俺SoC, Laxer Chip
AX1001 のご紹介

2024/NOV/08

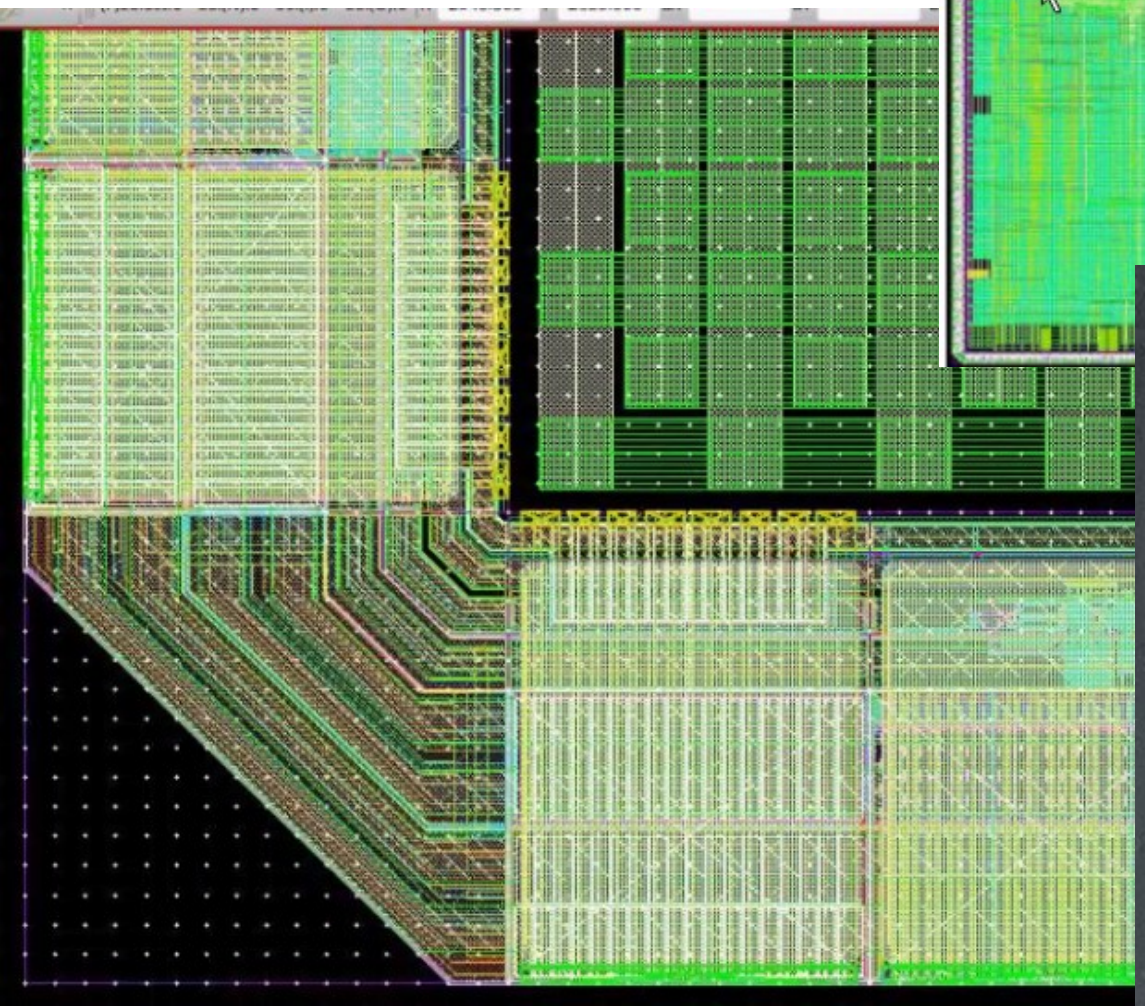
株式会社アックス

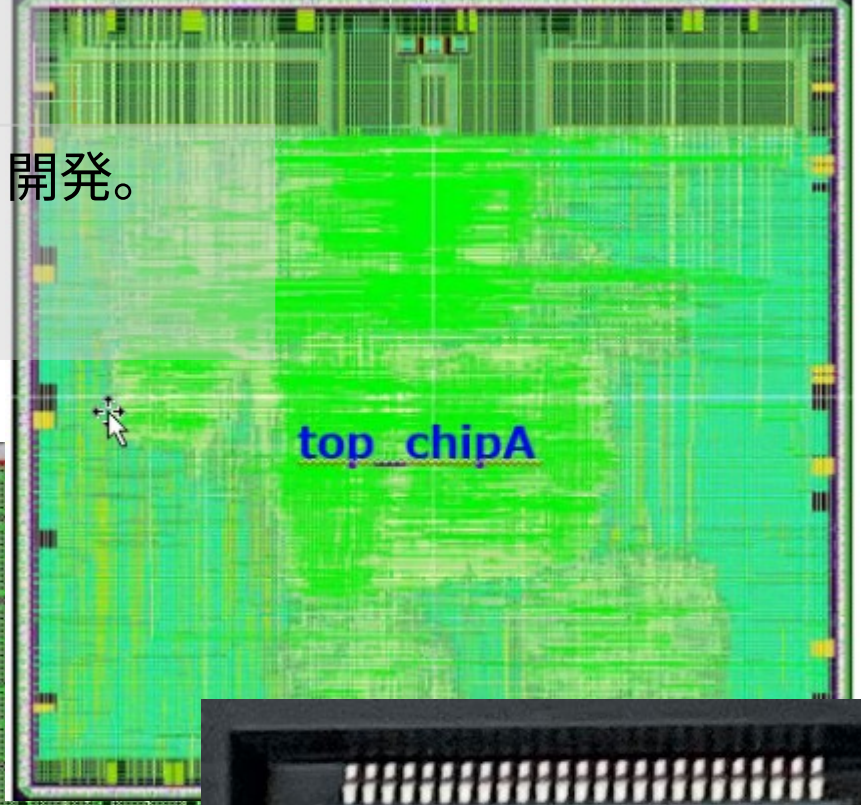
竹岡 尚三

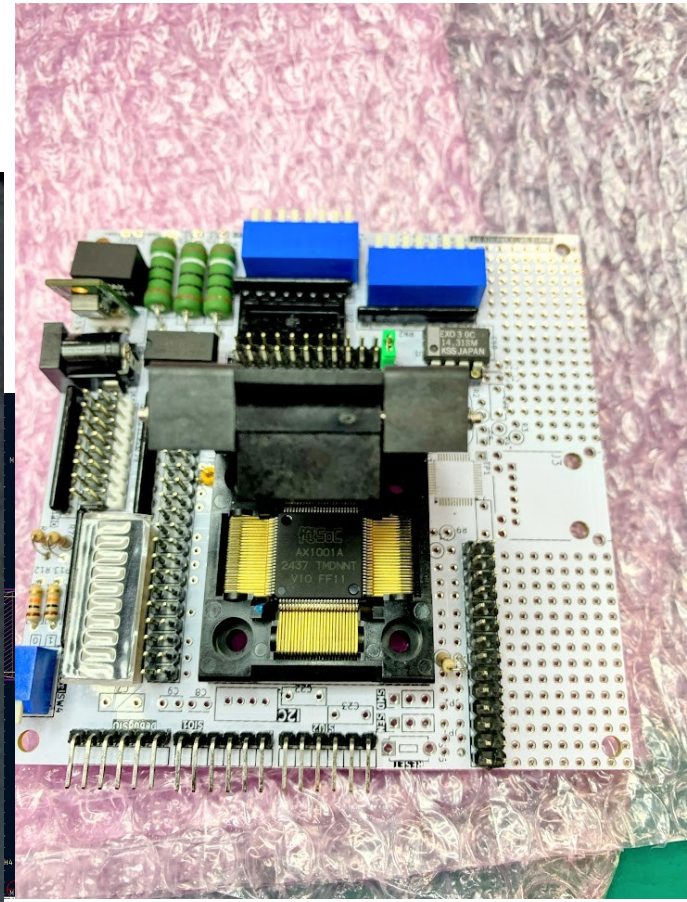
AXEの完全オリジナルLSI

TSMC 90nmで「俺SoC」LaxerChipを開発。
LQFP 100pin
27/SEP/2024 京都で1stシリコン動作

俺SoC







“俺SoC”,とことんOS無し

- 省電力、省メモリ、堅牢かつ高速
- ロボットの部品モジュールがローコストで簡単に作

俺SoC

オレ達のCPU「松竹V(しょうちくぶい)」

- 機械学習AI 加速用 ベクトル計算ユニット(8bit float,4SIMD,パイプライン)
- 論理推論加速 機構をRISC-Vコアに追加
  - 特許取得済み(第7506718号,2024年6月18日（東京エレクトロンと共同特許))
- GnuPrologのコンパイルド・バイナリを加速
- ハードウェア・マルチスレッド機構
  - OSソフトウェア一切なしで、スレッド切り替え
  - ハードウェア・セマフォで排他/同期。LR/SCもある
  - 外部ピンからの入力で、スレッド起床(ハードウェアのみで)
  - 割り込みなし(割り込み相当の処理は、専用スレッドで)

エッジデバイスでも
大脳的処理を!

OSプログラム・コード
OSワーキング・エリア
不要!

