processors/tofino.html

■ **高性能なパケット転送処理ロジックを高級言語で記述することが可能**

■ ただし、特性に合わせて使い分ける必要がある

- **FPGAは外部メモリを利用できるが、単体での帯域幅は大きくない**
- **インテル® Tofino™はメモリサイズが小さく、大規模なテーブル参照や大量のPacket Bufferingが難しい**

プログラマブルなデバイス

インテル/Silicom N5010 *2

Edgecore Networks DCS801(WEDGE100BF-32QS) *3

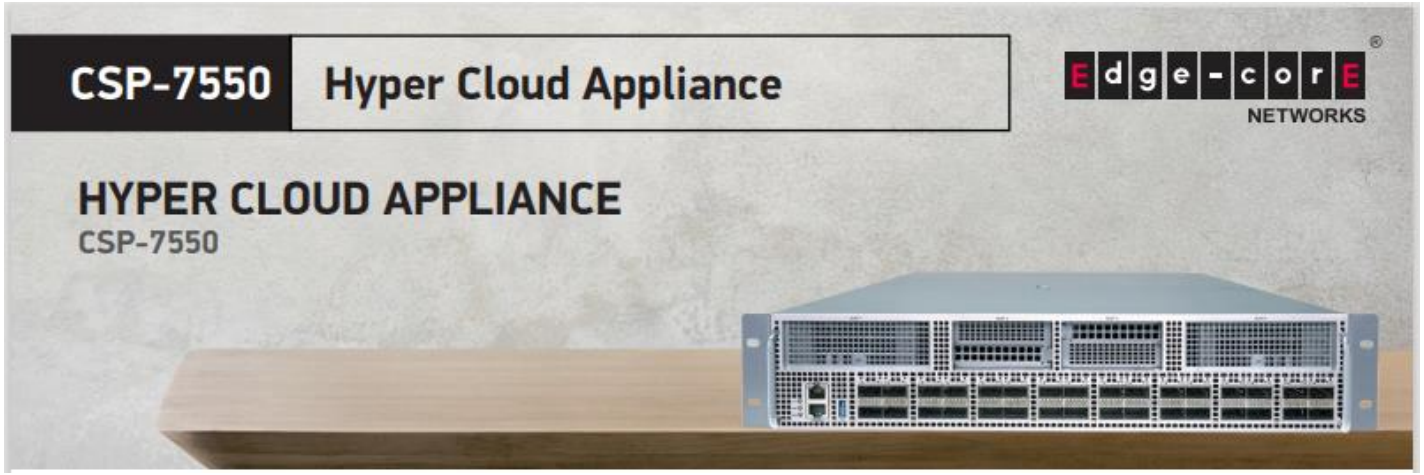
適用機能	FPGA	インテル® Tofino™	CPU(参考)
大容量トラフィック転送	△	○	×
大規模なテーブル参照	○	△	○
Packet Buffering	○	△	○

*2 Silicom FPGA SmartNIC N501x Series　https://www.silicom-usa.com/pr/4g-5g-products/4g-5g-adapters/silicom-fpga-smartnic-n5010_series/
*3 Edgecore Networks DCS801　https://www.edge-core.com/jp/productsInfo.php?cls=174&cls2=190&cls3=638&id=1032

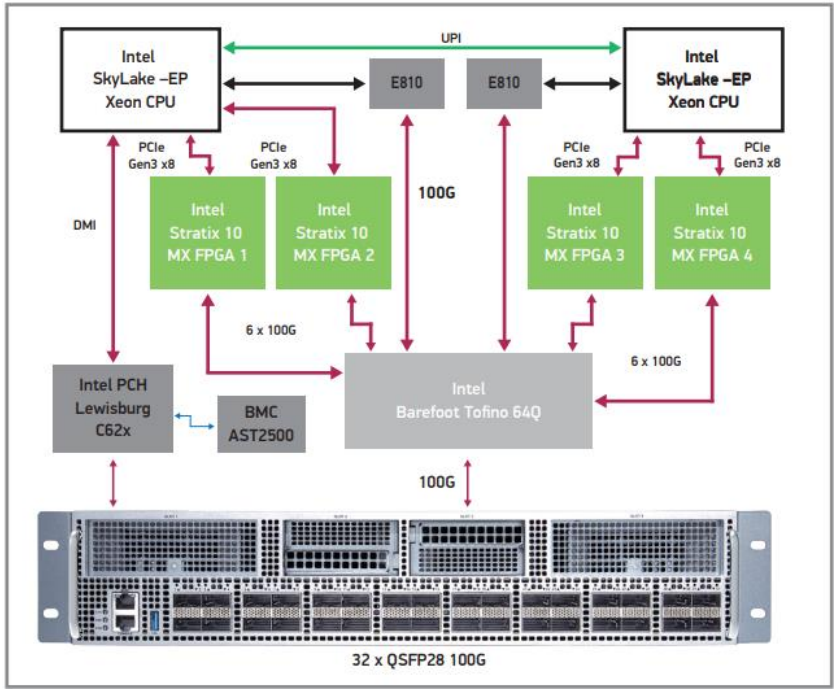
HWオフロード技術を使ってUPFを試作してみよう

インテル® Tofino™とFPGAを組み合わせたプログラマブルスイッチでは、
多数のセッション情報をFPGAに保持させることが可能で、複数のIAサーバに
収容していたトラフィックをプログラマブルスイッチ1台に集約できる

- **P4言語でアプリ開発のように回路設計が出来る良い感じのシステムが登場**



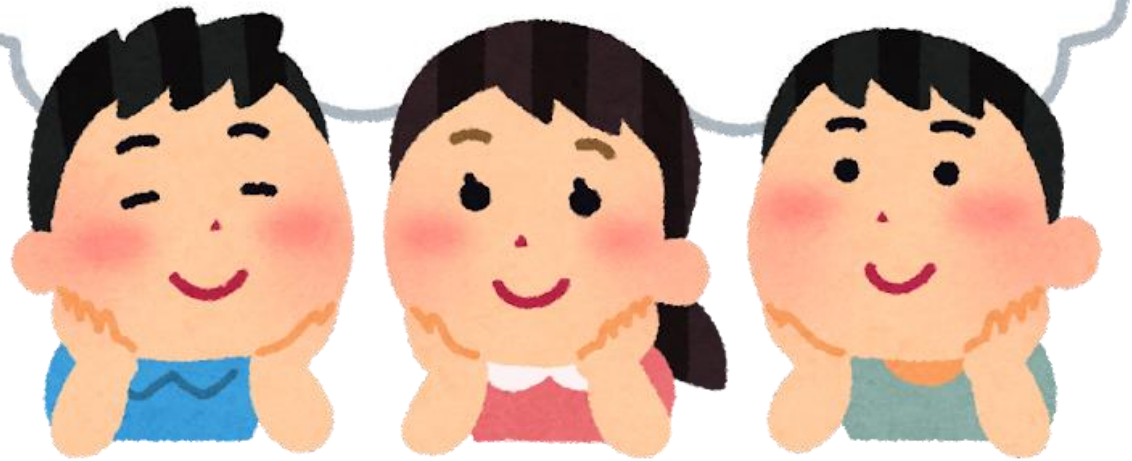
* CSP-7550 データシート　https://www.edge-core.com/_upload/images/2023-073-CSP-7550_DS_R05_20230526.pdf



UPFの電力消費量の増大を抑えられる可能性が見えてきた
➡ 試作開発、性能評価を実施してみよう！

UPFの試作体制

- 世の中に要素技術はあるけど、試作してくれるベンダさんを探すと、また時間がかかりそう
- イケてる技術かどうか確かめるだけなら、内製でサクッと確認してみればいいんじゃない？



アプリ開発者の領域拡大を狙って
とりあえず内製部隊* で

あわよくば商用化も…

* 純ソフトウェア開発者の集まりで、FPGAは素人。P4は触れたことがある人がいる。ぐらいの集団

UPFの試作

UPFの試作目的

HWオフロード技術を用いたUPFが
2030年度のCO_2削減目標達成に貢献できるかどうかを確認する

基本機能の実現性 ✖ 実用性の観点

■ HW処理で何ができる？

- GTPv1 Header Encap/Decap
- C-Plane/U-Planeの連携
- Idle-Timeout機能の実現
- 課金情報の保持＆レポーティング
- Packet Buffering
- IP Fragment/Reassemble

■ パフォーマンス

- エコになっても、処理が遅ければ要らない（耐えられない）
- 消費電力はどれぐらい下がる？

■ 商用化にあたって

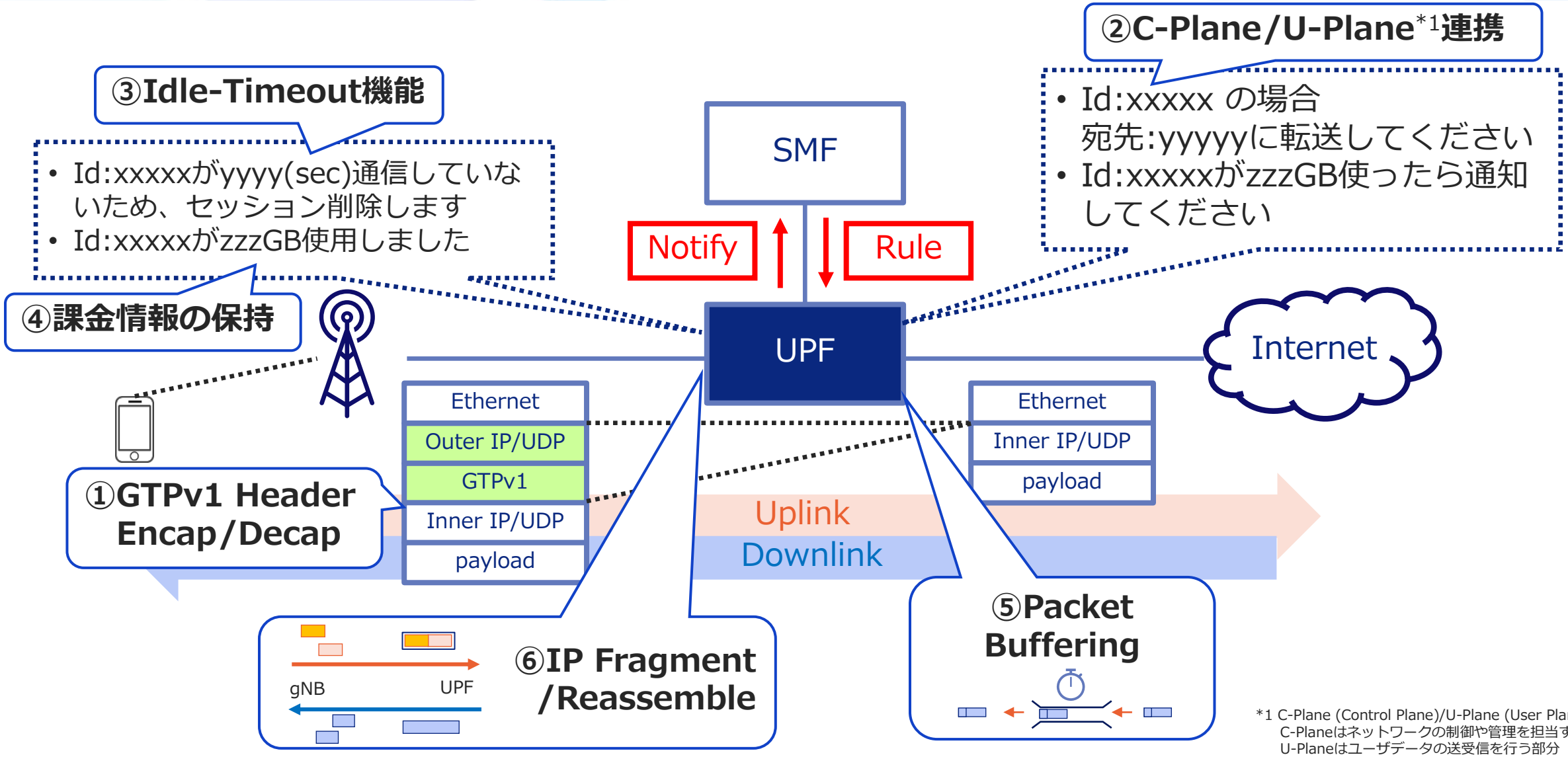
- （定性的に）内製での開発継続性、運用機能の実装容易性、保守対応の難しさなどはある？

UPF試作の流れ

STEP 1	UPFの機能はP4でどこまで実現できるのか	• UPFでやるべきことをP4言語で記述できるのか確認してみる
STEP 2	Tofino X アーキテクチャでUPF機能を実装してみる	• Tofino X アーキテクチャって何？ • UPF機能をどうやって実機に実装したか
STEP 3	試作したUPFの性能を測ってみる	• UPFとしてやりたかったことができたか？ • カーボンニュートラル実現に貢献できそうか？

