Institute of SCIENCE TOKYO

RISC-Vシステム設計プラットフォーム

RISC-V 統合設計環境（IDE）デモンストレーション

東京科学大学　工学院　情報通信系

一色剛研究室

CEATEC 2024 : NEDOブース

Oct.15 ～ 18, 2024

RISC-Vオープンアーキテクチャ

- ✧ RISC-V : https://riscv.org/
  - ➢ ライセンスフリー命令セットアーキテクチャ（ISA : Instruction Set Architecture）
  - ➢ ローエンド組込み仕様からハイエンド（HPC・サーバー）仕様まで多様なISAサブセット
- ✧ RISC-Vの特長
  - ➢ ライセンスフリーのため、設計・製造コストを抑制可能
  - ➢ 複数の命令セットプロファイルや、命令拡張可能な仕様のため、差別化した製品が開発しやすく、カスタマイズ設計によるHWコストや消費電力の削減効果が大きい
  - ➢ 命令拡張可能なRISC-Vプロセッサと最適化設計された複数のHWアクセラレータからなるシステム構成 → SW定義による機能の柔軟性と高い処理効率を両立するための有力なソリューション
- ✧ RISC-Vの課題
  - ➢ 製品化の歴史が浅く、SW開発環境やHW設計プラットフォームの整備が遅れている
  - ➢ 実用アプリケーション用の省電力・セキュリティ・仮想化などの補助機能が不十分
  - ➢ RISC-V仕様は、欧米・中国の大手企業中心に進められ、国内メーカの貢献が少ない

RISC-Vシステム設計プラットフォーム（NEDO委託事業）

✧ RISC-Vシステム設計プラットフォーム概要

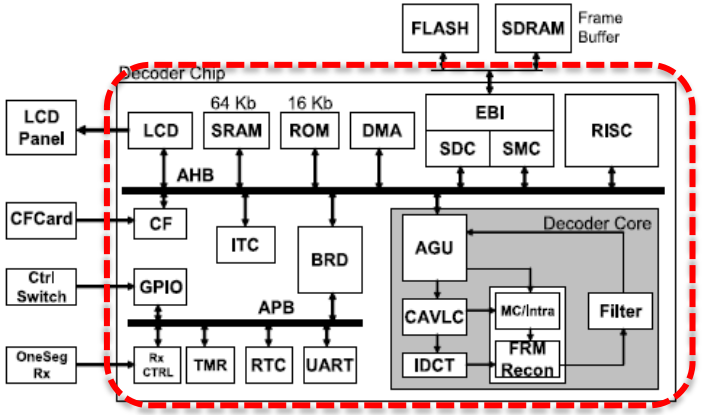
- 命令拡張可能なRISC-Vプロセッサと最適化設計された多数のハードウェアアクセラレータを組合わせたシステム全体のHW/SW設計検証を飛躍的に効率化
- **C2RTL高位システム設計検証ツール**：C++記述からの全システムのHW自動合成
- RISC-V統合開発環境（IDE）：RISC-V SoC のHW/SW協調設計検証環境（VS code活用）
- システム設計事例・チップ試作：**IoT/センサー/AI/セキュリティ**

✧ C2RTL高位システム設計検証ツール

- C++記述で、SoCの構成要素（プロセッサ、バス、HWアクセラレータ、周辺回路等）全ての回路を表現 → SoCのチップ全体をC++言語で設計可能（チップ試作実証済み）
- C++システムモデルで高速動作検証が可能 → システム開発スピードを大幅向上
- HW設計検証・デバッグをSW開発環境で実行 → HWデバッグ作業の大幅効率化

C/C++記述による次世代SoC統合設計検証環境

- ✧ **SW抽象度で全システムを見通す**
  設計の無駄を排除、容易な機能拡張、SW/HW境界のチューニング
- ✧ **SW記述によるHW-IPモデル**
  容易なESL開発フロー対応、容易なシステムインテグレーション、開発工数大幅削減
- ✧ **SW記述によるシステム検証**
  高速シミュレーション、容易な検証環境構築、SW開発環境下でHWデバッグ
- ✧ **システム設計の「見える化」:**
  SW記述（C/C++）による、全システムの動作検証モデル・論理合成モデルの統一的表現