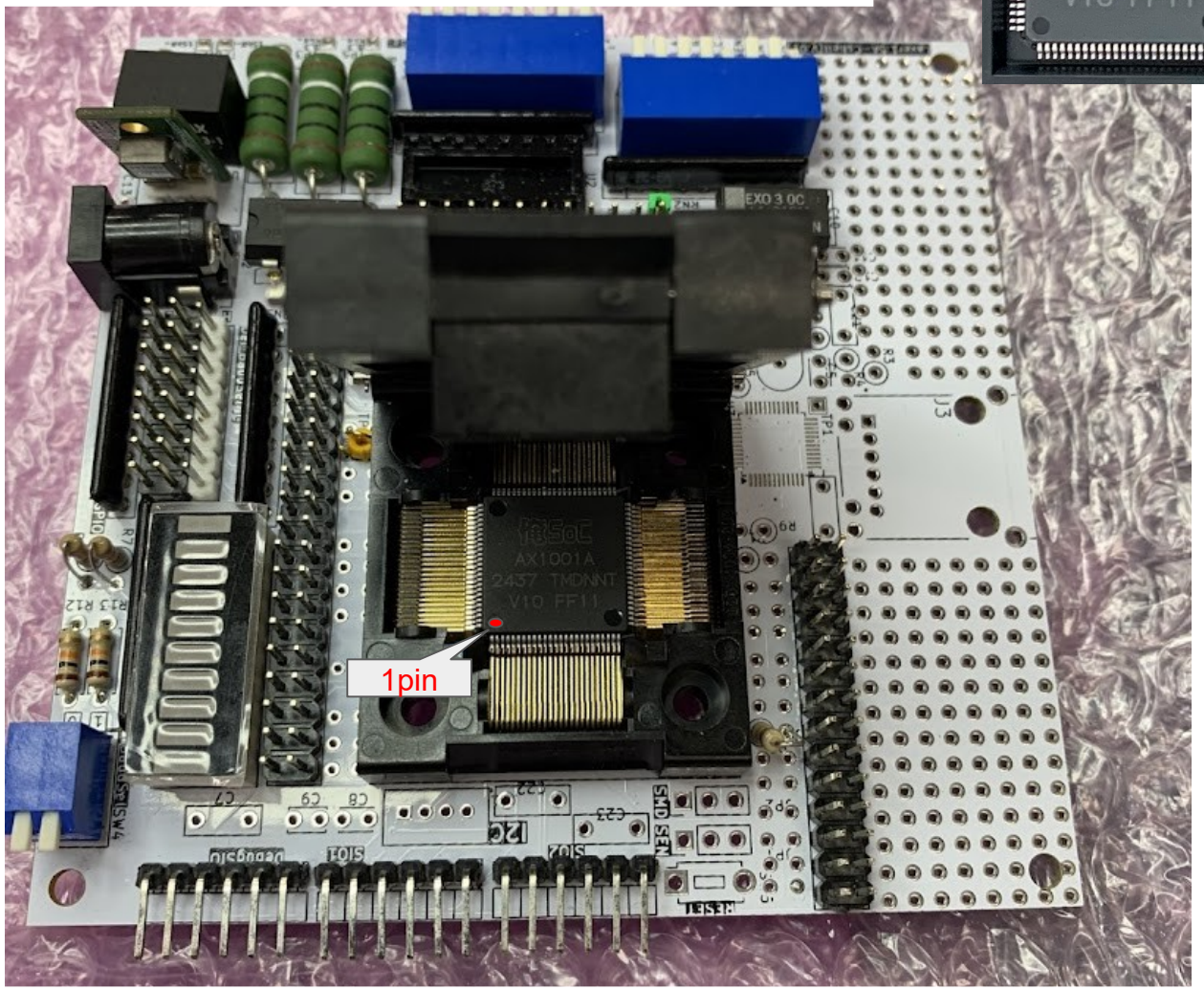
ROS2通信ハードウェア”ROS2rapper”を搭載

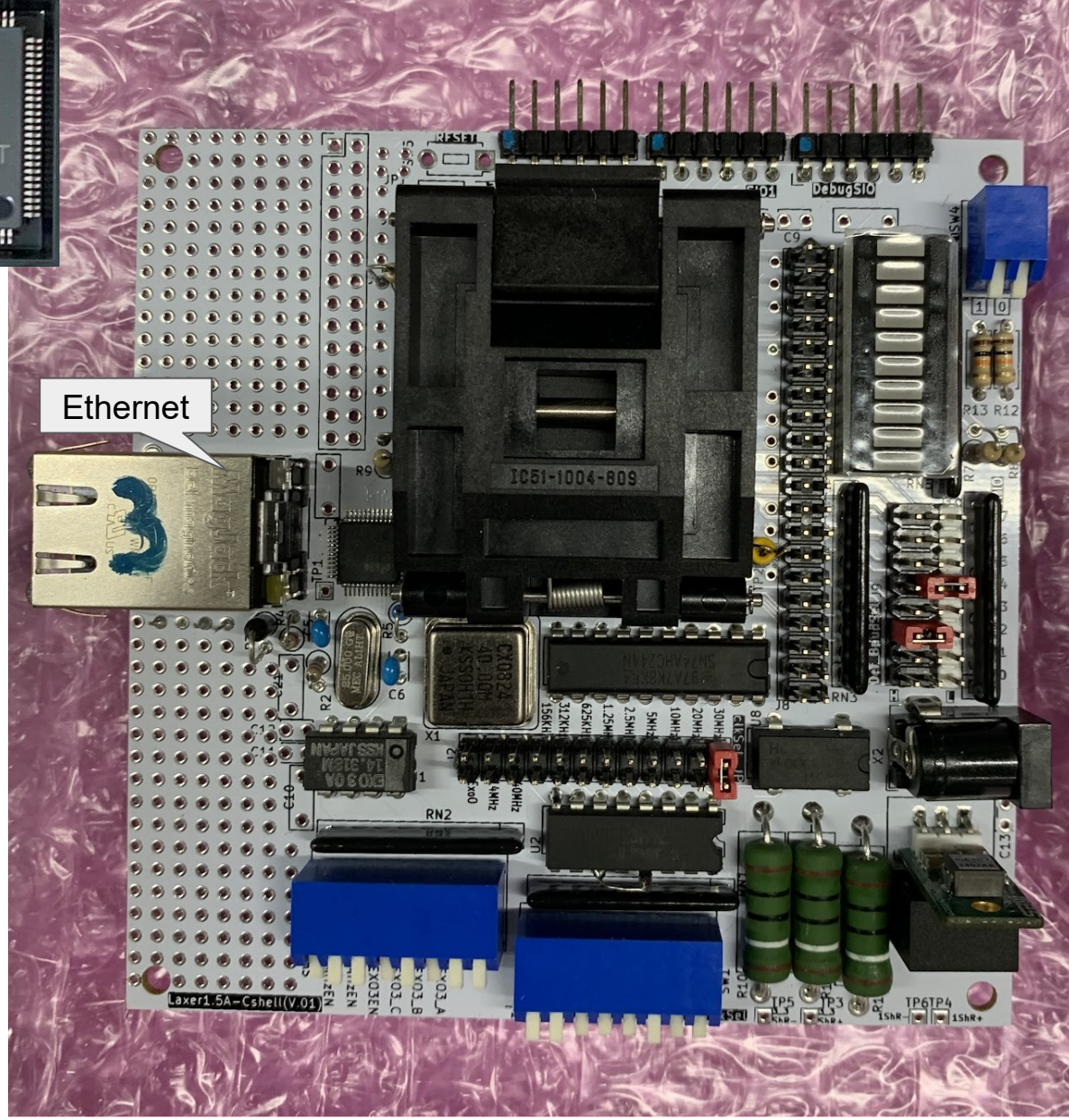
- CPUの助けなしにROS2通信

OS 不要のROS!