UPFの試作　Step 1
UPFの機能はP4でどこまで実現できるのか

Step 1：UPFの機能/動作説明

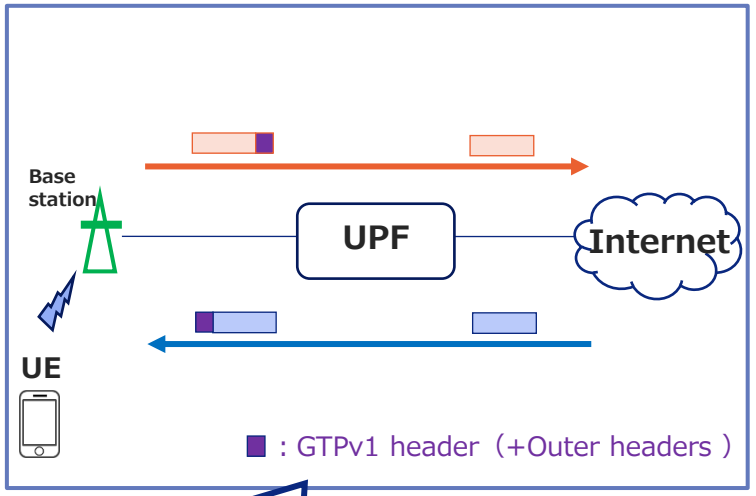


*1 C-Plane (Control Plane)/U-Plane (User Plane)
C-Planeはネットワークの制御や管理を担当する部分、
U-Planeはユーザデータの送受信を行う部分

Step 1：UPFの機能をP4で実現できるか①

当日投影なし

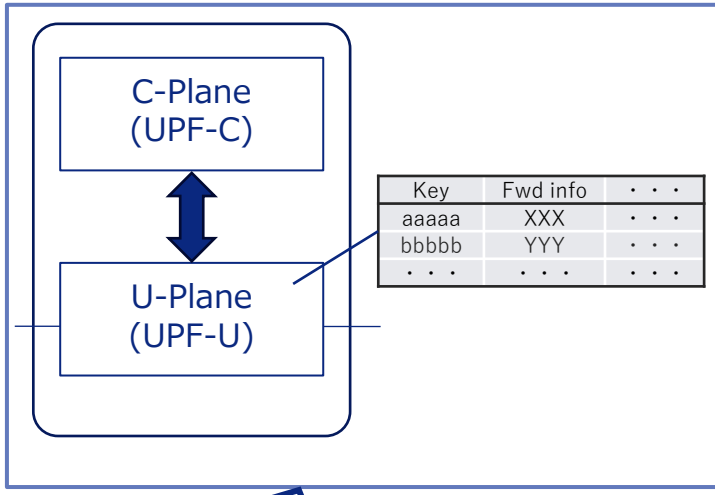
①GTPv1 Header Encap/Decap



・P4のTable機能にGTPv1ヘッダのEncap/Decapに必要なユーザ毎の情報を載せればできそう
・どのくらいエントリ登録できるのかな

※Table機能：
Routing Tableのように、入ってきたパケットのヘッダ情報等をKeyにしてTable内をLookupし、ヒットしたパケットのヘッダ情報を書き換える等のアクションを指定できる機能

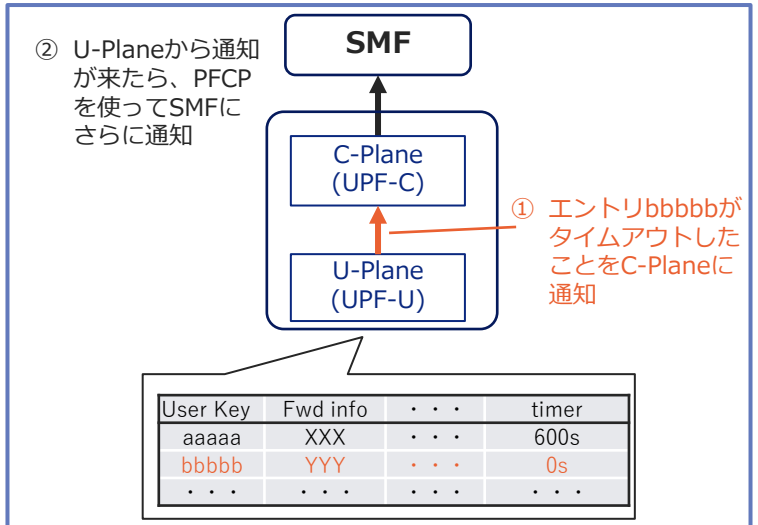
②C-Plane/U-Plane連携



・C-Planeから受け取った情報をU-Plane処理用のTableに反映させるにはP4 Runtimeを使えば良さそう
・P4 Runtimeの使い勝手は？
・更新レートはどれぐらい？

※P4 Runtime：
P4のTable機能を操作するためのAPI
※UPF-C：
UPFの中でも特に制御や管理を担当する部分
※UPF-U：
UPFの中でもユーザデータの転送に特化した部分

③Idle-Timeout機能



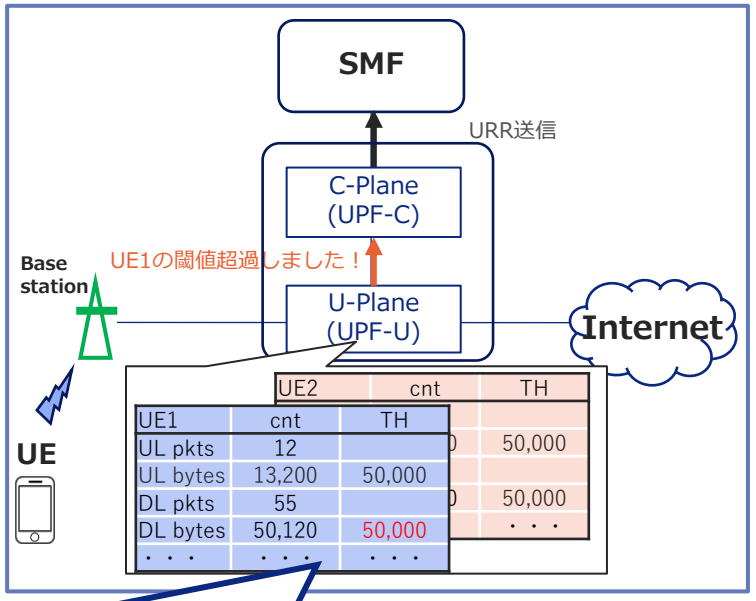
P4のSupport timeoutを使えば実現できそう…

※Support timeout：
Tableエントリが一定時間検索されない場合、C-Planeに通知するTableのオプション機能

Step 1：UPFの機能をP4で実現できるか②

当日投影なし

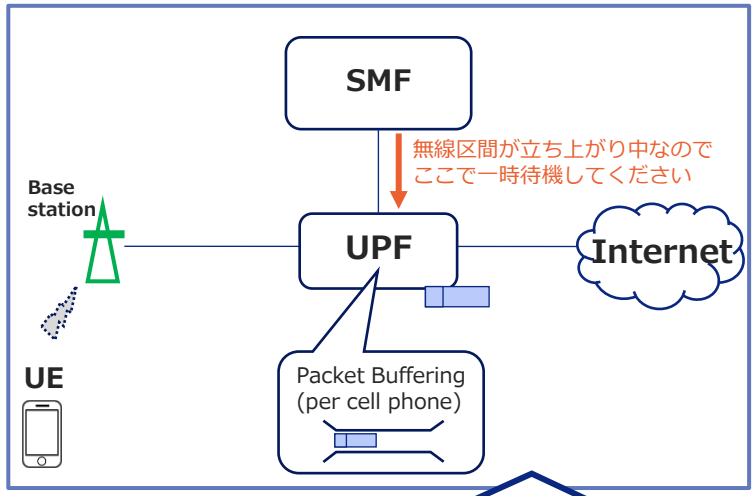
④課金情報の保持



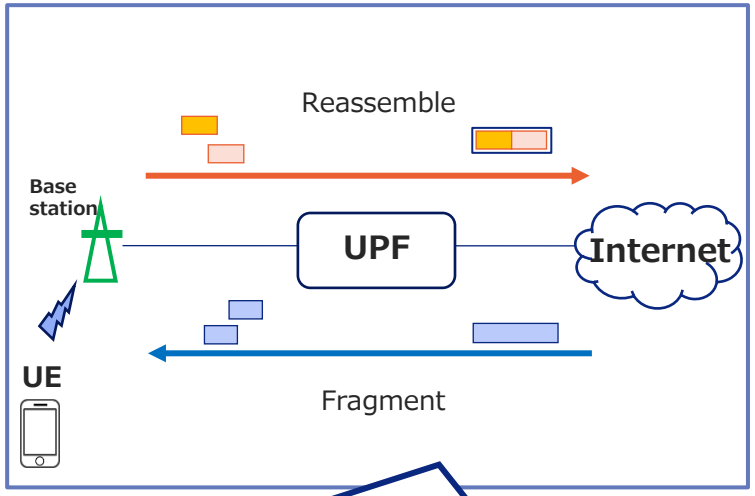
・ユーザ毎の統計情報を持つ必要があるけど、P4のCounter、Registerの使い勝手は. . .?
・閾値超過をリアルタイムに感知するにはRegisterが必要そう？

※Counter：Packet数とPacket長のインクリメントが可能。読み取りはC-Plane側のみ可能
※Register：U-plane側で現在値を読み取り、加算/減算/初期化などの処理ができる

⑤Packet Buffering



⑥IP Fragment/Reassemble



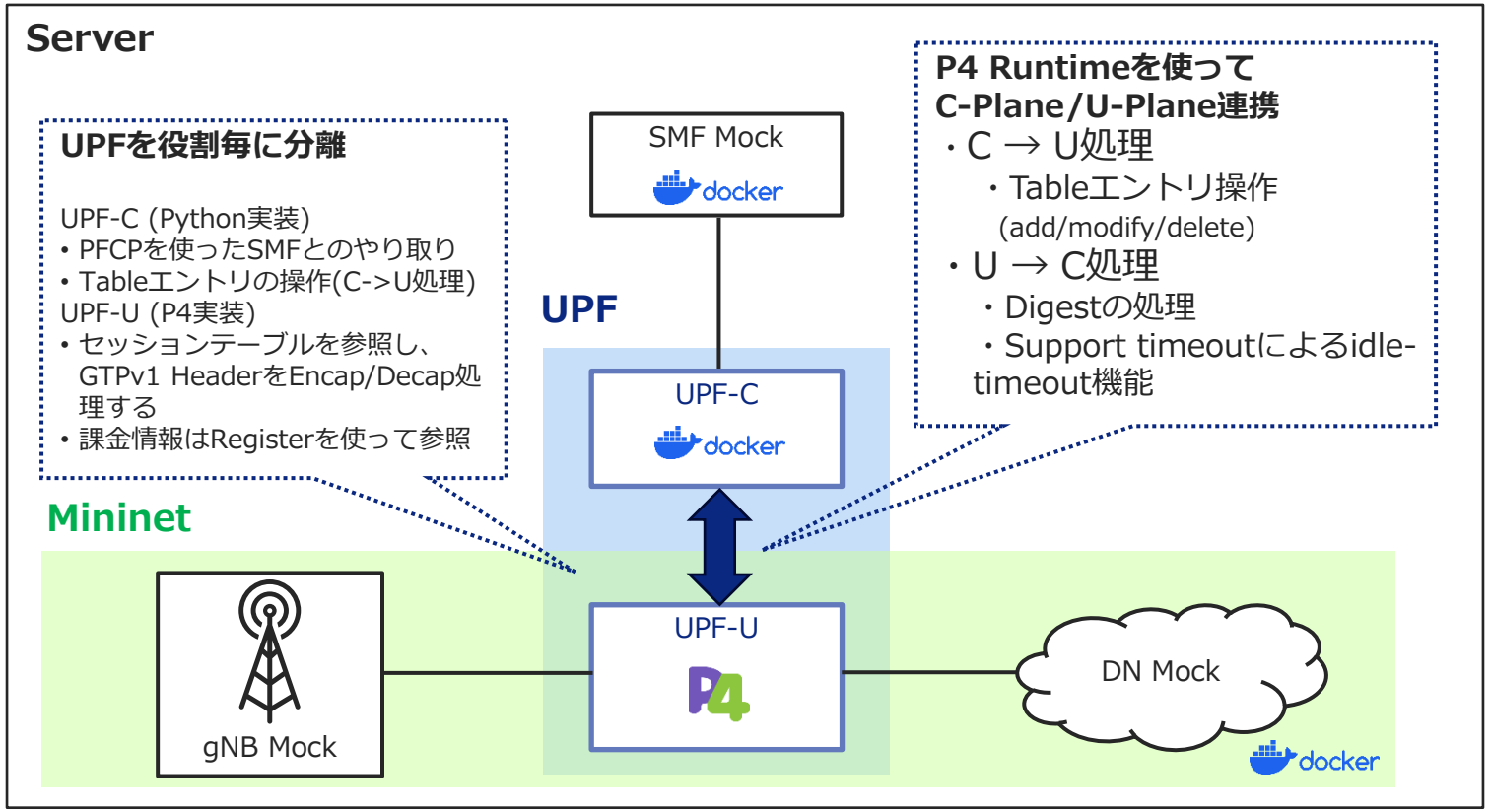
P4でこの振る舞いを記述することは難しそう

条件に当てはまったらCPU処理に回すしかない？

Step 1：簡易UPFを作りました

P4の学習を兼ねて仮想環境で動作する簡易UPFを実装し、P4でできること、できないことを改めて確認

P4はv1モデルを利用し、Mininet*1とDockerで環境を構成した



*1 https://mininet.org/

結果

できたこと

- GTPv1 Header Encap/Decap
- C-Plane/U-Plane連携
- Idle-timeout機能
- 課金情報の保持

できなかったこと（実装しなかったこと）

- 一時的にパケットを保持する処理
  - Packet Buffering
  - Reassemble
- 新しくパケットを作成する処理
  - IP Fragment