次世代設計

SW記述による SoC全体の見える化

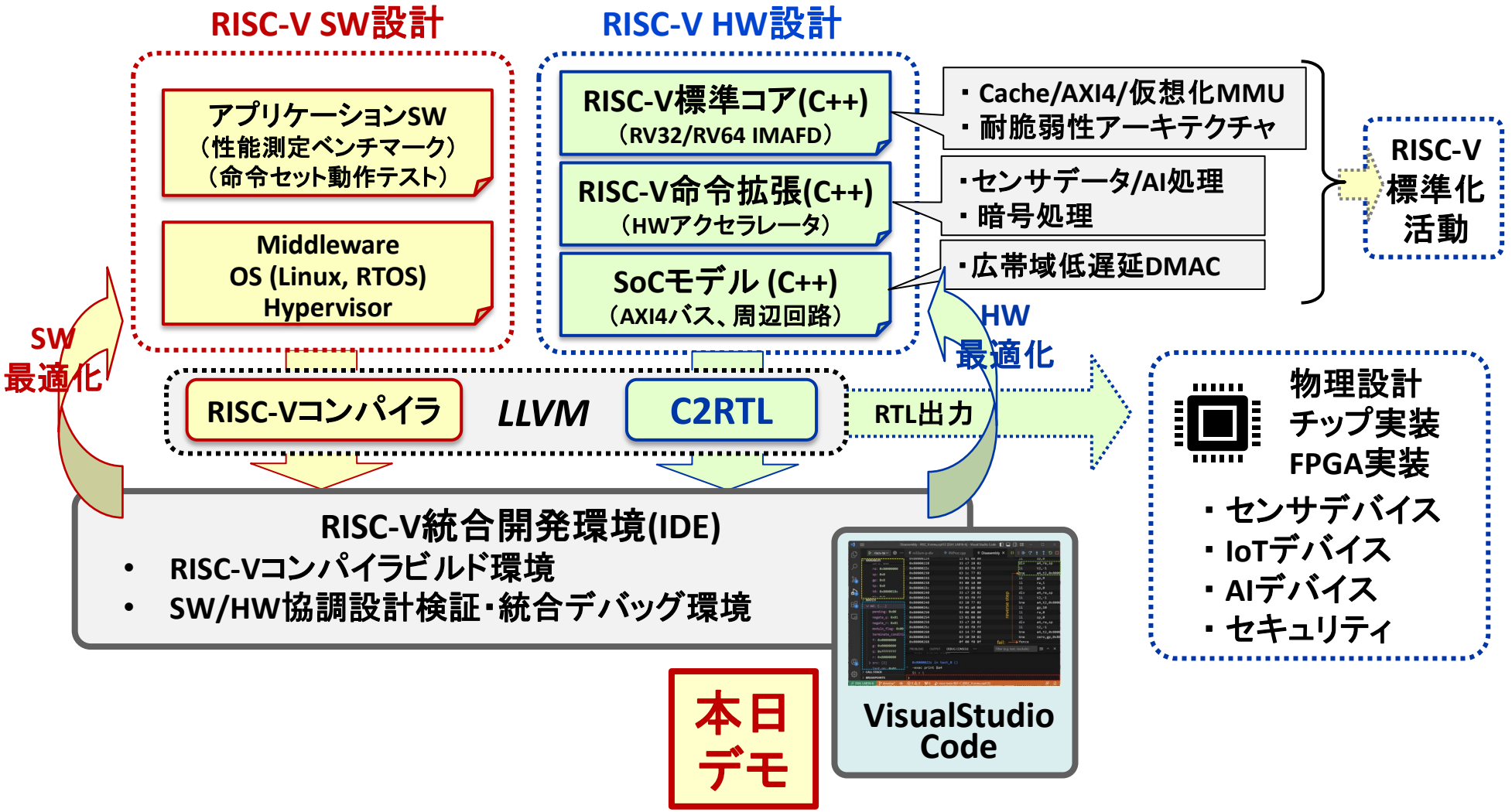
進化

本NEDOプロジェクトで実証！

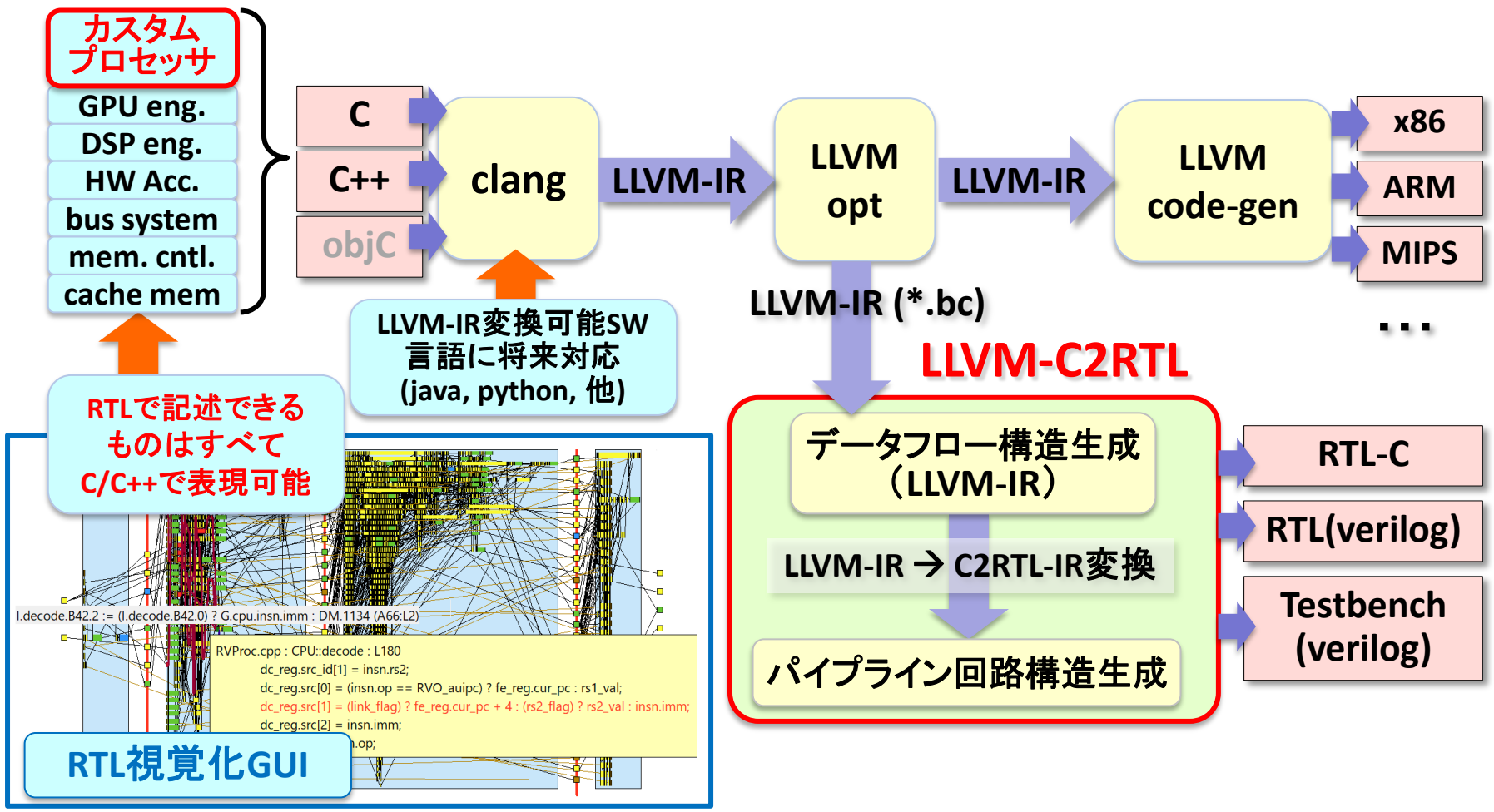
SW記述による SoC設計サインオフ とHW-IP流通

SoCをSW記述で表現！

RISC-Vシステム設計プラットフォームで実現される HW/SW協調設計検証メソドロジー



LLVM-C2RTLツール環境