外部I/O: Ethernet I/F, 任意波形生成器(AWG(PWMにもなる)), GPIO

テスト基板
Ethernetで、ハードウェアROS2が動作

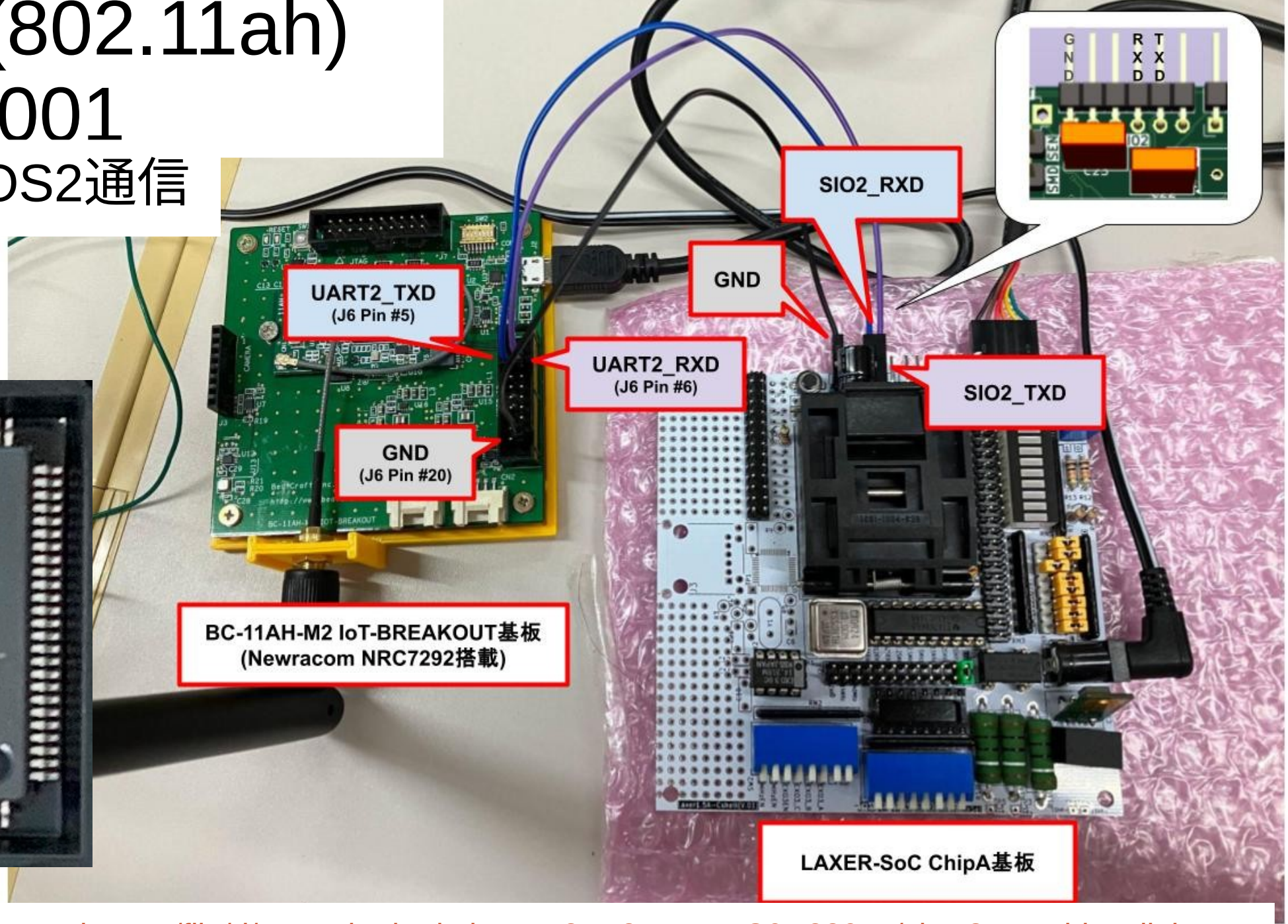




LEDチカチカ https://drive.google.com/file/d/10z89qI9ZQaNe8txQFIMjL5HRLol7bTh4/view?usp=drive_link
ROS2Subscribe https://drive.google.com/file/d/1M2Jt1u0B1dGReELrE8qoQbPvnz0AZl3c/view?usp=drive_link
ROS2Publish https://drive.google.com/file/d/1CCb1PVTXruBFXhebgIjcXFeVvfWzMIoC/view?usp=drive_link

Newracom(802.11ah) +LaxerAX1001

- 802.11ahでROS2通信

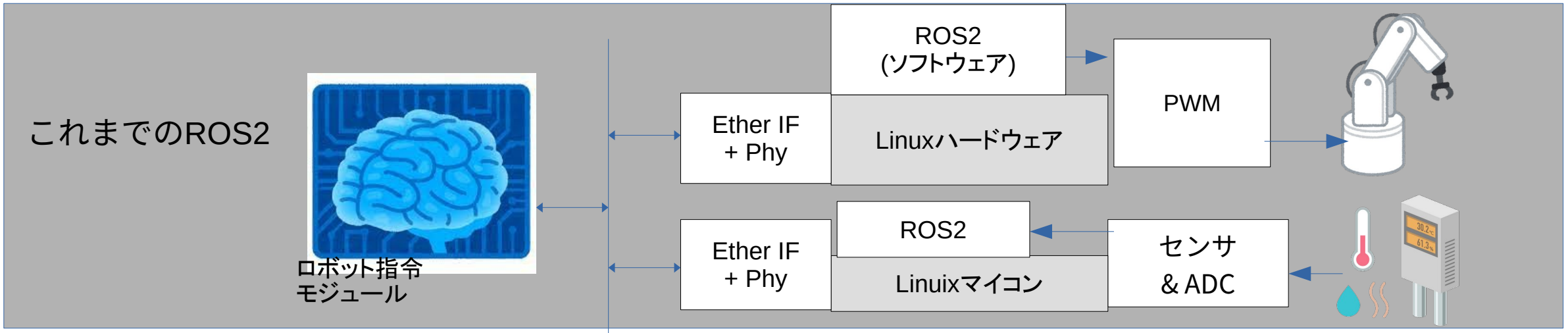


ROS2通信動画 https://drive.google.com/file/d/1WARjMdFghvjwwW-3gw8TR1HwC0N380Fx/view?usp=drive_link

ROS2プロトコル

- ROS2は、ロボット業界のデファクトスタンダードになろうとしている
  - ロボットの部品モジュールはROS2プロトコルで結びつける
  - ロボットの部品モジュールの流通性を高める
  - 自動運転でも使用されている
- ROS2は、これまで Linux、一部のRTOSで動作するソフトウェアだった
  - ✗ Linuxは、MMUのついた高級なCPUで動作。メモリが多く必要。高電力消費。

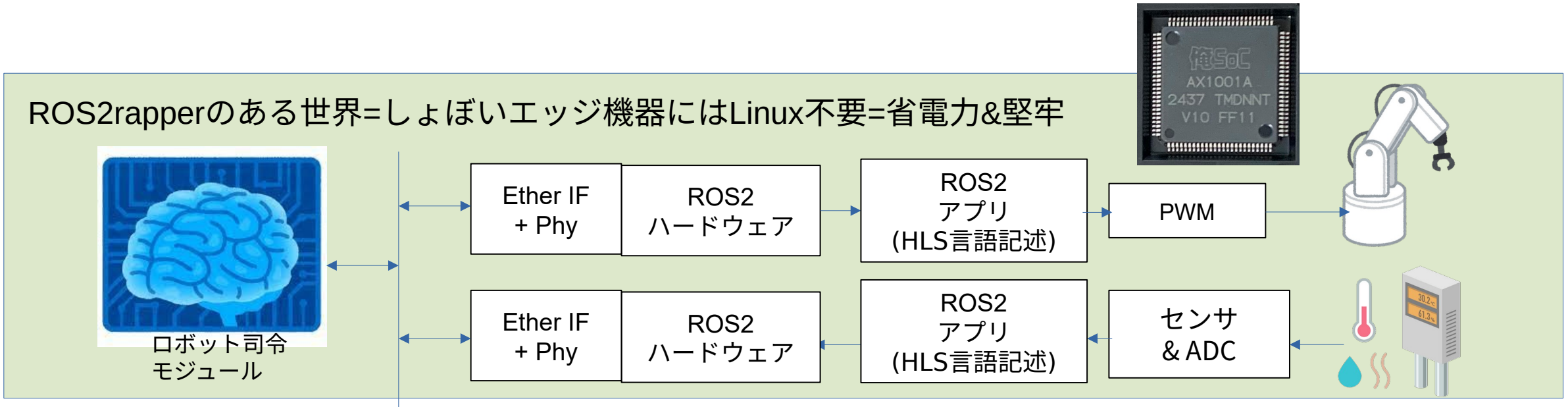
AXEでは、ROS2プロトコルを完全ハードウェア化した”ROS2rapper”を開発



ROS2プロトコルを完全ハードウェア化した…

AXE製、”ROS2rapper”

- CPU無しで、ロボットの部品モジュールができる
  - センサとROS2プロトコルHWだけで、センサ・モジュール
  - PWMとROS2プロトコルHWだけで、アクチュエータ・モジュール
  - アプリケーションはC言語で書いておけば、すぐハードウェア論理に合成
- ロボット部品が、ゴミのようなLSIでできる ←CPU不要



オレ達のCPU「松竹V(しょうちくぶい)」の排他制御

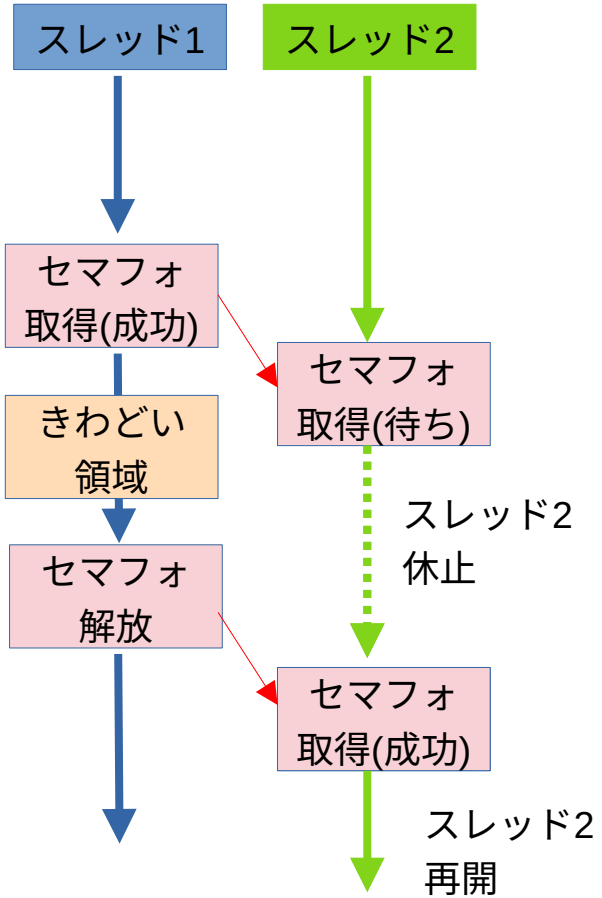
“俺SoC”,とことんOS無し

- **省電力、省メモリ、堅牢かつ高速**
- **ロボットの部品モジュールがローコストで簡単に作れる**

マルチタスクだが、
OSプログラム・コード
OSワーキング・エリア
不要!