※これらはCPU処理すれば実現できそう

UPFの試作　Step 2
Tofino X アーキテクチャでUPF機能を実装してみる

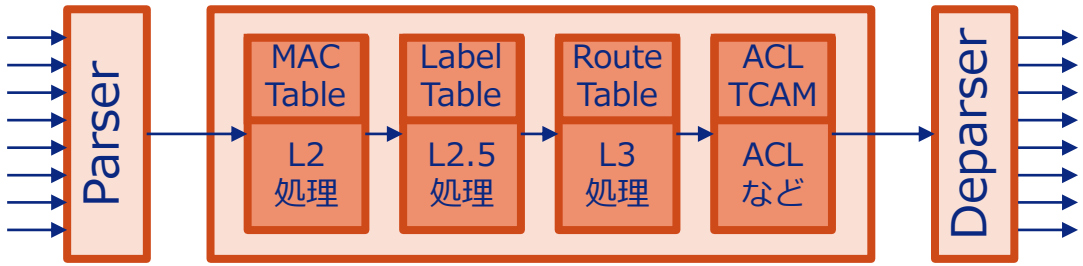
Step 2：インテル® Tofino™について

パケット処理ロジックをP4言語で記述できるスイッチASIC

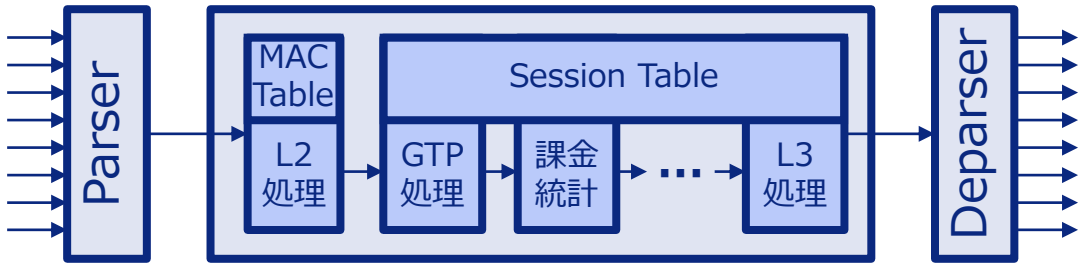
- インテル(旧Barefoot社)が開発したプログラマブルASIC
- 現時点で、インテル® Tofino™（本発表で利用）、機能強化版のインテル® Tofino™ 2の2種がリリースされている
  - 残念ながら、後継チップの開発は中止された

従来のASICとインテル® Tofino™の違い

従来：全ての処理/順番とテーブルサイズが固定

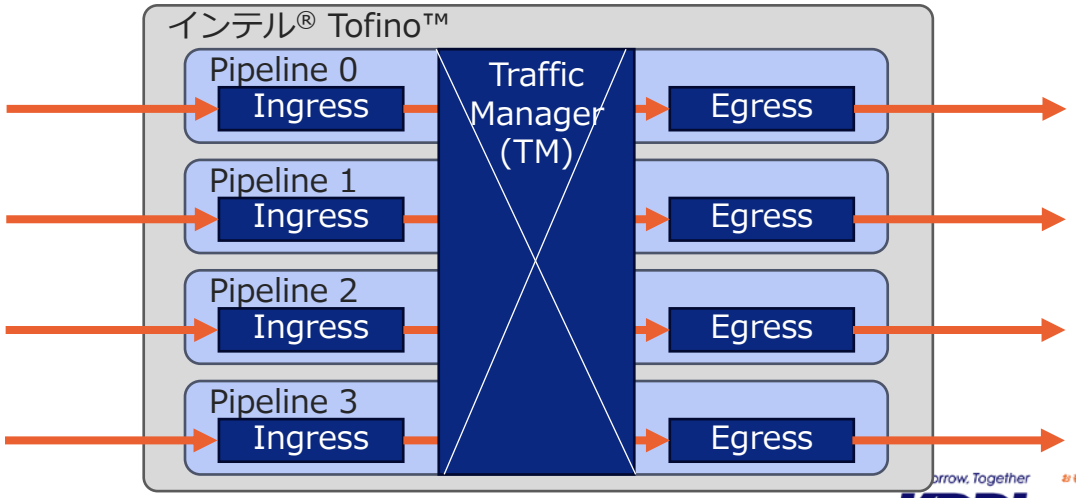


インテル® Tofino™：必要な処理を自分で定義できる



インテル® Tofino™の内部アーキテクチャ

- 4つのパイプラインとそれを接続するスイッチ(Traffic Manager)で構成される
- Ingress/Egressそれぞれ別々に処理を定義



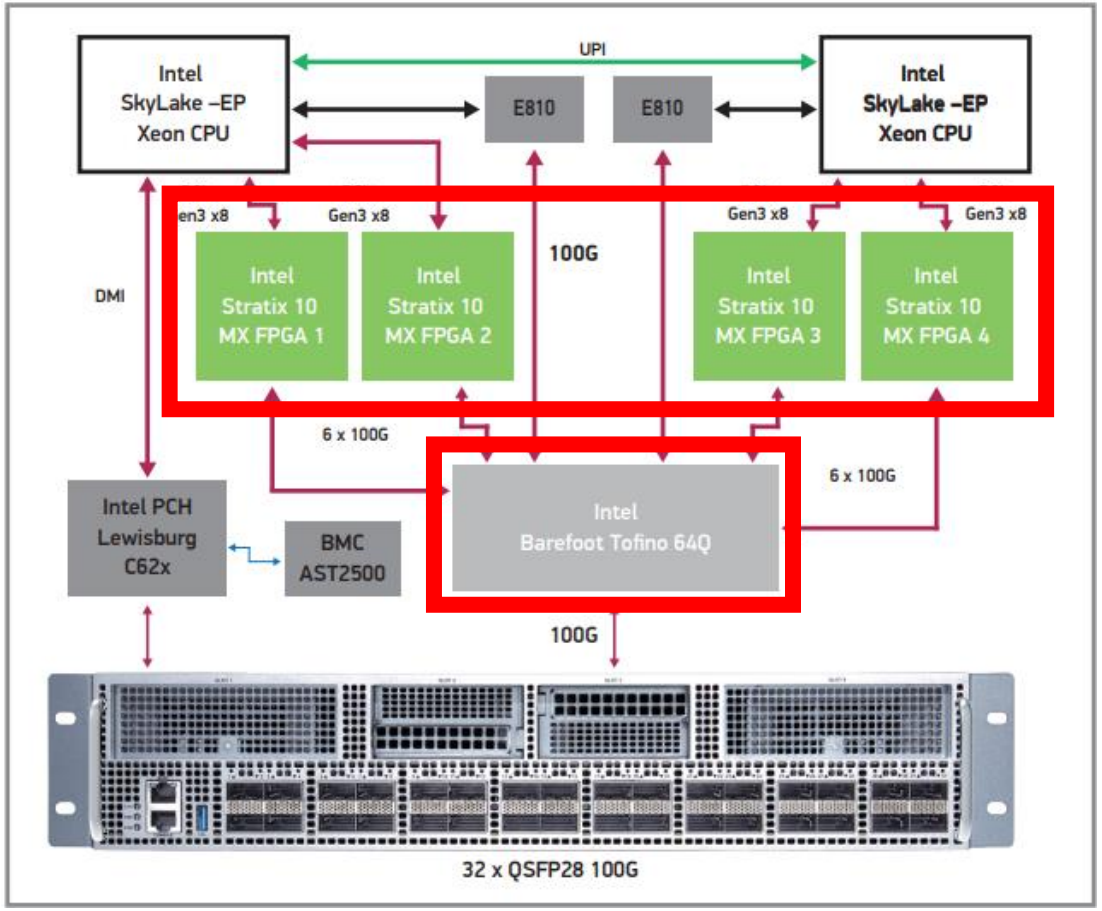
Step 2：Tofino X アーキテクチャについて

FPGA SoCとインテル® Tofino™を組み合わせ、大規模テーブル検索などの機能拡張を実現するアーキテクチャ

テーブル検索の違い

- インテル® Tofino™はTCAMを使用して、テーブルのlookupを高速処理する
  - TCAMサイズ：**24.8Mbits ≒ 3.1MB**
  - サイズが小さく、大規模なテーブルを持つことは困難

- FPGA SoCにはDMAがあり、HBM2などの外部メモリにテーブルを格納して使用できる
  - HBM2サイズ：**8GB～**



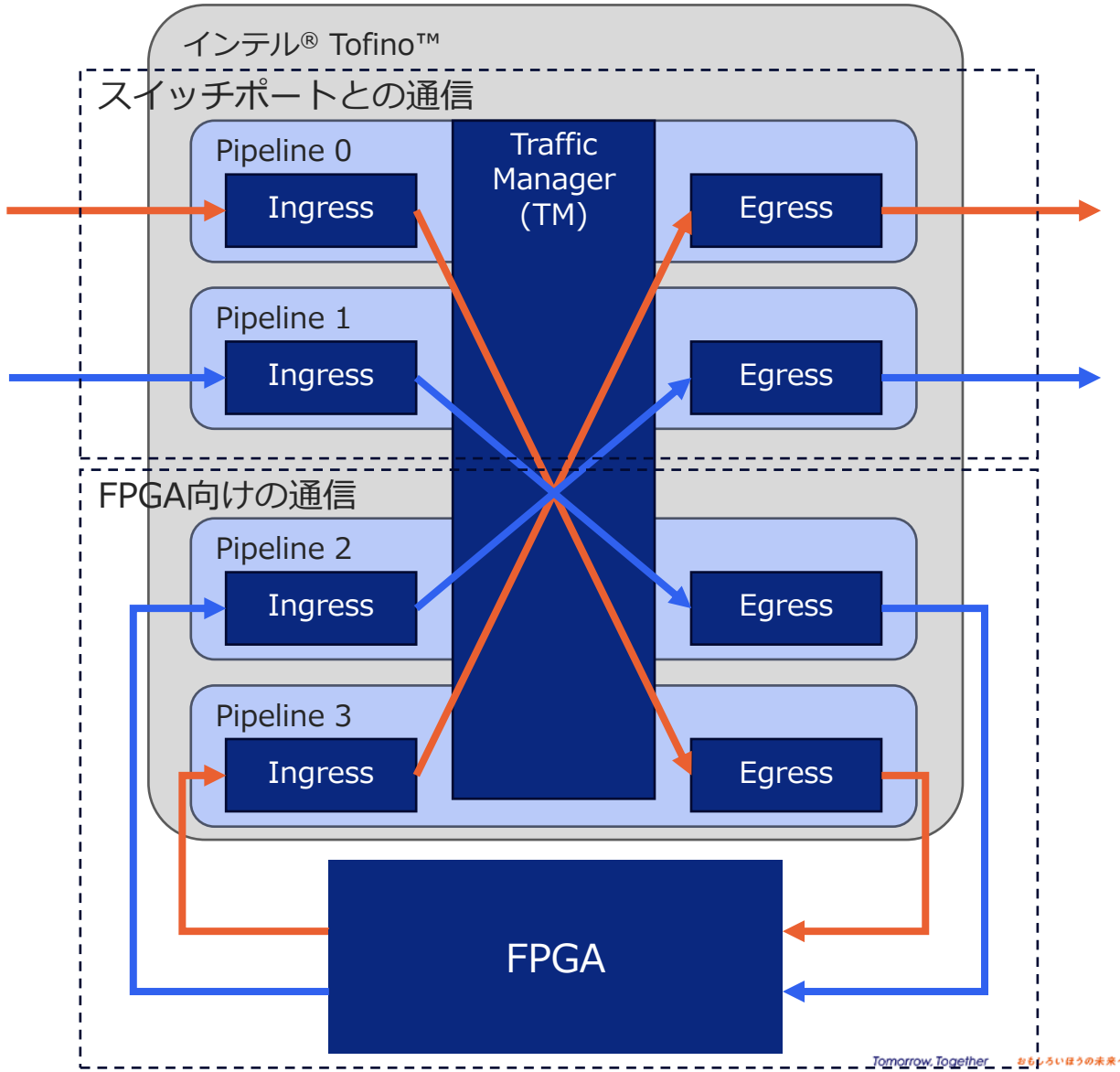
* CSP-7550 データシート　https://www.edge-core.com/_upload/images/2023-073-CSP-7550_DS_R05_20230526.pdf