Tokyo Tech

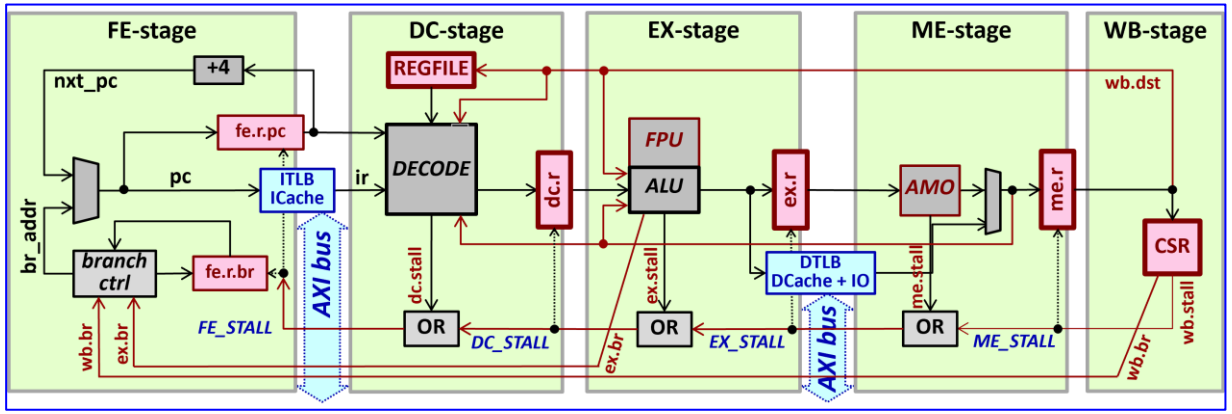
ライセンス
契約

New System Vision
Research and Development Institute
NSV財団

販売

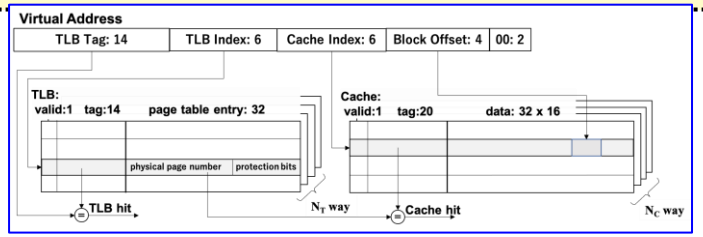
NSVT社/OTSL社

C++記述によるRISC-Vコアモデリング

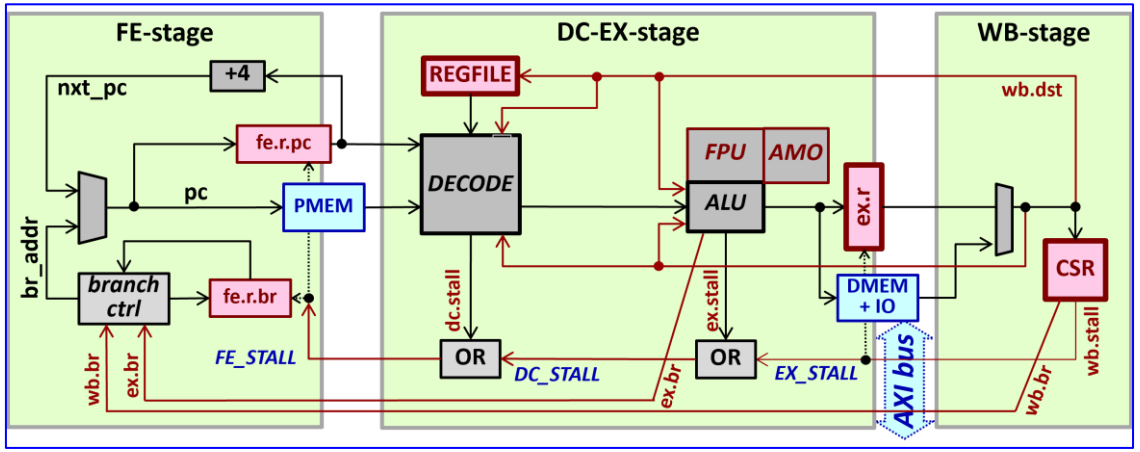


5-stage (Linux-OS enabled, MMU + Caches) : 740MHz @ 28nm

- MMU = TLB + page-walk logic (VIPT)
- I-Cache/I-TLB → AXI-Master port
- D-Cache/D-TLB/IO → AXI-Master port
- ISA : RV32/64-IMAFD (M/F/D : optional)
- Priviledge modes : U/M/HS/VU/VS