ハードウェア・マルチスレッド機構

- OSソフトウェア一切なしで、スレッド切り替え
- ハードウェア・セマフォで排他/同期(重要)
- LR/SC(RISC-Vの排他制御プリミティブ)もある
- 外部ピンからの入力で、スレッド起床(ハードウェアのみで)
  - 割り込みなし(割り込み相当の処理は、専用スレッドで)

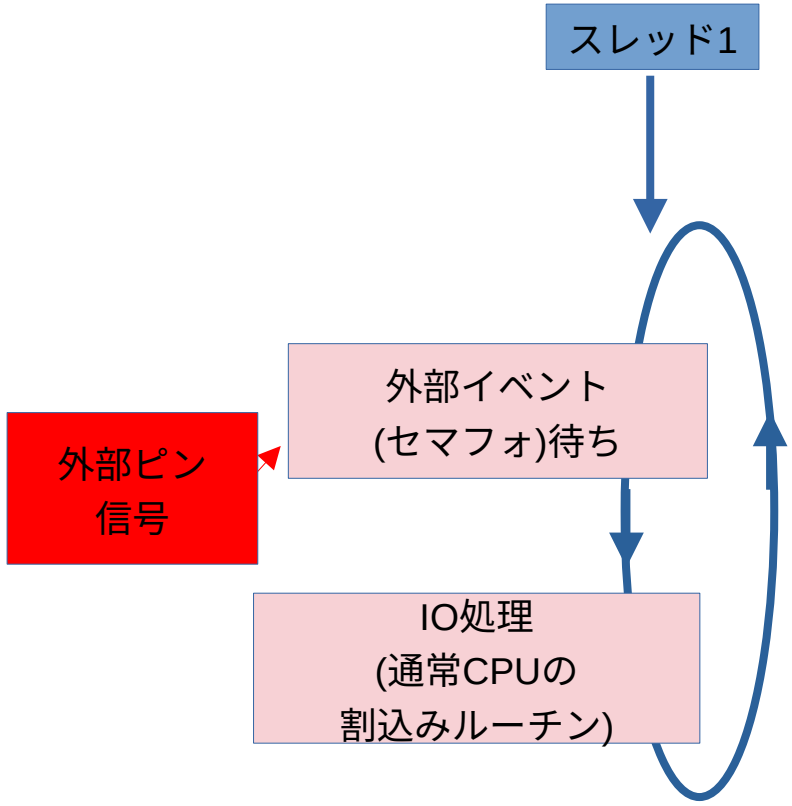


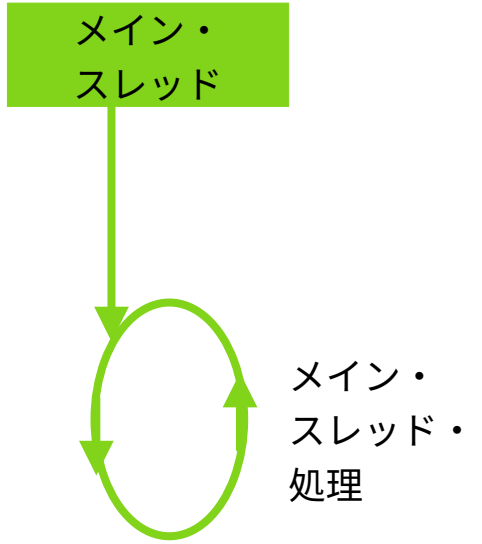
オレ達のCPU「松竹V(しょうちくぶい)」の外部イベント(割込み相当)

マルチタスクだが、
OSプログラム・コード
OSワーキング・エリア
不要!

ハードウェア・マルチスレッド機構

- OSソフトウェア一切なしで、スレッド切り替え
- 外部ピンからの入力で、スレッド起床(ハードウェアのみで)
  - 割り込みなし
- 割り込み相当の処理は、専用スレッドで。
- 割込み起動より、高速
  - レジスタ退避などしないから





オレ達のCPU「松竹V(しょうちくぶい)」

機械学習AI 加速用 ベクトル計算ユニット

8bit float,4SIMD,ベクトル・パイプライン

動的クロック切り替え機構で500KHz～320MHzで、動作

- 機械学習AIの推論に最適な8bit浮動小数点演算をベクトル処理
- エッジ機器内での、AI処理

↓

- データ通信量の削減

エッジデバイスでも
AI処理を!

- 分散処理による、全体の負荷の分散

オレ達のCPU「松竹V(しょうちくぶい)」

- **省電力、省メモリ、堅牢かつ高速**
- **ロボットの部品モジュールがローコストで簡単に作れる**

論理推論 AI加速 機構をRISC-Vコアに追加

- 特許取得済み
- 第7506718号,2024年6月18日（東京エレクトロンと共同特許)

- GnuPrologのコンパイルド・バイナリを加速
  - 詳細は、付録参照のこと

エッジデバイスでも
大脳的処理を!

Laxer AX1001 概要仕様

機能	数	備考
CPU	1	RISC-V RV32Iに、4SIMD 8bit浮動小数点ベクトル機構、ハードウェアによるマルチタスク制御機構、多重分岐命令(特許取得済、Prolog,Lisp,JavaVMの実行を加速)と、それらを操作する命令を付加
RAM	1	プログラム・コードRAM 64KBytes。reset後、bootloaderにより、SIO経由でプログラム・コードを配置可能。CPUコアによるコードRAMの内容変更はできない
	1	データRAM 64KBytes。
SIO(UART)	3	SIO0(UART0)は、bootloadingとデバッグ・コンソールに使用される。UART1,2はユーザが自由に使用できる
GPIO	38	18本はPullUP指定可能。20本はPullDown指定可能。2本は、SIO1と兼用。2本は、SIO2と兼用。18本は外部イベント源として使用可能。(※外部イベントとは、一般CPUにおける割込のようなもの。外部イベントが発生すると、イベントについて待機しているスレッドが起床する)
AWG	1	16bit出力,1ch。出力ピンは、PWMと兼用。PWMと排他的に使用
PWM	8	出力ピンは、AWGと兼用。AWGと排他的に使用
Ether I/F	1	NICはオンチップであり。PHY I/Fを持つ (PHYは外付け)
ROS2通信機能	1	"ROS2rapper"という名称の、新開発のハードウェアによるROS2通信機能。CPUから初期化することで、自動的に通信を行う
外部SRAM I/F	1	32bit,入力/出力独立。ただし、AX1001では外部ピンの制約により使用できない

Laxer AX1001 消費電力概要

	動かすテストプログラムの名前	CPUクロック周波数・消費電力(mW)					
-	-	320MHz			1.79MHz		
-	-	実測(1.0v)mA	実測(3.3v)mA	1.0V+3.3v計(mW)	実測(1.0v)mA	実測(3.3v)mA	1.0V+3.3v計(mW)
	メモリテスト	177.0	5.0	193.5	10.0	0.0	10.0
	有意義なことはしていない	113.0	11.0	149.3	30	10	63.0

参考	※実際には、DC-DCコンバータなどで損失が出るので、仮の理想値	電池容量(mW換算)	320MHz(時間)	1.79MHz(時間)
	単1形アルカリ電池容量 約10,000 mAh/1本, 3本4.5v使用	135,000	697.67	13,500.00
	単3形アルカリ電池 容量約1,000～2,900mAh/1本, 3本4.5v使用	39,150	202.33	3,915.00
	コイン電池CR2450 (3V) 620mAh，2個(6V)使用 (※最大電流30mA/1個)	7,440	(38.45 ※電流が足りない)	744.00

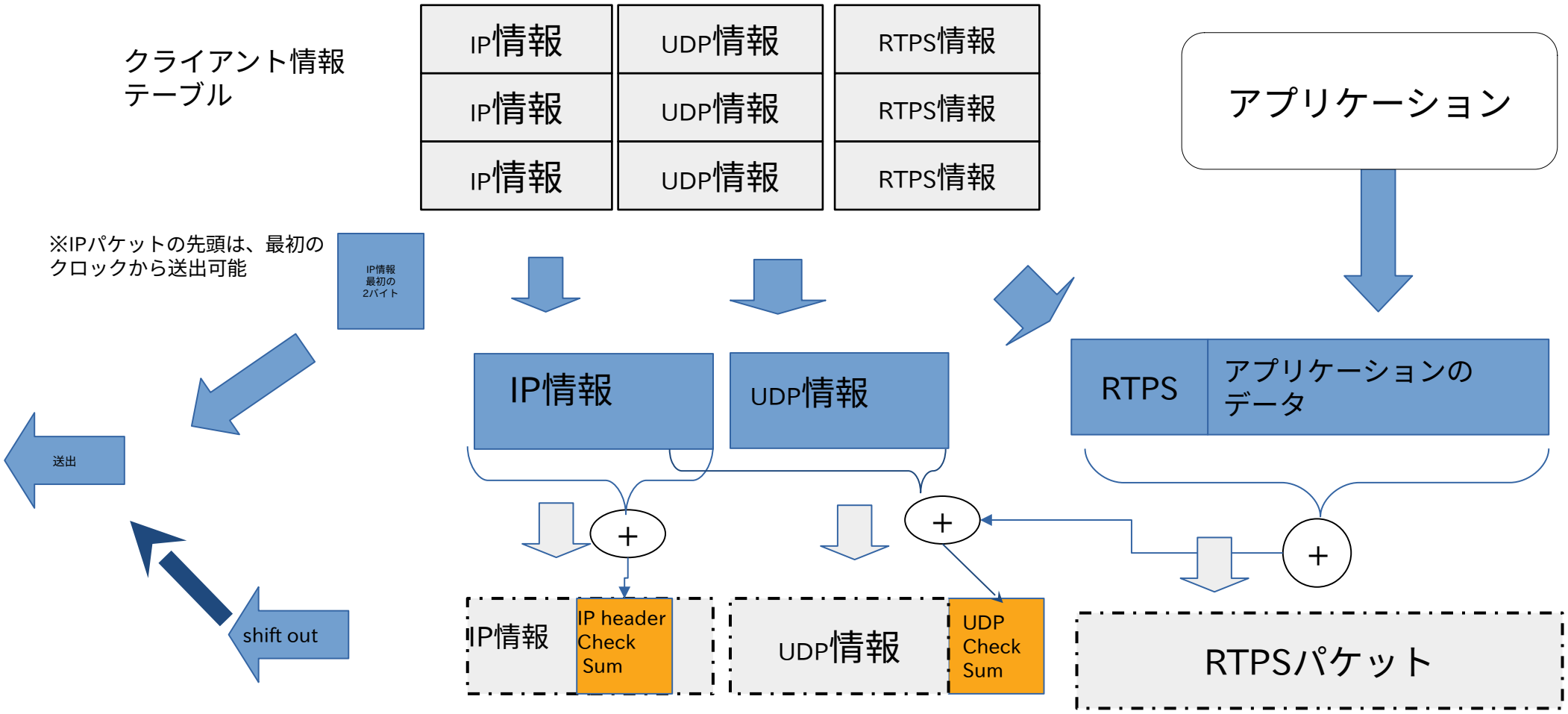
オープンソースな
ROS2通信ハードウェアIP
“ROS2rapper”