Step 2：CSP-7550について

Tofino X アーキテクチャを採用したEdgecore社のServer Switch

■ CSP-7550
- ● Tofino X アーキテクチャを採用
  - インテル® Tofino™とFPGAのポート間を接続
    - 1FPGA当たり6ポート接続し、最大600Gbpsまで処理可能（使用するIPコアに依存する）
  - FPGAを最大4台まで搭載可能
  - Pipeline 0、1はスイッチポートとの通信を処理
  - Pipeline 2、3はFPGA向けの通信を処理
  - Traffic Manager (TM)によって、Pipelineを跨ったIngress/Egress間のパケット転送を実施
- ● 開発言語
  - インテル® Tofino™ ：P4言語
  - FPGA ：HDL*（ハードウェア記述言語）

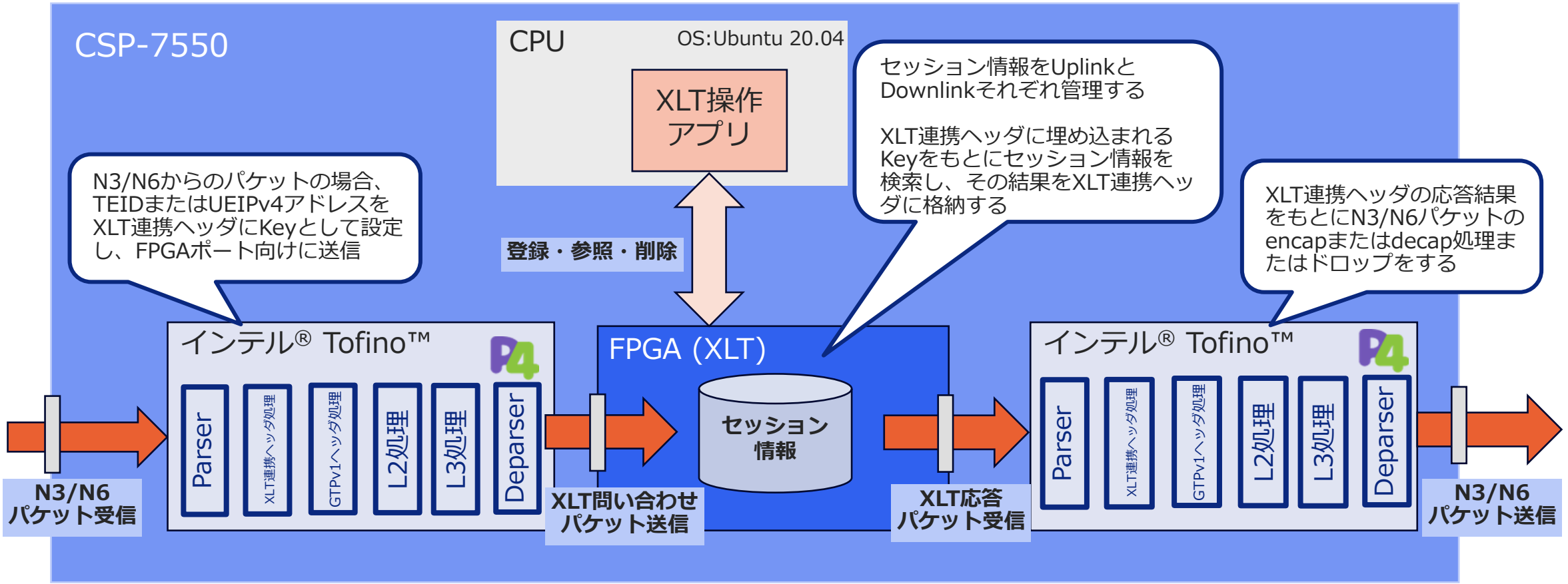
    * 今回はインテル提供のIPコアを利用
- ● 性能
  - **QSFP28(100GbE)×32**のポートを備えており、3.2Tbpsのトラフィックを処理することが可能（インテル® Tofino™単体時）



Step 2：試作UPFの概要

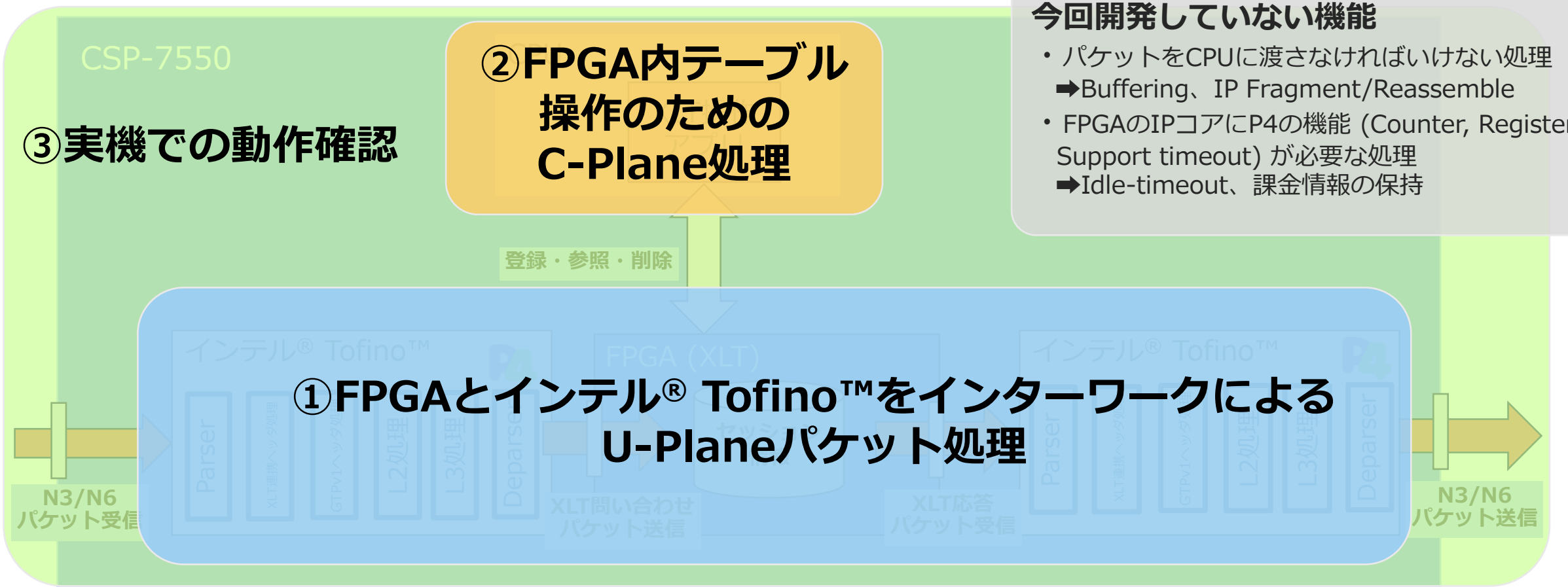
セッション情報をeXtra Large Table (XLT)*1 で管理
N3/N6パケットはXLTを経由し、セッション情報をもとにパケット転送

*1 インテルより提供を受けた大規模テーブル検索処理を実現するFPGA IPコア。2022年度時点ではβ版のXLTを利用



Step 2：試作UPFの概要

以降、試作したUPF機能を3パートに分けて紹介していく
今回は一部取り組めなかった機能もある

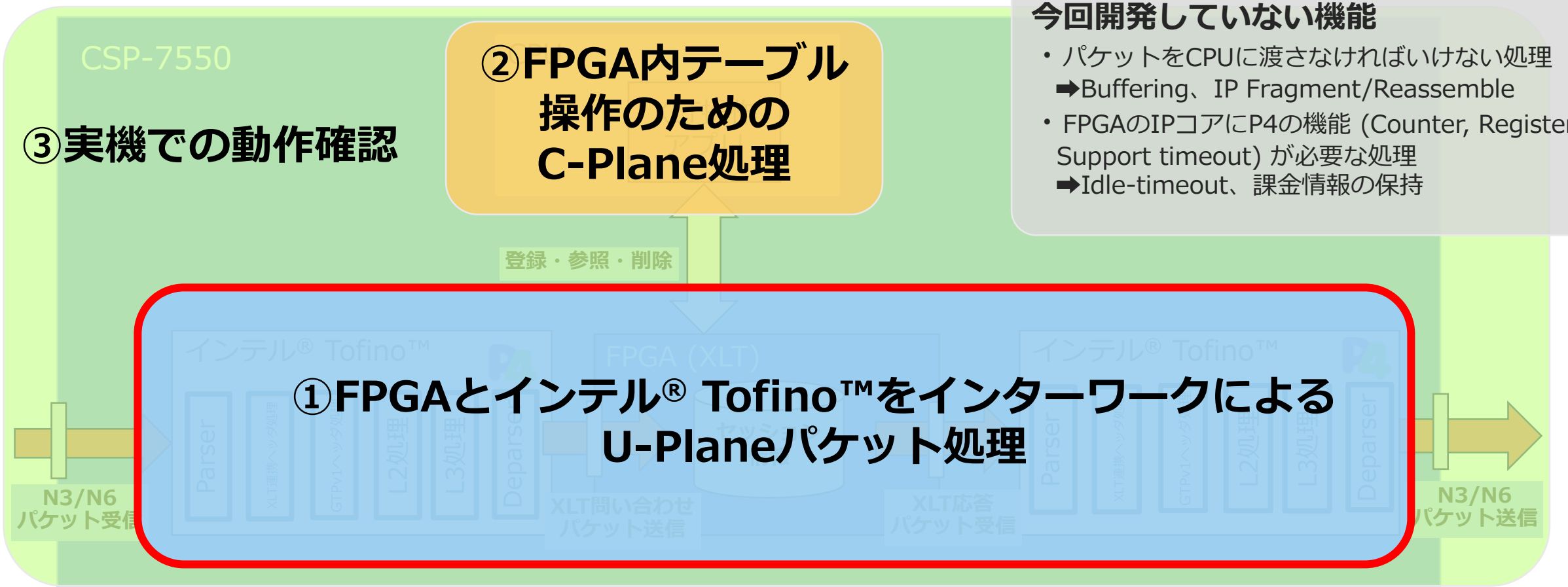


今回開発していない機能

- パケットをCPUに渡さなければいけない処理
  ➡Buffering、IP Fragment/Reassemble
- FPGAのIPコアにP4の機能 (Counter, Register、Support timeout) が必要な処理
  ➡Idle-timeout、課金情報の保持

Step 2：試作UPFの概要

以降、試作したUPF機能を3パートに分けて紹介していく
今回は一部取り組めなかった機能もある



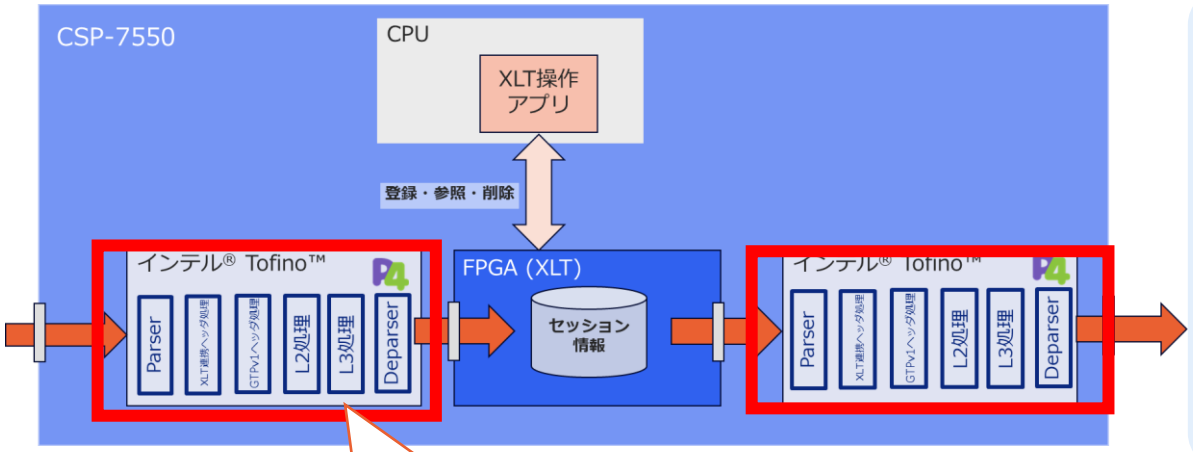
今回開発していない機能

- パケットをCPUに渡さなければいけない処理
  ➡Buffering、IP Fragment/Reassemble
- FPGAのIPコアにP4の機能 (Counter, Register、Support timeout) が必要な処理
  ➡Idle-timeout、課金情報の保持

Step 2：パート①　U-Planeパケット処理の開発方針

インテル® Tofino™用の開発ツール「インテル® P4 Studio*1」がリファレンス実装として提供している「switch.p4」を改造する

*1 インテル® P4 Studio　https://www.intel.co.jp/content/www/jp/ja/products/details/network-io/intelligent-fabric-processors/p4-studio.html