VIPT : Virtually-indexed physically tagged

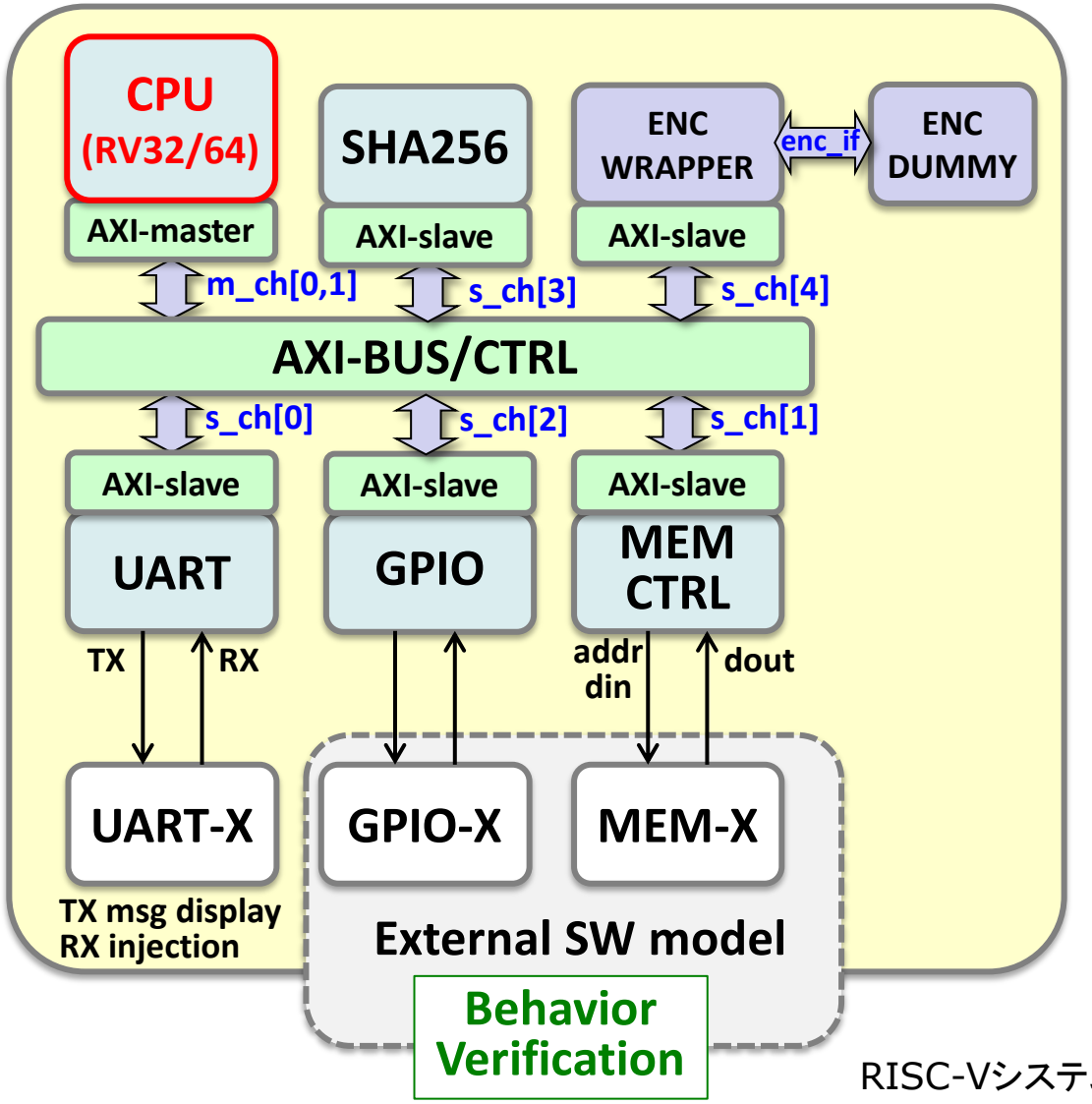


3-stage cache-less : 650MHz @ 28nm (est.)