https://github.com/AXE-jp/ros2rapp

セキュリティ面 でのオール・ハードウェアの優位性

- セキュリティ・ホールができにくい
  - スタック・オーバフロー 攻撃は、原則起きない
    - 「制御を取られる」ということが無い
      - CPUでは、悪者コードの先頭へ、プログラム・カウンタをセットする
  - ユーザの書いたアプリケーションが、セキュリティ・ホールを産むことはありえる
    - 下回りが完璧でも、ユーザのポカはあり得る
- DOSアタック (無駄なパケットを間断なく投げつけてくる) には、勝てない(負けるとも言わないが)
- 仮にセキュリティ・ホールを突かれても、リセットすると 完全復帰
  - FPGAのコンフィギュレーションROMを書き換えることは、ほぼ不可能
    - ユーザ・アプリケーションがバカでも、そこまではいかないだろう…(?)
  - ※Linuxなどの複雑なOSは、重要な設定ファイルなどを書き換えられてしまったら、再起動してもダメ

ネットワーク・パケットの並列生成(2021/SEP/27)

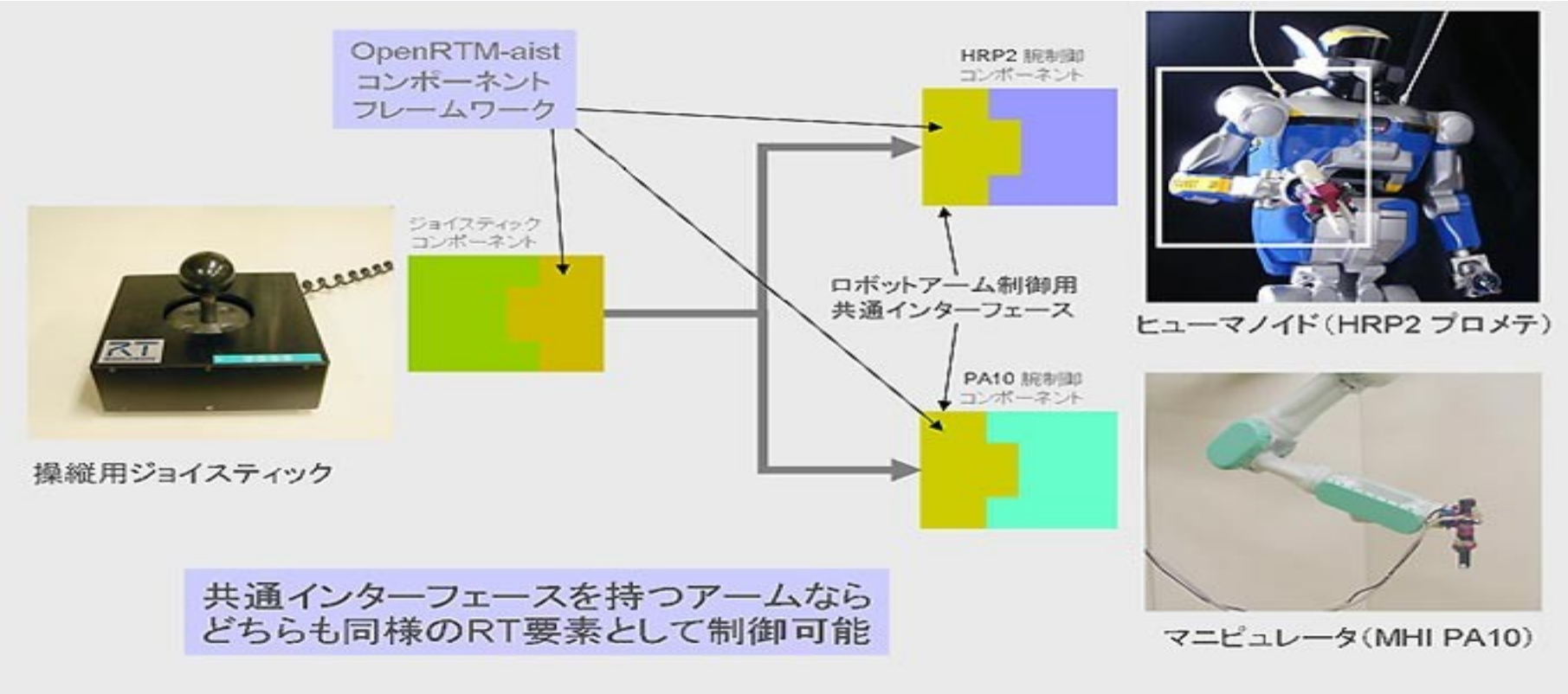


・FFで、シフトレジスタを構成
-FFなので 並列read可能
-FFなので、並列プリセット可能

※すべての情報が確定していれば、送信パケット生成は1クロック
- IPヘッダの最初の2バイトは、早期に、送出可能

ロボット用ミドルウェア(ROS, RTミドルウェア など)

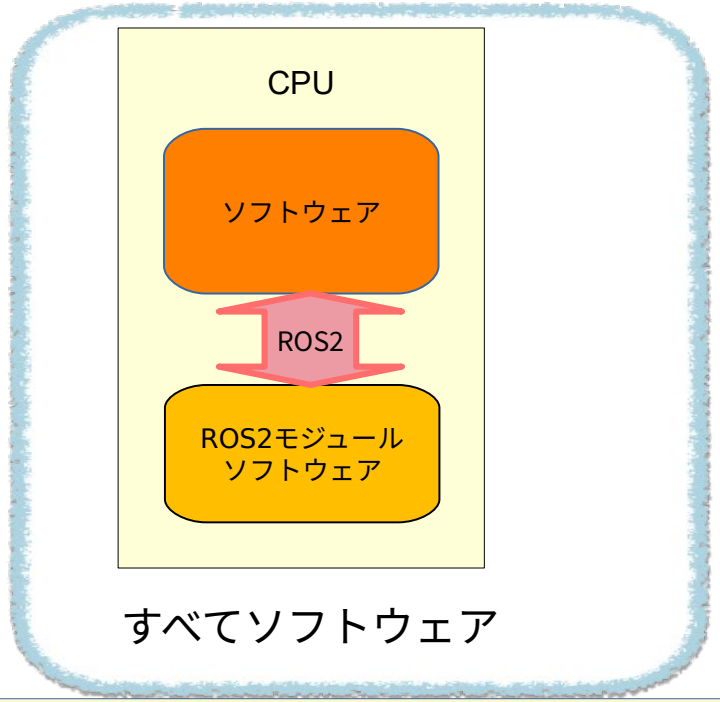
ROS ロボット・ミドルウェア = ソフトウェア・バス
ロボットのモジュールの流通性が高まる
日本のロボット業界は、ROSをデファクト・スタンダードにしようと活動している