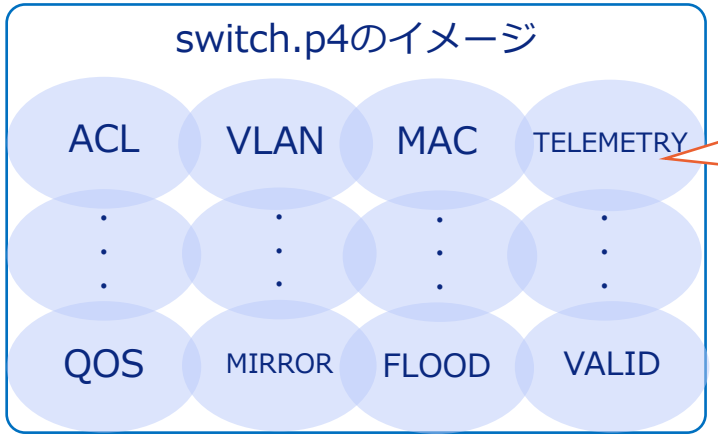


- インテル® Tofino™上で動作実績がある
- SONiC*2を使用してC/U-Plane連携ができる
- 基本的なSwitch機能（L2/L3）が存在する

*2 SONiC　https://sonicfoundation.dev/

「インテル® P4 Studio」が提供する
リファレンス実装の「switch.p4」

- 簡易UPFと同じくフルスクラッチで作成する？
- リファレンス実装を改造する？

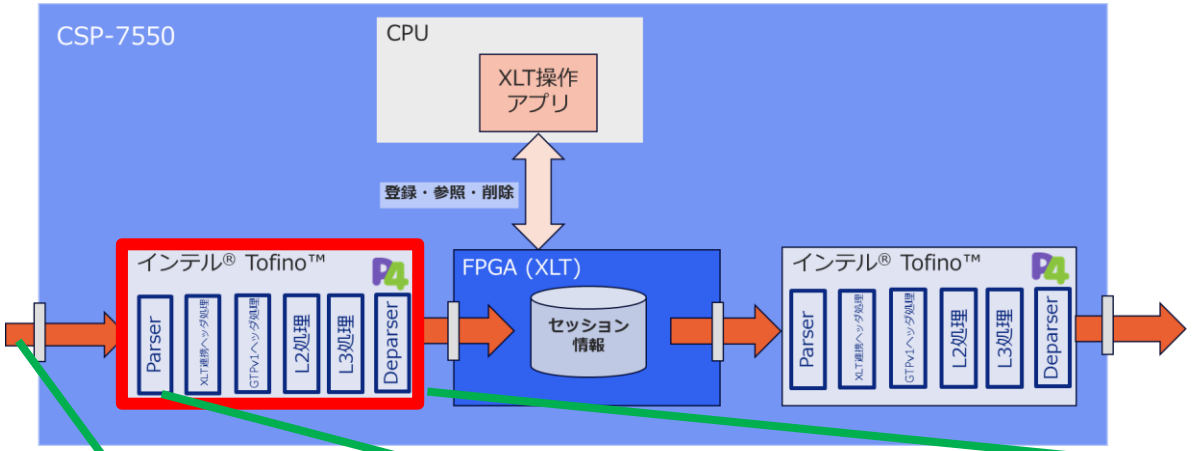


基本的なSwitch機能 (L2/L3)

switch.p4を改造する！

Step 2：パート① U-Planeパケット処理の流れ

インテル® Tofino™(P4) - FPGA間では特殊ヘッダを使用して情報を検索

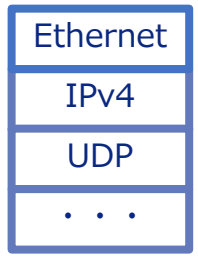


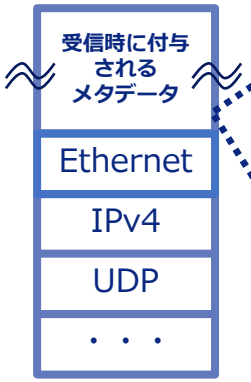
フロントポートから流入したパケットをXLT*1に渡すまで

*1 2022年度時点ではβ版のXLTを利用

1. 流入パケットの解析
2. UPFのどのインターフェースを識別して、
   専用のメタデータに情報を付与する
3. XLTヘッダ (検索Key) をセット

フロントポートに到達するパケット

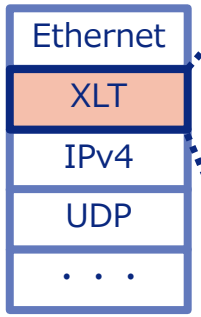


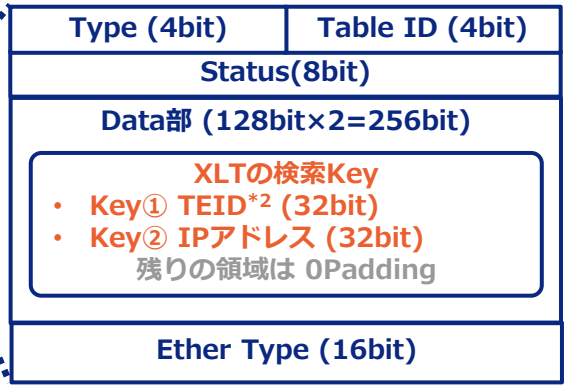


UPFのインターフェース情報

- **Uplinkの場合　：N3**
- **Downlinkの場合：N6**
- **SMFからの場合　：N4**

後続のロジック分岐に使用
- 内部でパケットの無限ループ防止
- 特殊ヘッダの付与時の条件に使用　等

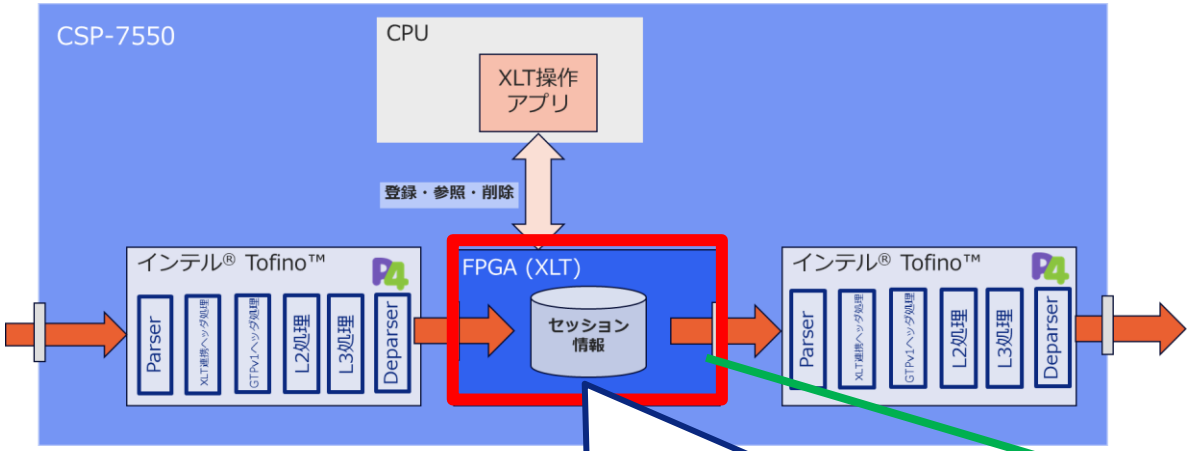




*2 Tunnel Endpoint Identifier, gNB, UPFでセッションを識別するためのID

Step 2：パート① U-Planeパケット処理の流れ

FPGAの検索結果は特殊ヘッダに格納される



XLT*1からの情報取得の流れ

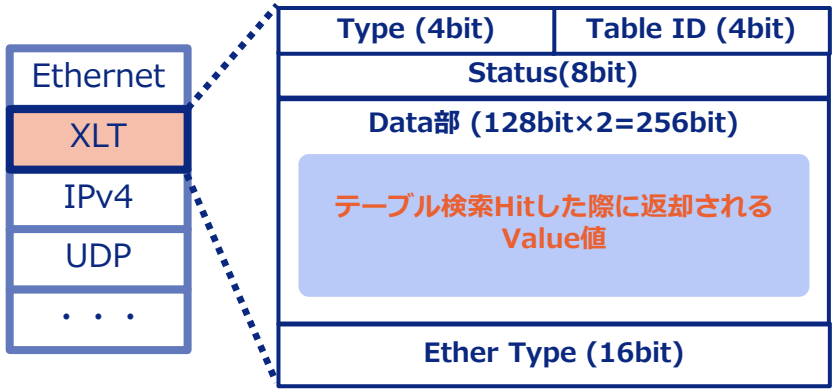
*1 2022年度時点ではβ版のXLTを利用

1. XLTヘッダ（検索Key）をセット
2. テーブル検索＆結果を特殊ヘッダ(XLTヘッダ)に格納
   - Table ID
   - Status (hit or miss)
   - Data部にValue値

FPGA（XLT）

TEID	IPアドレス	Value1	Value2	・・・	ValueN
aaaaa	XXX	FORW	10.20.30.40	・・・	
bbbbb	YYY	DROP	・・・	・・・	
・・・	・・・	・・・	・・・	・・・	

Key情報をもとにExactMatchでレコード検索

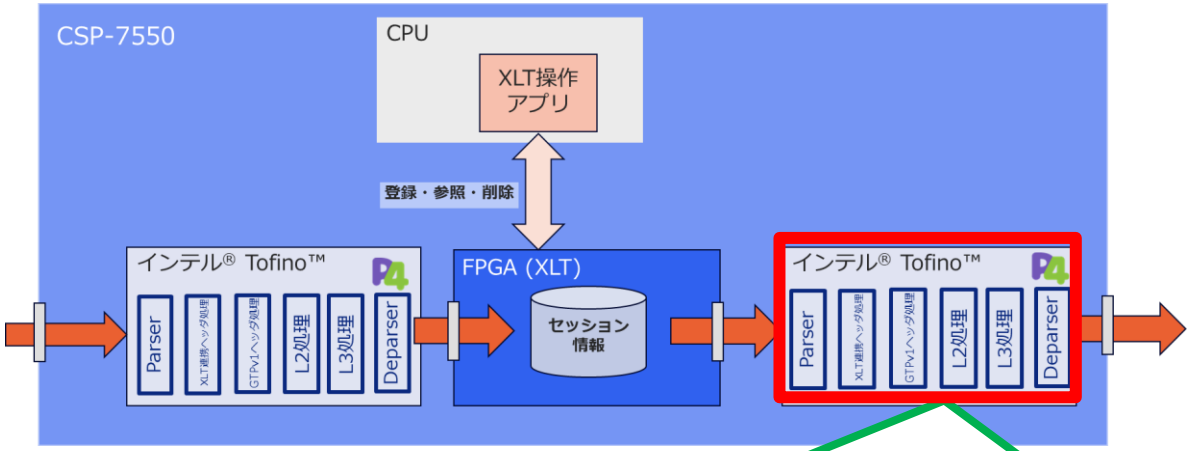


Value値で返される情報

- 宛先IPアドレス
- 宛先tunnel情報
- パケット処理情報（転送/破棄　等）
- Encap/Decap時のプロトコル
  スタック情報

Step 2：パート① U-Planeパケット処理の流れ

検索・取得結果を基にパケットを再構成、また外部へ放出

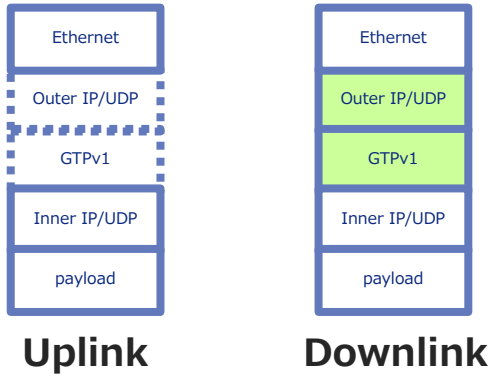


XLT*1から戻ってきた後の処理

*1 2022年度時点ではβ版のXLTを利用

1. XLTから戻ってきたパケットを解析
2. 取得したデータを基にパケットの再構成
3. 外部へのパケット放出

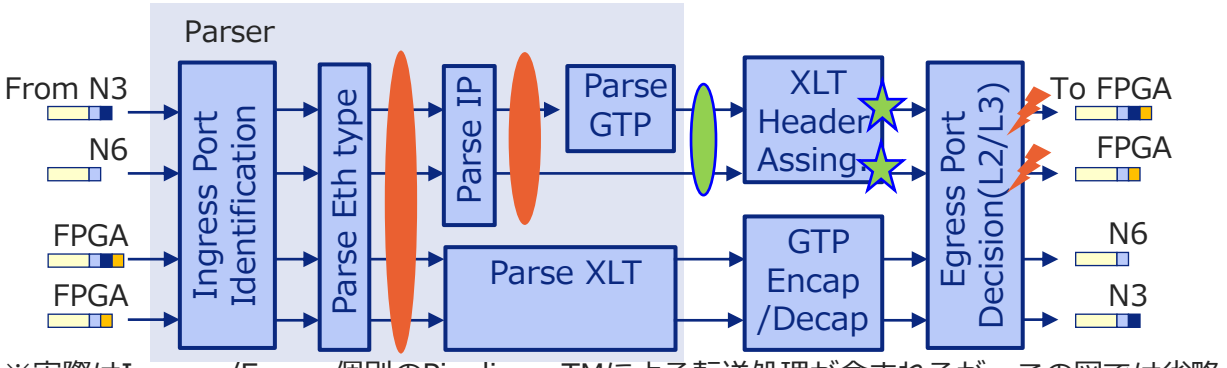
- 解析後XLTヘッダは除去する
  - 取得した転送情報はmetadataへ退避
- metadataの情報を基にGTPv1ヘッダEncap/Decap処理
  - Uplinkの場合：Decap（点線部分の削除）
  - Downlinkの場合：Encap（緑のヘッダ部分の付加）