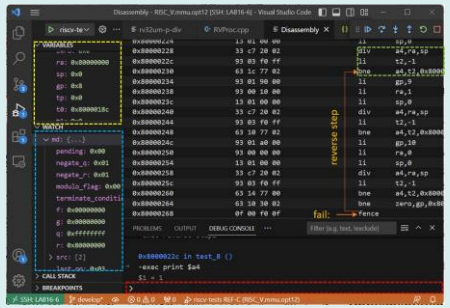
- For embedded apps: no cache, no MMU
- ISA : RV32-IMAFD (M/F/D : optional)
- Priviledge modes : U/M
- Higher IPC : no branch stalls

C2RTL : C++記述RISC-VモデルからRTLを自動生成 → チップ実装検証

RISC-V SoC 統合SW/HW開発検証環境



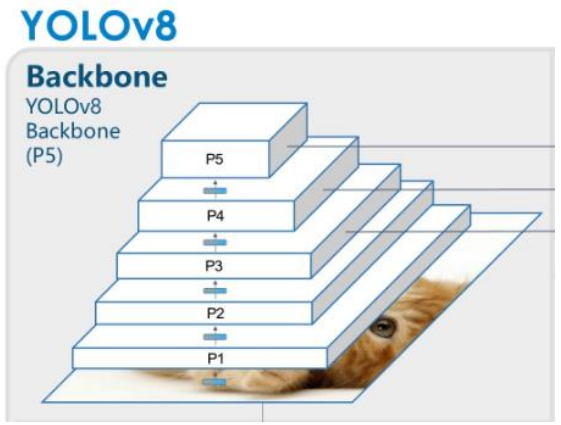
- RISC-V SoCシステム設計
  - C2RTL : C++記述 → HW記述（RTL）自動合成
  - SoC階層・IP階層の2階層記述構造
  - マルチクロック設計対応
- RISC-V SoC動作検証
  - C2RTL : RTL等価Cモデル自動生成による高速システム検証
  - VisualStudio Codes SWデバッグ環境 : RISC-V SoC動作の可視化 → 全システムのHW/SW協調デバッグを実現