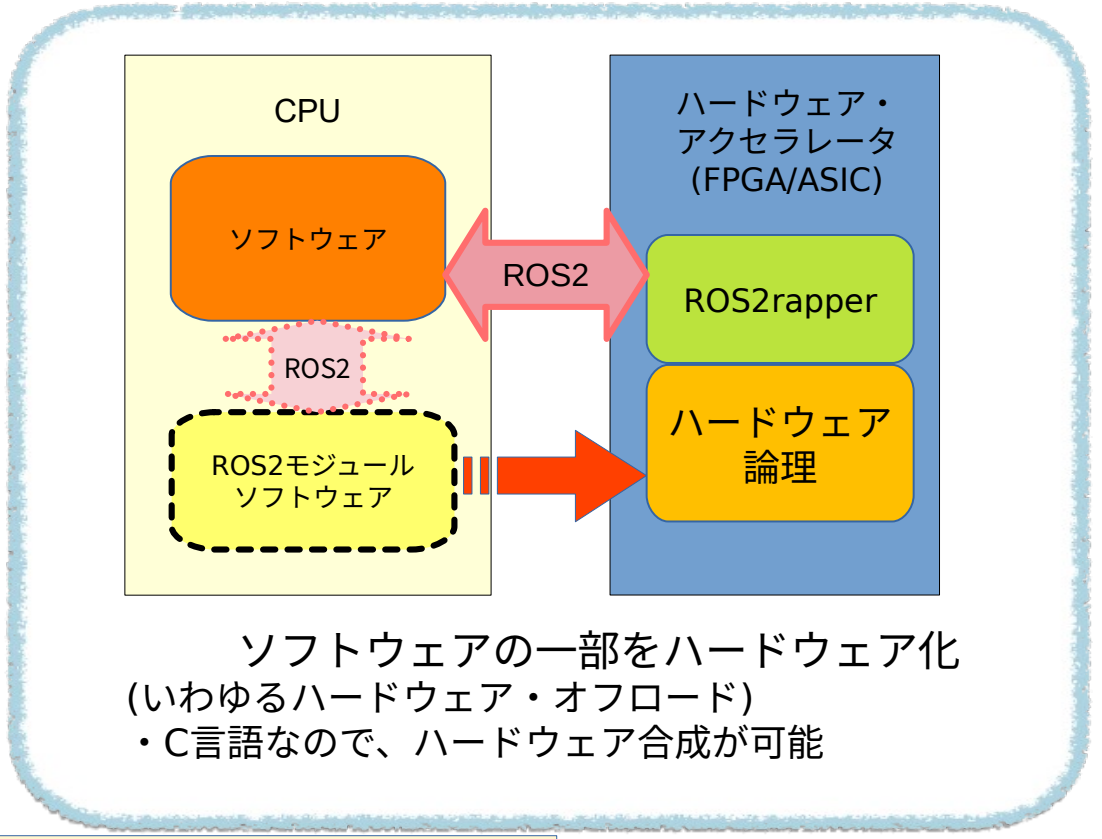
引用元 http://www.aist.go.jp/aist_j/press_release/pr2005/pr20050224/pr20050224.html

高位合成によりHW⇔SWの行き来が自在
そこで、ROS2便利

- HLS(C言語風高位記述言語)で書くと、どんな形でも、どこでも実行できる
- ROS2インターフェースで柔軟



すべてソフトウェア



ソフトウェアの一部をハードウェア化
(いわゆるハードウェア・オフロード)
・C言語なので、ハードウェア合成が可能

ROSインターフェース=インターフェース部分の書き換え不要
・デバイス・ドライバなどの開発が不要

ROS2って、Linuxが要るんじゃね?

- “ros2rapper”は、No OSだ!
- コンパクト
- 高速
- 堅牢
  - ハードウェア論理なので、ウイルスがつかない

ROS2プロトコルを完全ハードウェア化

XEでは、ROS2プロトコルを完全ハードウェア化した”ROS2rapper” . https://github.com/AXE-jp/ros2rapper

- ソフトウェア技術者がハードウェア論理を書き、LSI化。現実に!
- OSSとして、近日 配布開始
  - GPLと AXEプロプライエタリの、ダブル・ライセンスを検討中
- CPU無し,Linux無しで、ロボットの部品モジュールができる
  - センサとROS2プロトコルHWだけで、センサ・モジュール
  - PWMとROS2プロトコルHWだけで、アクチュエータ・モジュール
  - アプリケーションはC(HLS)言語で書いておけば、すぐハードウェア論理に合成
- ロボット部品が、ゴミのようなLSIでできる ← CPU不要
- CPU脳の敗北
- ハードウェアなので、堅牢 かつ 高速。そしてコンパクト

※ROS2ハードウェアには、コンフィギュレーション用のPROMがあることが望ましい

Arty A7-35ボード(xc7a35ti-csg324-1Lチップ)において

FPGA使用資源

- LUT: 33666
- FF: 13087

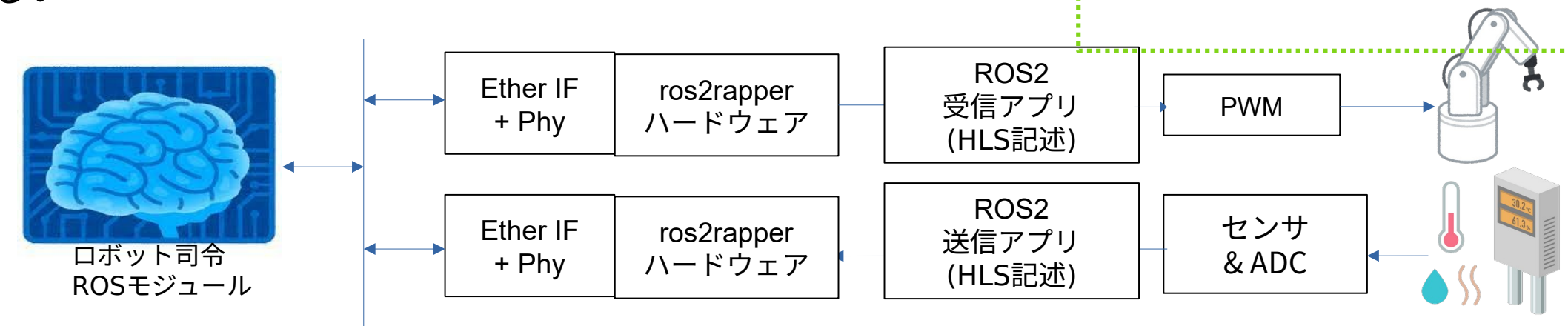
最大周波数: 121.01MHz

送信パケット生成 所要時間:

- IPデータグラム生成複数サイクル版:
  - 14サイクル=140nsec@100MHz
- IPデータグラム生成1サイクル板):
  - 8サイクル=160nsec@50MHz

受信 処理時間:

- 46サイクル=460nsec@100MHz



実現している機能

Table 1.1 サポートしている機能の一覧

機能		内容	備考
プロトコル	RTPS (SPDP /SEDP)	ROS2トピックのパブリッシュ	
		ROS2トピックのサブスクライブ	
	UDP/IP	UDPデータグラムの送信	
		UDPデータグラムの受信	
通信物理層	Ethernet	Ethernet層	Ethernet以外の物理層に差し替え可能
	ARP	物理アドレス解決	

- 各機能は、個別に有効化/無効化できる
  - ROS2トピックのパブリッシュ
  - ROS2トピックのサブスクライブ

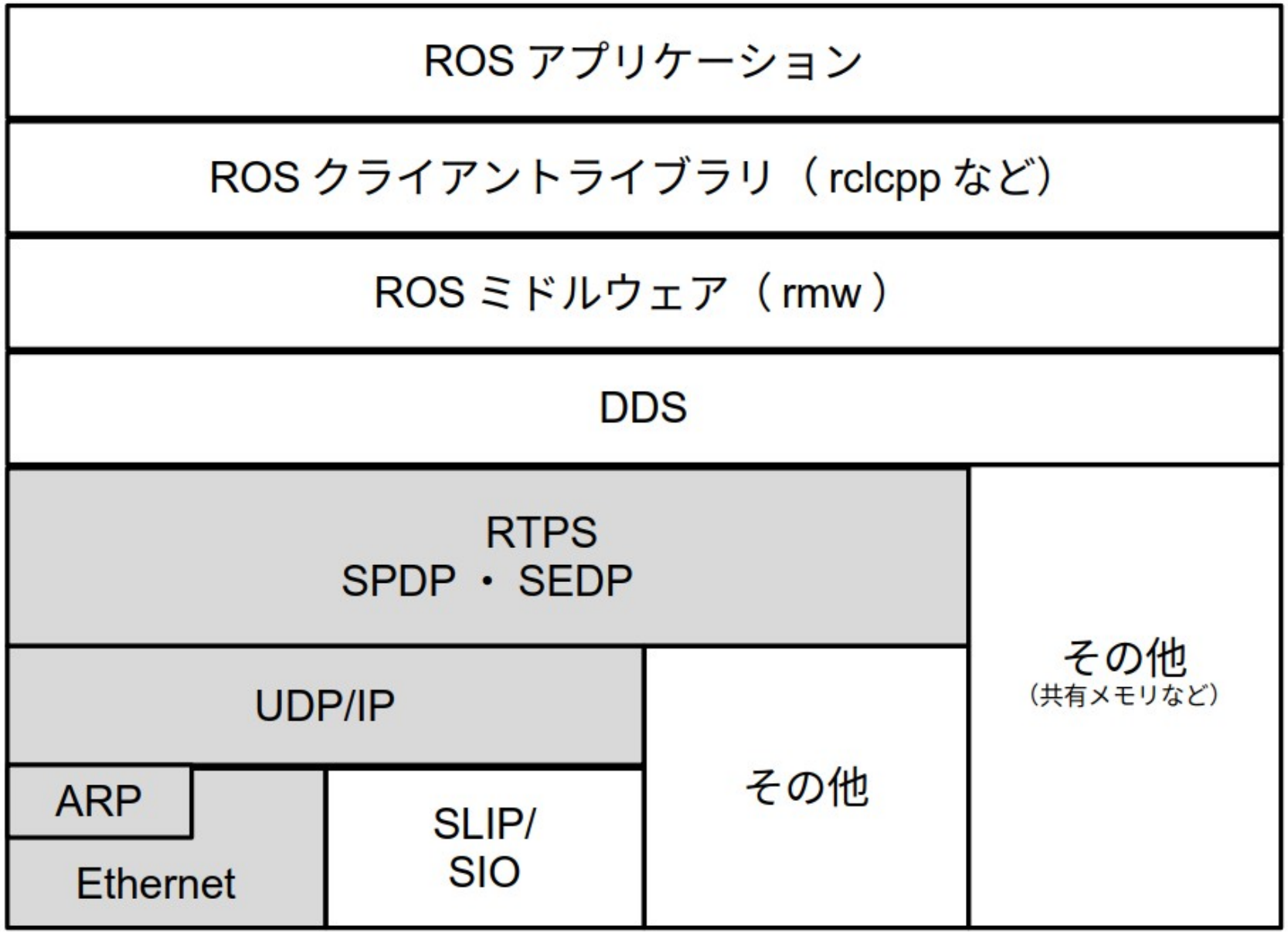


Fig.1.1 ROS2 アーキテクチャにおける ROS2rapper の位置

ROS2プロトコル

- RTPS
  - 低位プロトコル
- SPDP
  - 参加者 情報を送信
    - 信頼性が無い。マルチキャスト
  - 一定時間ごとに、マルチキャスト(ブロードキャスト)送信している