Step 2：パート① FPGAへのパケット転送で苦戦

当日投影なし

インテル® Tofino™のRegister不足のため、細かな分岐はしない処理に変更
FPGAへの転送にIPルーティングを採用



※実際はIngress/Egress個別のPipeline、TMによる転送処理が含まれるが、この図では省略

- Ingress Portで処理分岐したかった箇所
  （実際はEther typeとPort番号で分岐）
- Ingress Portの処理分岐を実際に使った箇所
- Egress Portを指定した場所
- switch.p4の処理でルーティングができず、
  Drop処理フラグが付与された場所
  （FPGA向けルーティングエントリーを追加して回避）

■当初の想定

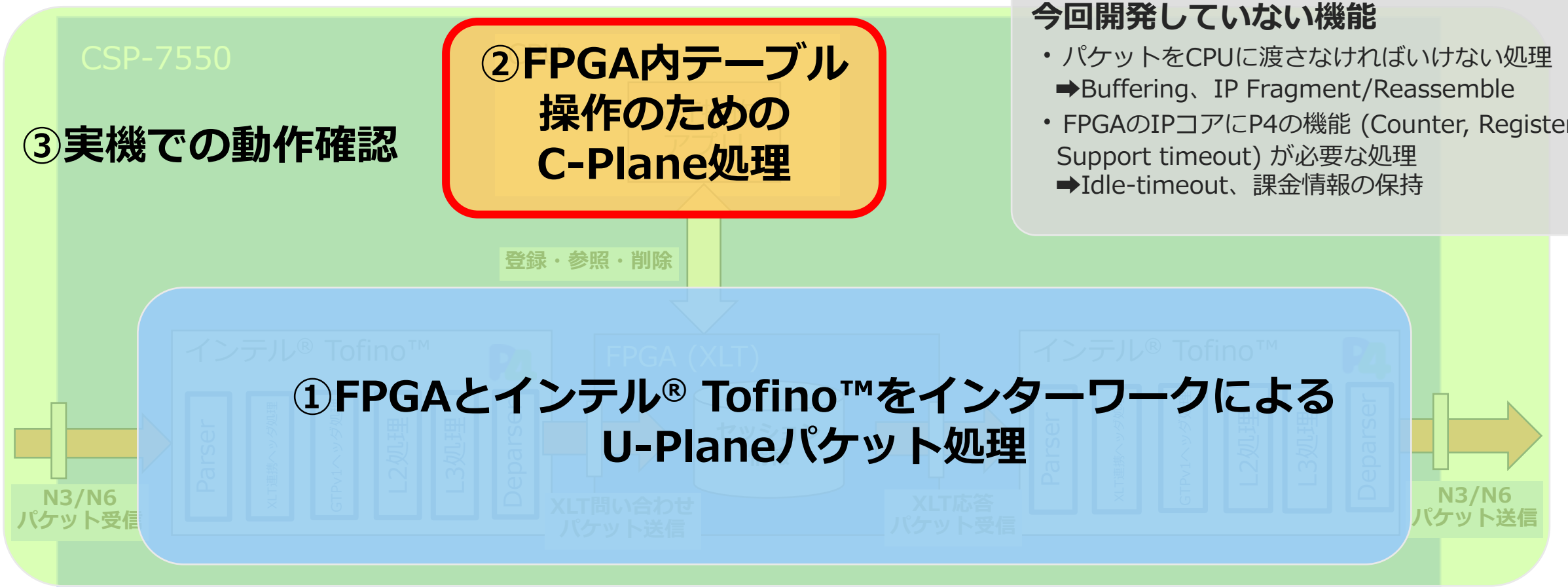
- **Ingress Portを識別 ➡ 必要十分なHeaderのみParseし、FPGAまたはN3/N6に転送**
  - N6から入ってきたパケットにGTP Headerが付いていても無視する、など
- **FPGAへの転送には直接Egress Portを指定**
  - XLTヘッダ付与処理時にmetadataに直接Egress Port番号を書き込んでインテル® Tofino™のTMに転送
  - Egress処理後に指定した物理ポートから出力

■ 実際

- **Parser内でポート識別子を分岐処理に使おうとするとビルドエラー**
  - 原因はParser内のRegister不足。回避方法も入手したものの、修正より先に性能測定できる環境を作ることを優先
- **Egress Portを指定してもインテル® Tofino™内でDropしてしまう**
  ➡ FPGA向けにIPルーティングするように変更
  - switch.p4ベースの処理ではL2、L3ルーティングテーブルにMissするとDropする

Step 2：試作UPFの概要

以降、試作したUPF機能を3パートに分けて紹介していく
今回は一部取り組めなかった機能もある



今回開発していない機能

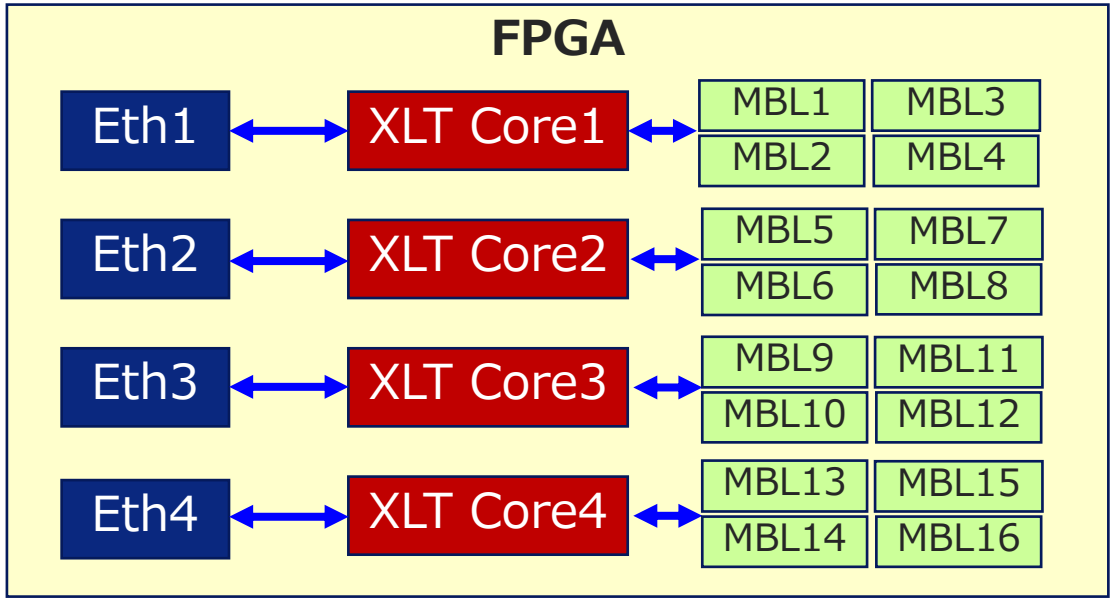
- パケットをCPUに渡さなければいけない処理
  ➡Buffering、IP Fragment/Reassemble
- FPGAのIPコアにP4の機能 (Counter, Register、Support timeout) が必要な処理
  ➡Idle-timeout、課金情報の保持

Step 2：パート② FPGA内テーブル操作のためのC-Plane処理

FPGA内テーブル書き込み/削除/読出しを行うプログラムをC言語で作成 またXLT IPコア*1の制約による課題が明らかになった

- **FPGAのコアとメモリブロック**
  - XLT Coreは、入出力の Ether ポートに１つ接続されている
  - 各コアは、4つのメモリブロック(MBL)を利用できる
  - 各コアのバランシング、使用する MBL は P4で記述が必要
- **課題（制約）**
  - 使用したXLT*1のIPコアは更新ができない
    - 更新には、削除 ➡ 登録の手順が必要で処理時間が増える
    - 一時的にセッション情報が無い瞬間が生じる

    *1 2022年度時点ではβ版のXLTを利用
  - API が thread safe ではない
    - マルチスレッドで処理できず、C-Plane の性能を上げにくい

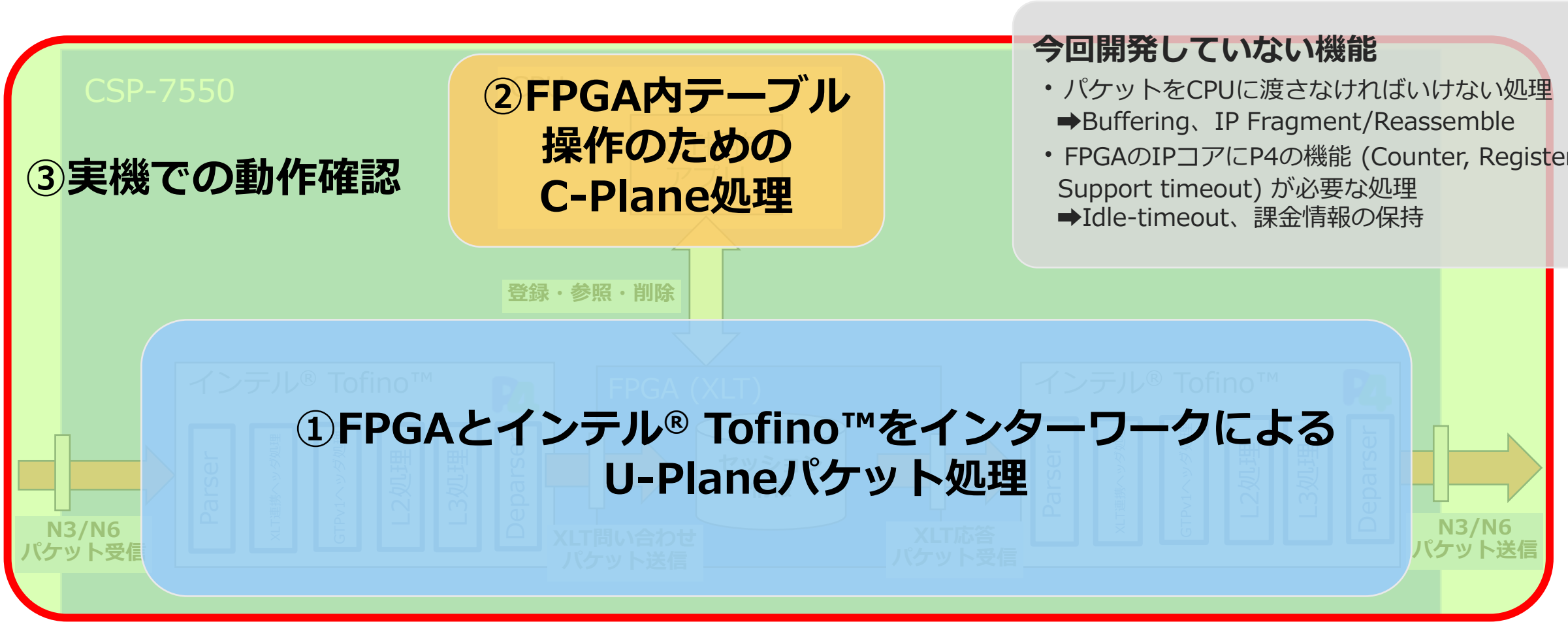


FPGA性能試験は、あらかじめ必要なデータを用意し実施

TEID	IPアドレス	Value1	Value2	・・・	ValueN
aaaaa	XXX	FORW	10.20.30.40	・・・	
bbbbb	YYY	DROP	・・・	・・・	
・・・	・・・	・・・	・・・	・・・	

Step 2：試作UPFの概要

以降、試作したUPF機能を3パートに分けて紹介していく
今回は一部取り組めなかった機能もある

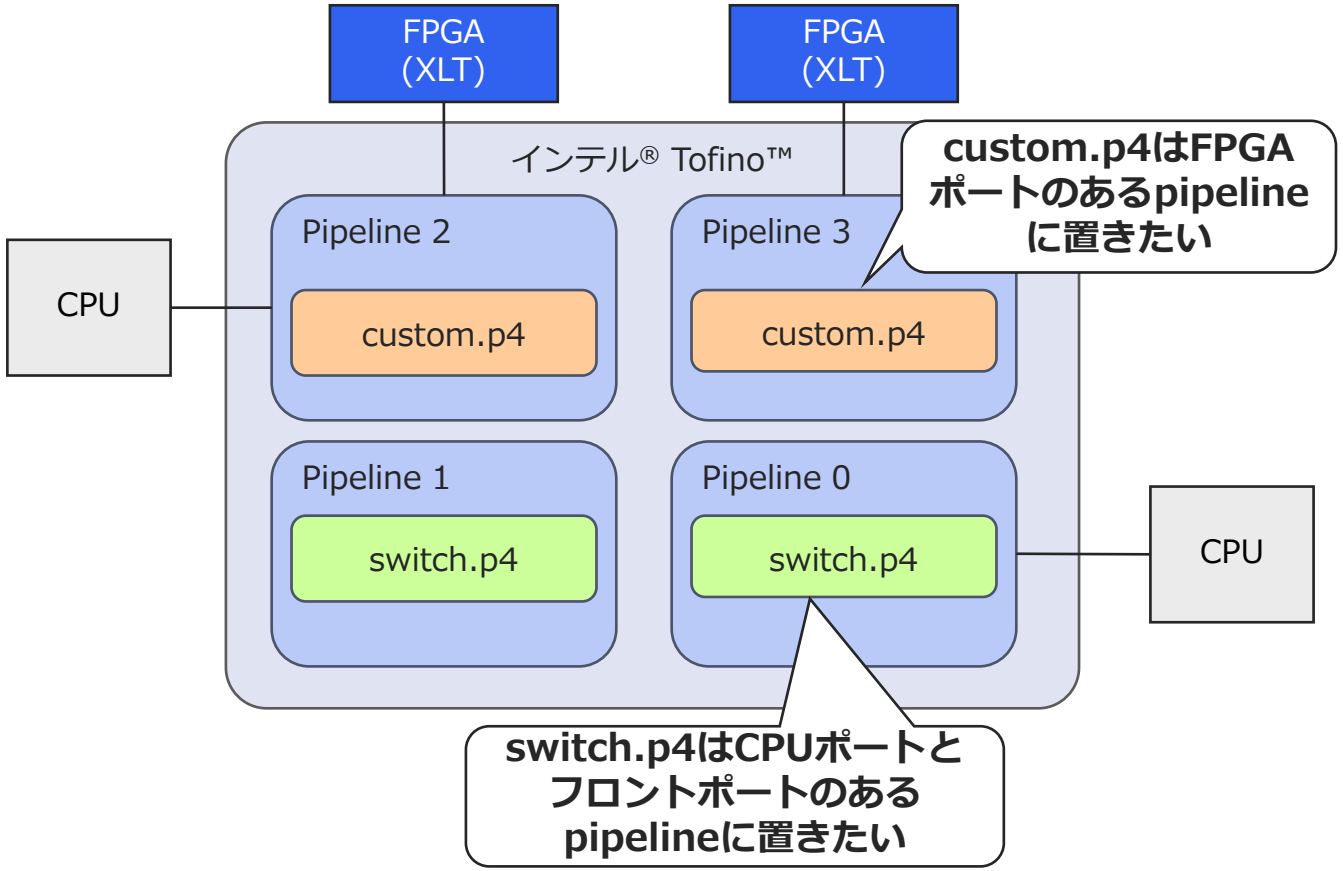


今回開発していない機能

- パケットをCPUに渡さなければいけない処理
  ➡Buffering、IP Fragment/Reassemble
- FPGAのIPコアにP4の機能 (Counter, Register、Support timeout) が必要な処理
  ➡Idle-timeout、課金情報の保持

Step 2：パート③ インテル® Tofino™の内部構成とPipeline

P4プログラムをインテル® Tofino™で動作させるためには HW構成とPipelineを意識する必要がある



■ **PipelineごとにP4プログラムを設定可能**

- **HW構成を意識して、Pipelineに設定するプログラムを実装すると以下のようなメリットがある**

・プログラムがシンプルになる
・Pipelineリソースの消費量が少なくなる
例) switch.p4 = L2/L3処理
custom.p4 = GTPv1ヘッダ、xlt関連の処理

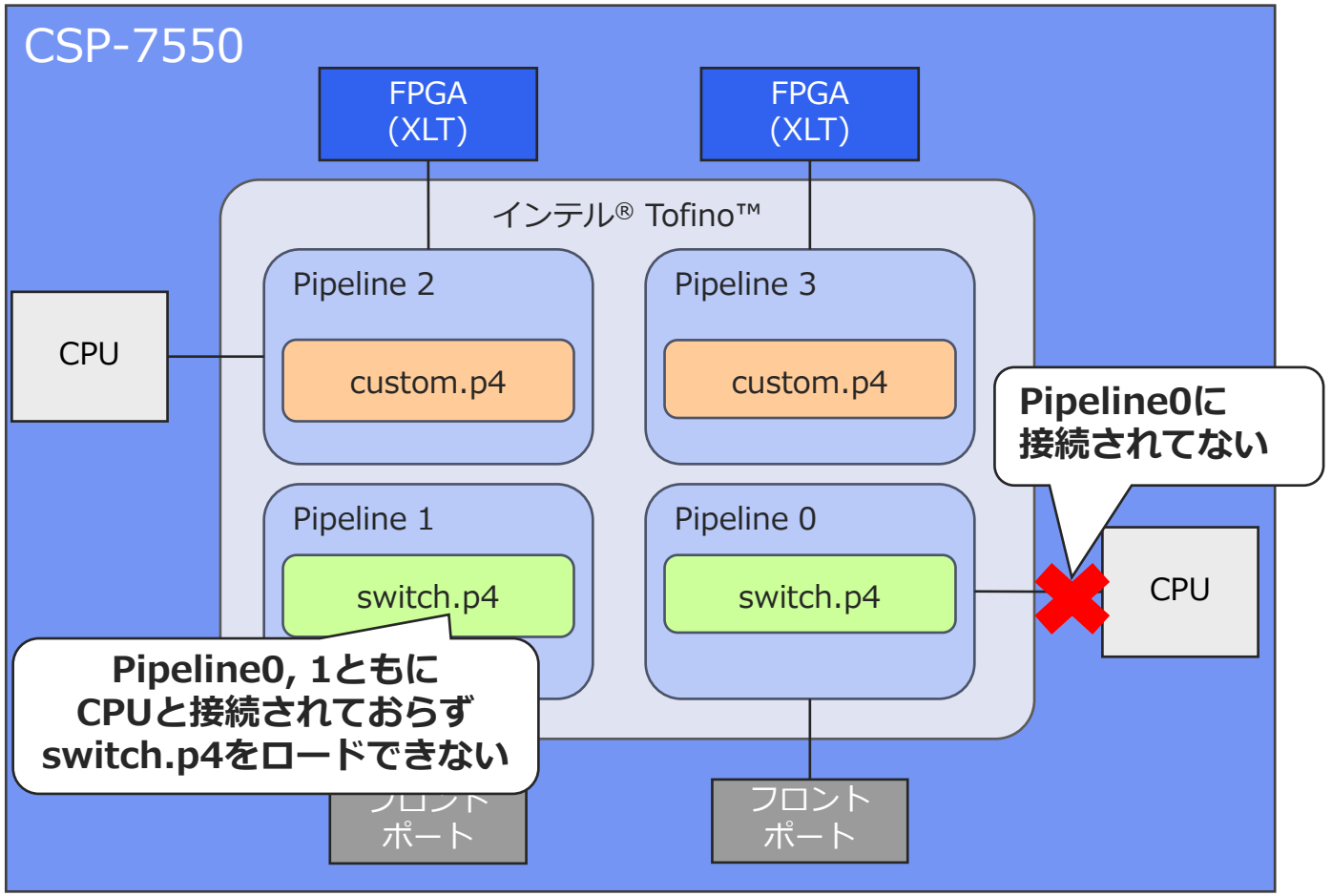
■ **インテル® Tofino™の内部構成**

- **Pipeline0, 2はCPUポートが接続**

CPUポートはswitch.p4を初期化するために必要で、利用するためには少なくともPipeline0, 2のどちらか一つにロードする必要がある

Step 2：パート③ 想定と異なるHWレイアウトに直面

CSP-7550のCPUポート接続が想定と異なり Pipelineの見直し、またはプログラムの改修を余儀なくされる



■ **対応方針は2つ選択肢がある**

① **Pipeline2にswitch.p4をロード**

フロントポートにつながらないため、実際のパケット処理はできずに初期化専用になってしまう

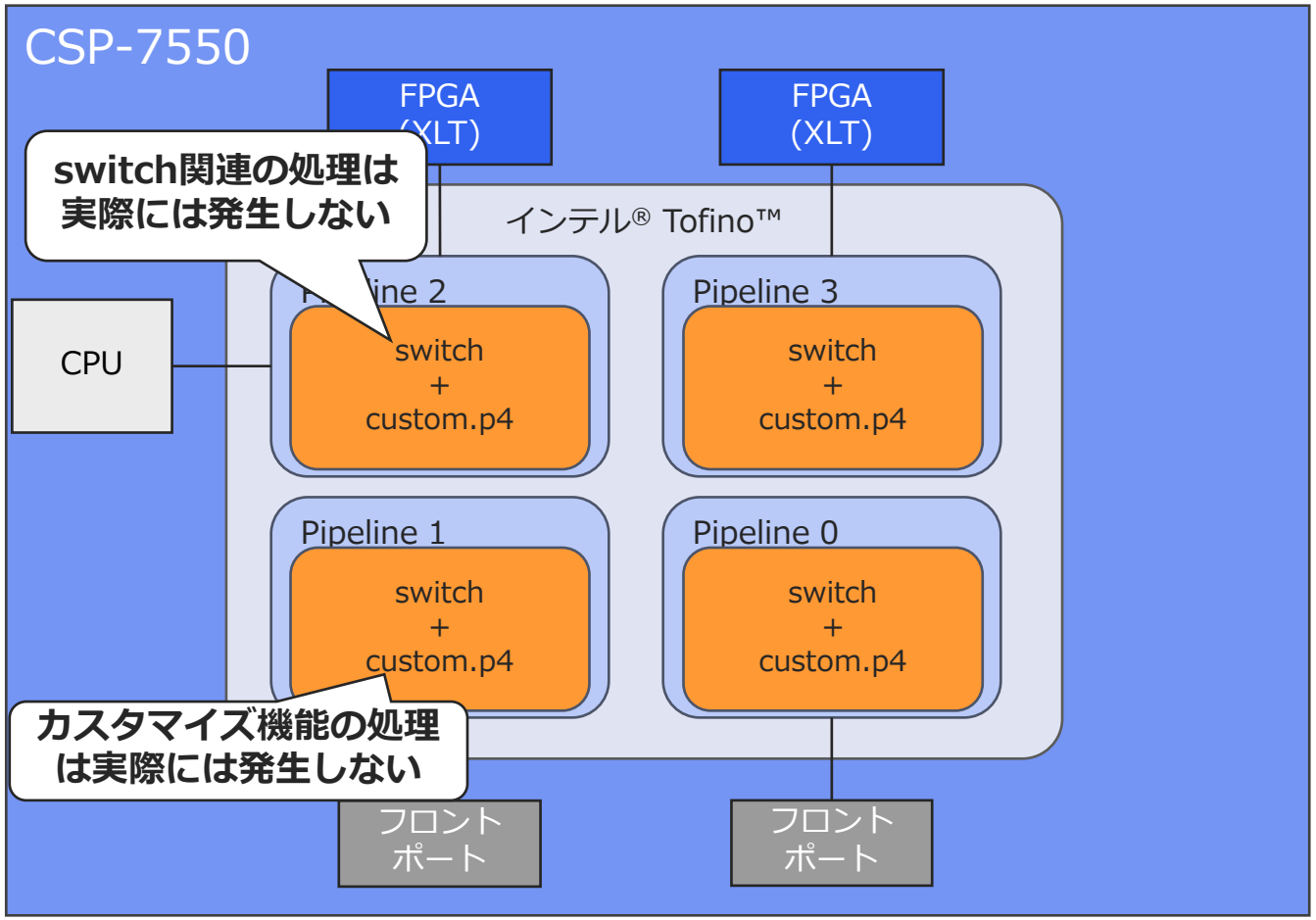
② **switch.p4 + custom.p4を1つのプログラムにする**

全てのPipelineでパケット処理できるが、プログラムが複雑になり、インテル® Tofino™のリソース消費量も多くなる

今回はこちらを選択

Step 2：パート③ 時間がかかりつつも実機で動作確認できた！

全てのPipelineで同じプログラムが動作するように対応
ビルドエラーに苦しむが実機で動作するところまで確認



■U-Planeパケット処理を1つのプログラムにする

- リファレンス実装の既存機能
- GTPv1ヘッダ処理
- XLTヘッダ処理

■上記全てを含めてしまうとリソース不足になるため、不要な機能を削除してテーブルサイズを縮小

- QoS
- ACL
- Telemetry
- IPv6

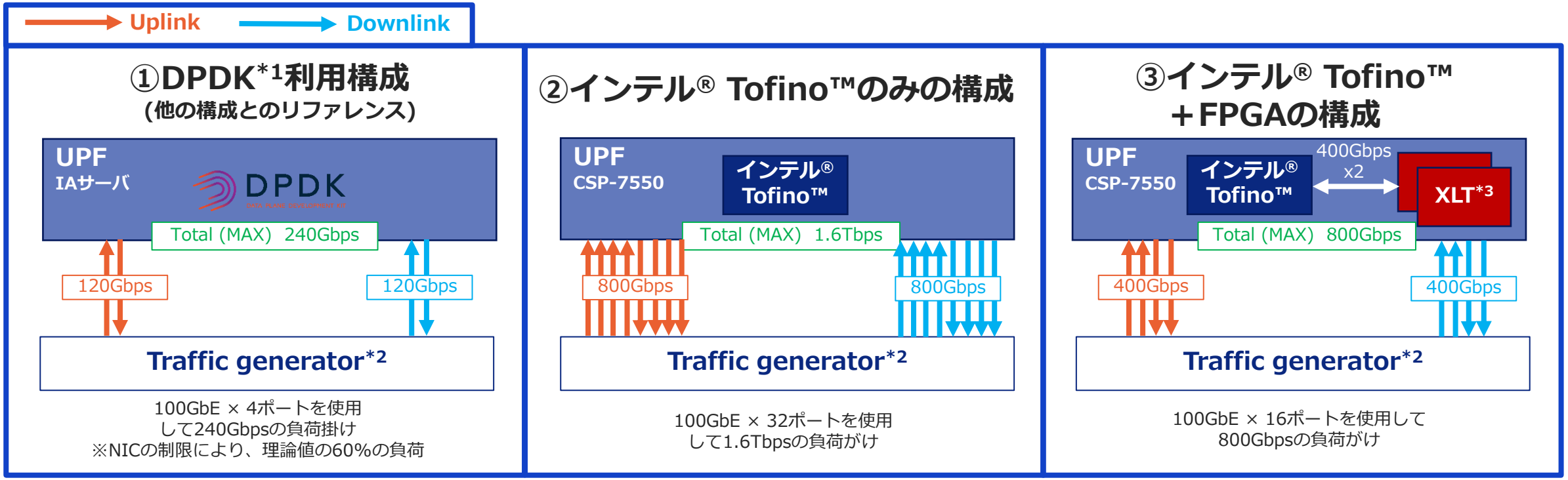
等

UPFの試作　Step 3
試作したUPFの性能を測ってみる

Step 3：性能試験 [構成]