- SEDP
  - エンドポイント情報を送受信
    - ACK/NAKで信頼性がある。ユニキャスト
  - Heart beat送信
  - 詳細は次ページ

ROS2プロトコル: SEDP

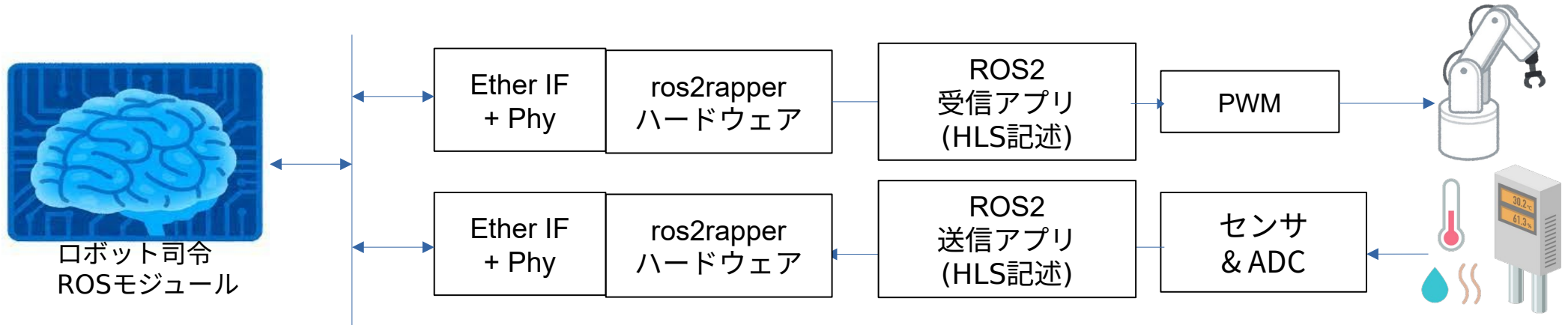
- SEDP
- 今回の試作はセンサ機器として開発
- 情報受信したいクライアント情報を受信し、管理
  - クライアント・テーブルを作る
  - 新しいクライアントは登録
- 情報送信は、クライアント・テーブルに登録されているクライアントへ順次
  - ラウンドロビン (アプリケーション層の仕事)

ros2rapper とユーザ・ロジックのインターフェース

- 主にレジスタ(D-FF)かポート(ワイヤ)で接続
- 1port RAM(など)をユーザ・ロジックで用意
  - D-FFで実現しても良い
  - ROS2は受信用バッファRAMのみ必要
  - UDPを使用する場合は、送受信用のRAMが必要
- Vivado, NEC CWBで合成可能

ROS2rapperと低消費電力 運用

- CPUは不要 (CPUはクロック停止/電源OFF)
- センサとROS2rapperだけ動かす
  - 周期的に、センサ・データを送出.数分おき～数時間おき
  - ROS2rapperも何もしないときは、クロック停止可能
- PWMとROS2rapperだけ動かす
  - 低クロック周波数で受信動作させる
  - 受信データでPWMを制御



低消費電力での、
分散エッジAI

低消費電力での、分散エッジAI

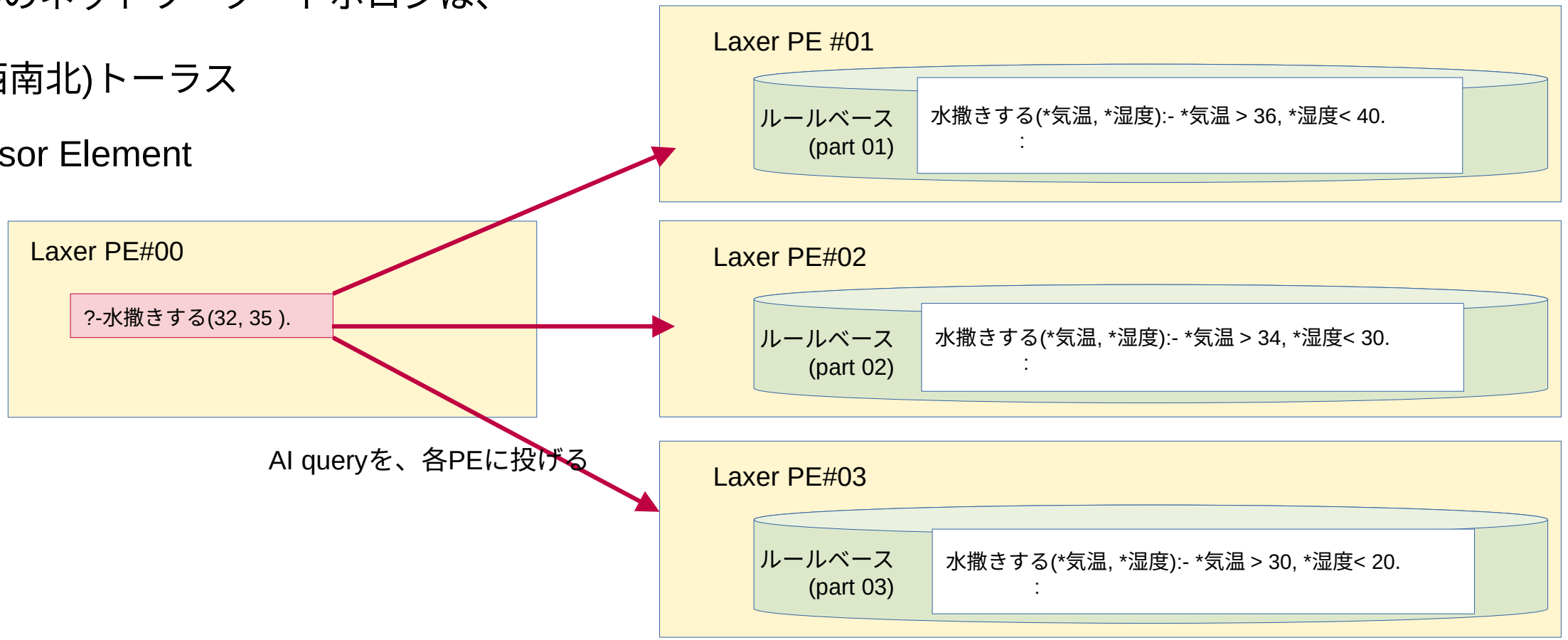
- 電力あたりの、推論能力は高い
  - 機械学習AI用の計算能力は高い方
- CPUなどのクロック周波数を動的に変更可能
  - エッジで、低速でよい場合は、低クロック周波数で、ゆっくり推論
- 論理推論のルールを、小さな単位にすれば小さいメモリで推論の実行が可能
- 小さい単位の推論を、並列に実行すれば、高速になる