■ 3つの構成で性能を比較

- 性能指標：スループット、消費電力、FPGAの検索遅延、
  最大収容可能セッション数、FPGAのセッションテーブル処理速度　等



*1 Data Plane Development Kit https://www.dpdk.org/

*2 Traffic generatorとしてOSSのTRexを使用
TRex https://trex-tgn.cisco.com/

*3 2022年度時点ではβ版のXLTを利用

①DPDK利用構成

CPU：インテル® Xeon® Gold 6338N プロセッサー@2.20GHz
NIC：インテル® イーサネット・ネットワーク・アダプター E810-CQDA2

Step 3：性能試験 [結果]

- DPDKモデルと比較してワットパフォーマンス約5.9倍向上
- FPGAテーブルオフロードの結果、FPGA2枚で1,664万セッション収容可能

	測定項目		構成①（DPDK）	構成②（インテル® Tofino™）	構成③（インテル® Tofino™+FPGA）
U-Plane	スループット	ロングパケット（ペイロードサイズ1450byte）	233Gbps	1.576Tbps	(未測定) *4
		ショートパケット（ペイロードサイズ22byte）	141Gbps	917Gbps	475Gbps
	消費電力		4,113W @1.6Tbps処理時*1,*2	618W @1.6Tbps処理時*1	697W @1.6Tbps処理時*1,*2
	ワット・パフォーマンス		0.39Gbps/W	2.6Gbps/W	2.3Gbps/W
	FPGAの検索遅延		N/A	N/A	約2μsec*1
	最大収容可能セッション数		未測定 (試験時は2.5万登録)	10万	52万/MBL FPGA2枚だと1,664万
C-Plane	セッションテーブル処理速度 (1セッションあたり)		N/A	N/A	追加 約138μsec*3 約0.7万セッション/秒*3
					読出 約153μsec*3 約0.7万セッション/秒*3
					削除 約130μsec*3 約0.8万セッション/秒*3
その他	温度 (吸気側の電源ユニットで測定)		(未測定)	いずれも常に約30℃をキープ*1 筐体の空冷性能にはまだ余裕がある状態	

*1 U-Plane負荷のみ　C-Plane負荷なし
*2 他の構成/条件の測定結果から算出した推測値(ショートパケット想定)
*3 FPGA1枚に対してU-Plane負荷なし　C-Plane負荷のみ　50万のIPv4セッション処理時間から算出
*4 XLTがβ版のためパケットサイズにより動作確認が不可能であったため

Step 3：性能試験 [考察]

当日投影なし

おおむね期待通りの性能を出せたが、セッションテーブル処理速度は期待よりも性能が出なかった

評価項目	考察
スループット	**Tofino X でも1ポート当たりのPPSはTofinoと変わらない** • Tofino X はFPGA2枚で最大800Gbpsのため、インテル® Tofino™単体(1,600Gbps)の半分で測定 • Tofino X でFPGAが加わっても、1ポート当たりのPPSはインテル® Tofino™と変わらない(約700Mpps)
消費電力	**Tofino X はFPGA分だけ消費電力が高くなる** • インテル® Tofino™単体よりもFPGA分だけワットパフォーマンスは低下 ➡ しかしながら、DPDKより圧倒的に高い • 結果 ○ 構成①DPDK：0.39Gbps/W, 構成②インテル® Tofino™：2.6Gbps/W, 構成③インテル® Tofino™+FPGA：2.3Gbps/W
FPGAの検索遅延	**Tofino X でFPGAが加わってもほぼ遅延なし(2μsec)**
最大収容可能セッション数	**50万セッション × 16MBL × 2FPGA = 1,600万セッション登録可能** • 1FPGA当たり16MBL(メモリブロック)保有 **しかしながら、Support timeoutやRegister/Counter等のP4で実装されていた機能は未実装**
セッションテーブル処理速度	**0.7万セッション/秒で期待よりも処理速度が出なかった*** * FPGA1枚あたり • APIがスレッドセーフではない ➡ クライアントからシングルスレッドでしかセッション追加できず…

UPFを試作してみての考察

UPFの試作目的(再掲)

HWオフロード技術を用いたUPFが
2030年度のCO_2削減目標達成に貢献できるかどうかを確認する

基本機能の実現性 ✖ 実用性の観点

■ HW処理で何ができる？

- GTPv1 Header Encap/Decap
- C-Plane/U-Planeの連携
- Idle-Timeout機能の実現
- 課金情報の保持＆レポーティング
- Packet Buffering
- IP Fragment/Reassemble

■ パフォーマンス

- エコになっても、処理が遅ければ要らない（耐えられない）
- 消費電力はどれぐらい下がる？

■ 商用化にあたって

- （定性的に）内製での開発継続性、運用機能の実装容易性、保守対応の難しさなどはある？

基本機能の実現性に対する考察

全機能をP4とTofino X で実現できなかったが、P4自体は可能性に期待

基本機能の実現性

■ **HW処理で何ができる？**

- GTPv1 Header Encap/Decap
- C-Plane/U-Planeの連携
- Idle-Timeout機能の実現
- 課金情報の保持&レポーティング
- Packet Buffering
- IP Fragment/Reassemble

今回Tofino X では検証できなかった

■ **P4言語**

- アプリ開発者にもパケット処理を簡単に記述できる

■ **インテル® Tofino™**

- プログラマブルで高速/効率的な処理が可能
- ツール類 (PTF/P4i/bfshell) による効率的なロジック検証、バグ改修
- UPF機能すべてをインテル® Tofino™ベースで実装するにはリソースが足りない

■ **FPGA**

- テーブル領域の拡張が可能だった
- 取り扱いに四苦八苦

実用性の観点に対する考察

アプリ開発者目線だとHWを意識した開発が難しかった

実用性の観点

■ **パフォーマンス**
- エコになっても、処理が遅ければ要らない（耐えられない）
- 消費電力はどれぐらい下がる？

■ **商用化にあたって**
- （定性的に）内製での開発継続性、運用機能の実装容易性、保守対応の難しさなどはある？

■ **主要機能のパフォーマンス**
- FPGAによる処理遅延増は微小
- DPDK構成と比較するとワットパフォーマンスで約5.9倍効率アップ

■ **商用化に向けた開発継続性**
- FPGA/HW開発者とアプリ開発者の協業（双方を理解した開発）
- 今回開発できなかったUPF機能＋運用保守に必要な機能を誰が開発できるかという課題

■ **商用運用・保守**
- HWを外部調達、アプリは内製で保守ができるか、体制構築が鍵

まとめ

今回お話したこと

- トラフィック量増加への対応とカーボンニュートラル実現を両立させるため、HWオフロード技術を用いて、モバイルトラフィックの要であるUPFの低消費電力化を試みた
- プラットフォームとしてTofino X アーキテクチャを採用し、UPFの一部機能を試作/性能測定した
  - UPFのユーザセッションをFPGAで管理し、N3/N6パケットをXLTにルーティングするさせるアーキテクチャを設計し実装
  - 性能はCPU版構成と比較すると約5.9倍のワットパフォーマンスであることを確認した
- 開発中にインテル® Tofino™のリソース不足、CSP-7550のHWレイアウトの制約、FPGAの取り扱いに苦労した
- Tofino X で全てを実現することができなかったが、P4の将来性には期待が持てた

ディスカッションしたいこと [1/4]

テーマ①	P4言語を用いた開発/P4への要望
詳細	• Network Functionの機能を全てP4で実現するような大規模な開発を実施したケースはあるか？ • ユーザトラフィックをP4で記述する際に足りていないと感じる機能やそのユースケースはあるか？
KDDIの考え	• 自分たちと同じような道を歩んでいる方がいたら、是非先人の方々の知恵をお借りしたい • 今回はBufferingやFragment/Reassembleはノータッチだったが解決策のご提案、また、もし同じようにP4で書けたら嬉しい機能のご提案があれば伺いたい

ディスカッションしたいこと [2/4]

テーマ②	**P4を使った開発者はHWをどこまで意識すべき？簡単にならない？**
詳細	• P4を扱うソフトウェアエンジニアはどこまでHWを意識できるのか？またはどこまで意識するべきなのか？ • P4を用いた開発中にハードウェアのリソース不足で困ったことがあるか、どう対応したのか？ • ハードウェアを拡張した場合に簡単にExtern*1 を作れると嬉しいが、実現可能だろうか？
KDDIの考え	• 今回はHWを意識せずにスタートを切ったが、最初からHWエンジニアも体制に組み込まないといけないものなのか？需要の大きくない領域で協力体制を築けるものなのか？ • やりたいこと(アプリ規模)をハードウェア技術者やベンダ様にインプットして共同開発すれば回避可能？ • TableをFPGAに拡張した際、P4から簡単に参照できると開発効率も良かったと思う。そのような取り組みがったら採用していきたい（最近、SmartNIC側でこういった動きが盛んになっている？）

*1　P4ではパケット処理でよく使われる機能(パケットカウンタやチェックサム計算など)や、独自に実装した機能をExternという形で記述して利用することができる

ディスカッションしたいこと [3/4]

テーマ③	プログラマブルなパケット処理プラットフォームについて
詳細	• 中長期的にみてインテル® Tofino™を採用することについて • その他のプログラマブルなプラットフォームとして台頭してきているもの(提案者目線)、期待できるもの(利用者目線)等あるか？特にP4が活用できると嬉しい
KDDIの考え	• オペレータ目線だと長くお付き合いできるパートナーづくりが重要と認識。一方で特定用途であれば使い続けたいニーズは残り続けるのか • インテル® Tofino™の後継のような大規模なプログラマブルASIC、使い勝手の良いSmartNICが登場し、CN貢献できそうであれば試してみたい

テーマ④	オペレータが内製開発することについての本音と建前
KDDIの考え	• 建前的には内製の領域拡大と早期の実現性判断のため新機能をアジャイルでどんどん開発していく • 本音はこの後、質問があれば答えます

ディスカッションしたいこと [4/4]

テーマ	詳細
①P4言語を用いた開発/P4への要望	• Network Functionの機能を全てP4で実現するような大規模な開発を実施したケースはあるか？ • ユーザトラフィックをP4で記述する際に足りていないと感じる機能やそのユースケースはあるか？
②P4を使った開発者はHWをどこまで意識すべき？簡単にならない？	• P4を扱うソフトウェアエンジニアはどこまでHWを意識できるのか？またはどこまで意識するべきなのか？ • P4を用いた開発中にハードウェアのリソース不足で困ったことがあるか、どう対応したのか？ • ハードウェアを拡張した場合に簡単にExternを作れると嬉しいが、実現可能だろうか？
③プログラマブルなパケット処理プラットフォームについて	• 中長期的にみてインテル® Tofino™を採用することについて • その他のプログラマブルなプラットフォームとして台頭してきているもの(提案者目線)、期待できるもの(利用者目線)等あるか？特にP4が活用できると嬉しい
④オペレータが内製開発することについての本音と建前	• 建前的には内製の領域拡大と早期の実現性判断のため新機能をアジャイルでどんどん開発していく • 本音はこの後、質問があれば答えます

Tomorrow, Together
KDDI
おもしろいほうの未来へ。
au

「つなぐチカラ」を進化させ、
誰もが思いを実現できる社会をつくる。

KDDI VISION 2030