小さなルール集合を、分散計算機で並列に推論する

- LaxerChipのネットワーク・トポロジは、

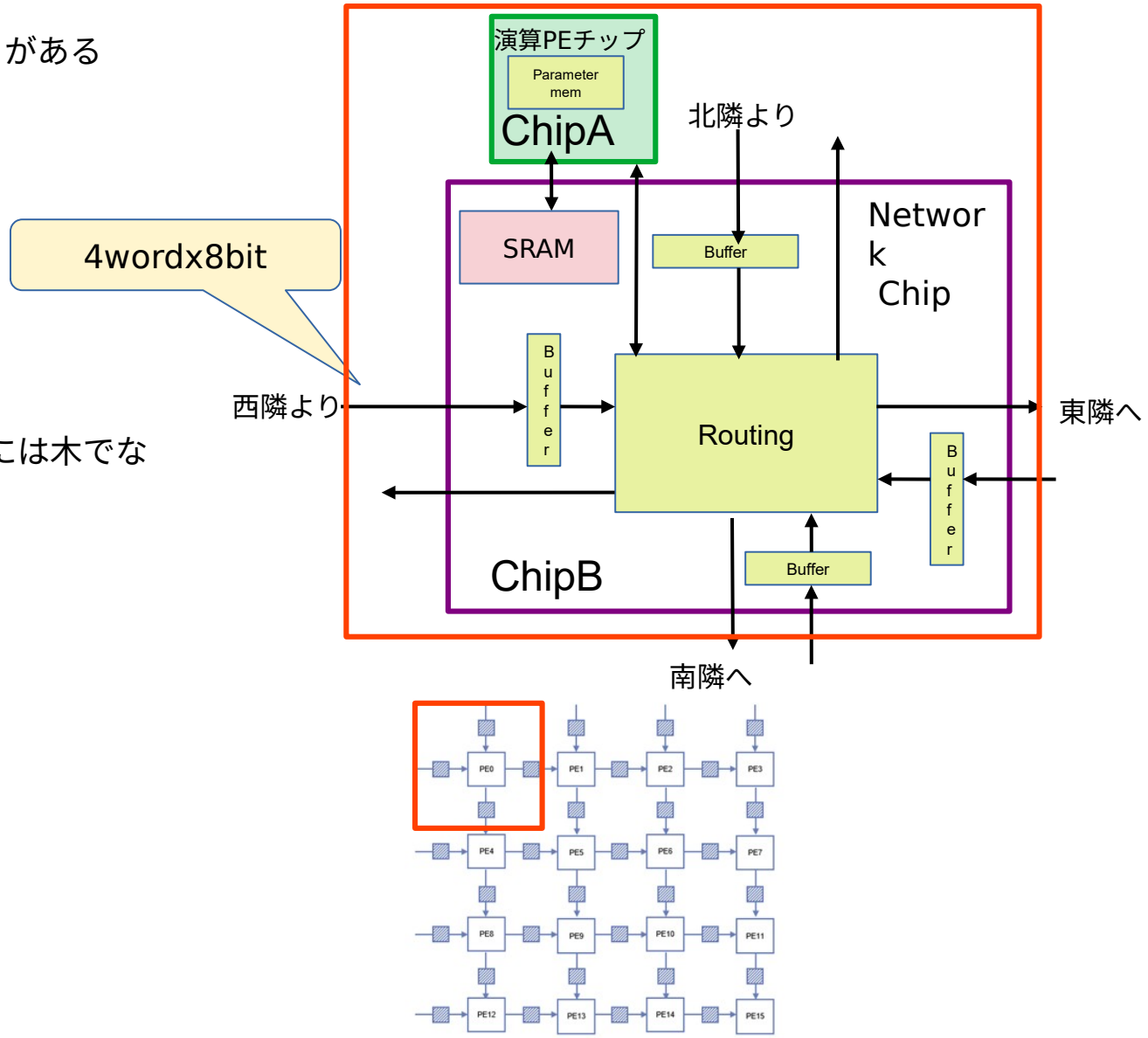
  NEWS(東西南北)トーラス

※PE=Processor Element



Laxerチップのネットワーク

- 通信 4方向シリアル通信、1方向につき入力/出力 各1本
- 通信部は、入力部に、1通信分=32bit(8bitx4word)のバッファがある
- パケットのデータ(ペイロード)サイズ
  - 4word x 8bit
- 論理的に望まれる、ネットワーク・トポロジ
  - シストリック・アレイ(6 or 4 本腕)
  - 木構造
    - 論理推論AIはこちらの方が使いやすい。が、物理的には木でなくてよい
- PE(演算ユニット)=Chip A
  - 機械学習PE
    - FP8 vector付き
  - 論理推論PE

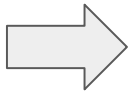


付録:
新命令による
論理推論AI(Prolog)を
高速化

GNU-PrologのWAMコード

- GNU-Prolog の WAM コードでは、switch_on_term で 1つめの arity の型(変数/アトム/整数/リスト/構造体)によって分岐する
- 分岐先で各arityの値をチェックする

```
foo(1, a, X) :- bar(1, X).
foo(a, [c,d], X) :- bar(2, X).
foo([a,b], c(d), X) :- bar(3, X).
foo(a(b), 1, X) :- bar(4, X).
```



```
predicate(foo/3,1,static,private,monofile,global,[
    switch_on_term(1,4,2,6,8),

label(1),
    try_me_else(3),

label(2),
    get_integer(1,0),
    get_atom(a,1),
    put_value(x(2),1),
    put_integer(1,0),
    execute(bar/2),

label(3),
    retry_me_else(5),

label(4),
    get_atom(a,0),
    get_list(1),
    unify_atom(c),
    unify_list,
    unify_atom(d),
    unify_nil,
    put_value(x(2),1),
    put_integer(2,0),
    execute(bar/2),

label(5),
    retry_me_else(7),

label(6),
    get_list(0),
    unify_atom(a),
    unify_list,
    unify_atom(b),
    unify_nil,
(以下略)
```

Pl_Switch_On_Term()の内容

ランタイム関数
Pl_Switch_On_Term() では、1つめのarityであるA(0)のタグ(下位3ビット)によって、次に実行する分岐先を返すようになっている

```
CodePtr FC
Pl_Switch_On_Term(CodePtr c_var,
          CodePtr c_atm, CodePtr c_int,
          CodePtr c_lst, CodePtr c_stc)
{
 WamWord word, tag_mask;
 CodePtr codep;

 DEREF(A(0), word, tag_mask);
 A(0) = word;

 if (tag_mask == TAG_INT_MASK)
   codep = c_int;
 else if (tag_mask == TAG_ATM_MASK)
   codep = c_atm;
 else if (tag_mask == TAG_LST_MASK)
   codep = c_lst;
 else if (tag_mask == TAG_STC_MASK)
   codep = c_stc;
 else                          /* REF or FDV */
   codep = c_var;

 return (codep) ? codep : ALTB(B);
}
```

RISC-V 64bit Gnu Prolog コンパイルド・バイナリとその高速化

gplcでコンパイルした native codeをobjdump -d の一部

```
00000000000096b8 <X0_ap__a3>:
    96b8: 00000517      auipc a0,0x0
    96bc: 02050513      addi a0,a0,32 # 96d8 <X0_ap__a3+0x20>
    96c0: 00000597      auipc a1,0x0
    96c4: 02458593      addi a1,a1,36 # 96e4 <X0_ap__a3+0x2c>
    96c8: 00000617      auipc a2,0x0
    96cc: 05460613      addi a2,a2,84 # 971c <X0_ap__a3+0x64>
    96d0: 014900ef      jal ra,996e4 <Pl_Switch_On_Term_Var_Atm_Lst>
    96d4: 00050067      jr a0
    96d8: 00000517      auipc a0,0x0
    96dc: 04050513      addi a0,a0,64 # 9718 <X0_ap__a3+0x60>
    976c: 00ab3823      sd a0,16(s6)
    9770: f49ff06f      j 96b8 <X0_ap__a3>
    9774: 00000013      nop
```

レジスタ間接によるレジスタ指定ができる新命令を追加

特にジャンプ命令

ロード命令、ストア命令

例えば、WAMコード switch_on_term をアセンブリ・コードにする場合に、型を示すタグ(3bit)によって、レジスタ間接によって指定されたレジスタの値(アドレス)にジャンプすると高速になる

すなわち

jal ra,Pl_Switch_On_Term@PLT ;サブルーチン・コールなので、とても遅い

jr a0

を、下記1命令で置き換えて、高速化

branch_reg_indirect IREG ; pc ← pc + xreg[IREG]

この2行の部分を、
あるレジスタ中の値で指定された、
別なレジスタ中の値を番地として、
jumpするようなコードに変更する
(コンパイラが新命令を使用するように変更)

新命令「 branch_reg_indirect %rax 」で置き換えて、高速化

Ruby,Haskell, Lispなど高級な言語は同様に高速化可能

```
%%% Prologソース
ap([],Y,Y).
ap([A|X],Y,[A|Z]) :- ap(X,Y,Z).
%% ap(X,Y,[a,b,c]).
```

参考:高速化対象: GNU-Prologの80x86(64bit)版オブジェクト(アセンブリ・コード)

この2行の部分を、
あるレジスタ中の値で指定された、
別なレジスタ中の値を番地として、
jumpするようなコードに変更する
(コンパイラが新命令を使用するように変更)
新命令「 branch_reg_indirect %rax 」
で置き換えて、高速化

```
X0_foo__a3:
        movq    Lpred1_1@GOTPCREL(%rip),%rdi
        movq    Lpred1_4@GOTPCREL(%rip),%rsi
        movq    Lpred1_2@GOTPCREL(%rip),%rdx
        movq    Lpred1_6@GOTPCREL(%rip),%rcx
        movq    Lpred1_8@GOTPCREL(%rip),%r8
        call    Pl_Switch_On_Term@PLT
        jmp     *%rax

Lpred1_1:
        movq    Lpred1_3@GOTPCREL(%rip),%rdi
        call    Pl_Create_Choice_Point3@PLT

Lpred1_2:
        movq    $15,%rdi
        movq    0(%r12),%rsi
        call    Pl_Get_Integer_Tagged@PLT
        test    %rax,%rax
        je      fail
        (略)
        jmp     X0_bar__a2@PLT

Lpred1_3:
        movq    Lpred1_5@GOTPCREL(%rip),%rdi
        call    Pl_Update_Choice_Point3@PLT

Lpred1_4:
        movq    ta@GOTPCREL(%rip),%rdi
        movq    0(%rdi),%rdi
        movq    0(%r12),%rsi
        call    Pl_Get_Atom_Tagged@PLT
        test    %rax,%rax
        je      fail
        (略)
        jmp     X0_bar__a2@PLT

Lpred1_5:
        movq    Lpred1_7@GOTPCREL(%rip),%rdi
        call    Pl_Update_Choice_Point3@PLT

Lpred1_6:
        movq    0(%r12),%rdi
        call    Pl_Get_List@PLT
        test    %rax,%rax
        je      fail
(以下略)
```

- GNU-Prolog のアセンブリコードでは、WAM(Prolog中間仮装マシン)のswitch_on_termに対応するランタイム関数 Pl_Switch_On_Term() を呼んでいる
- その返り値でジャンプしている

ここで、

- レジスタ間接によるレジスタ指定ができる新命令を追加
  - 特にジャンプ命令
  - ロード命令、ストア命令
- 例えば、WAMコード switch_on_term をアセンブリ・コードにする場合に、型を示すタグ(3bit)によって、レジスタ間接によって指定されたレジスタの値(アドレス)にジャンプすると高速になる
- すなわち

```
call    Pl_Switch_On_Term@PLT ;サブルーチン・コールなので、とても遅い
jmp     *%rax
```

- を、下記1命令で置き換えて、高速化

branch_reg_indirect IREG ; pc ← pc + xreg[IREG]

以上