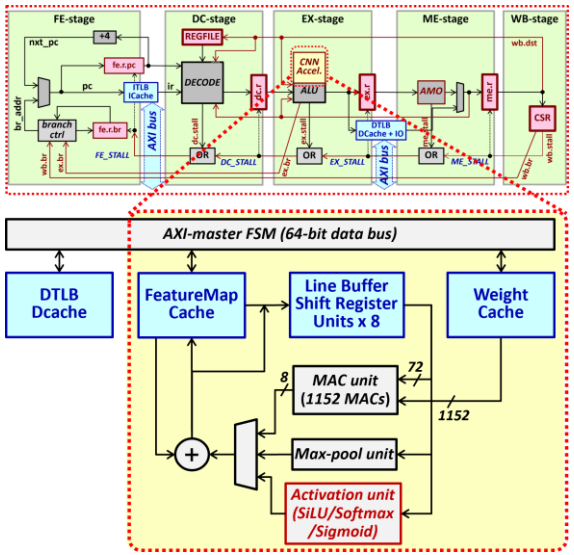


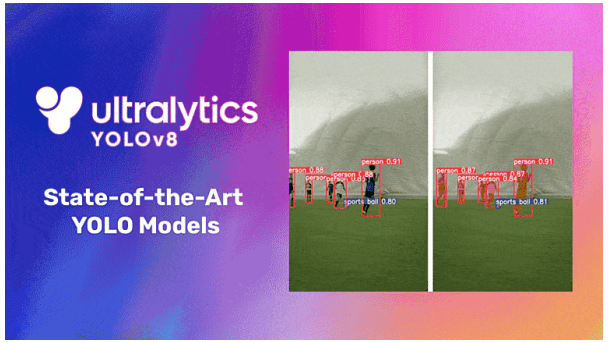
VisualStudio Code

RISC-V 命令拡張のAI適用事例

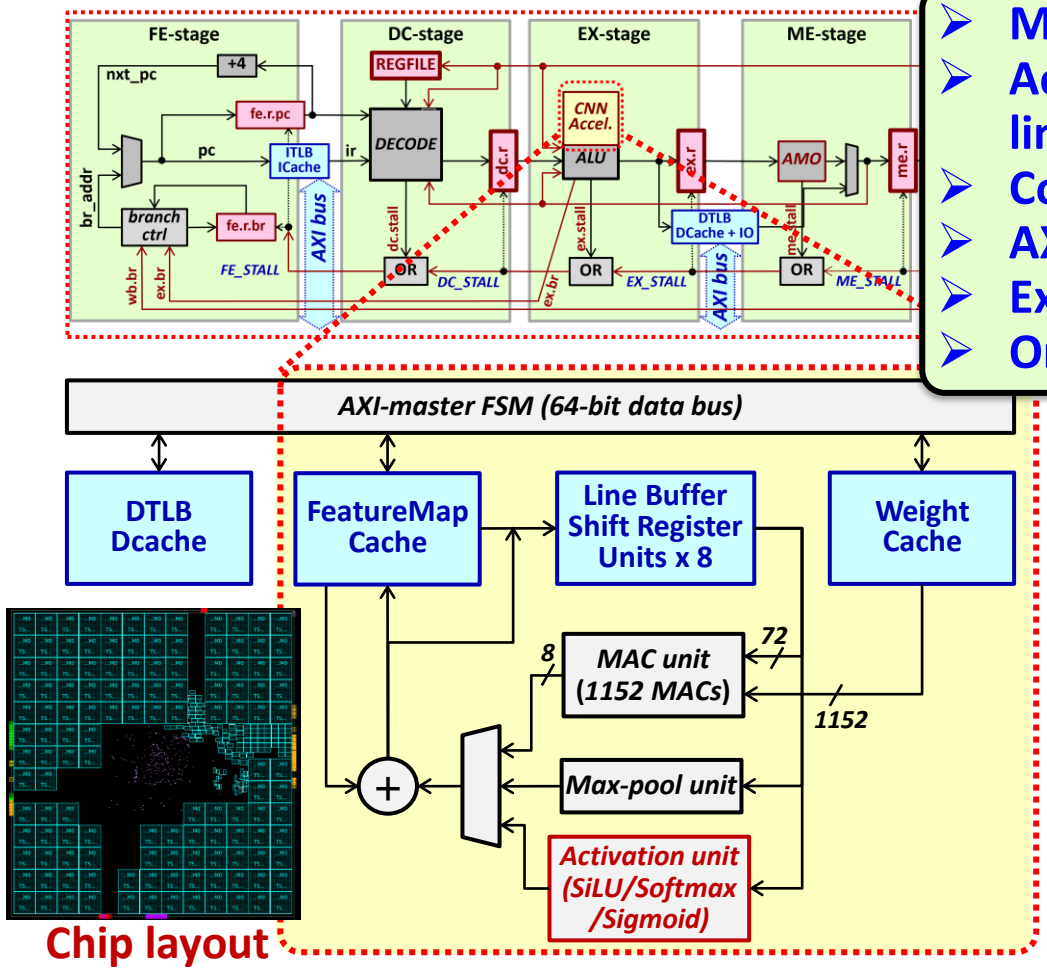
- ✧ **YOLOv8 : enhanced backbone network and features**
  - ➢ 機能 : detection, segmentation, pose estimation, tracking, classification
- ✧ **YOLOv8実装向けRISC-V 命令拡張**
  - ➢ HWアクセラレータを拡張命令でSW制御（全体処理の96.5%）
  - ➢ SW処理（全体処理の3.5%）: Non-maximum suppression, anchor box等







RISC-Vコア内搭載 YOLOv8アクセラレータ [3][4]



- MAC Array : 1152 MACs → 1.37 TOPs (peak) @ 595MHz
- Activation unit (SiLU, softmax, sigmoid) → hardwired piecewise linear approximation
- Core Layout @ 28nm → 202.5mW (6.76 TOPs/W peak), 7.9mm^2
- AXI-BUS : 64-bit data width
- External memory bandwidth : 4.76 GB/s (dedicated DMAC)
- On-chip memory bandwidth : 733 GB/s

Application	Ave. MAC util.	TOPs	TOPs/W
YOLOv8s	42.3%	0.58	2.85
YOLOv8s-seg	51.8%	0.71	3.50
YOLOv8s-pose	43.0%	0.59	2.90

vs. GTX1080Ti	Throughput	Energy consumption
YOLOv8s	0.133倍	1/166倍
YOLOv8s-seg	0.132倍	1/165倍
YOLOv8s-pose	0.136倍	1/170倍

[3] H. Wang, D. Li, T. Isshiki, "A power-efficient end-to-end implementation of YOLOv8 based on RISC-V", CAIT 2023

[4] H. Wang, D. Li, T. Isshiki, "Energy-Efficient Implementation of YOLOv8, Instance Segmentation, and Pose Detection on RISC-V SoC", IEEE Access (2024)

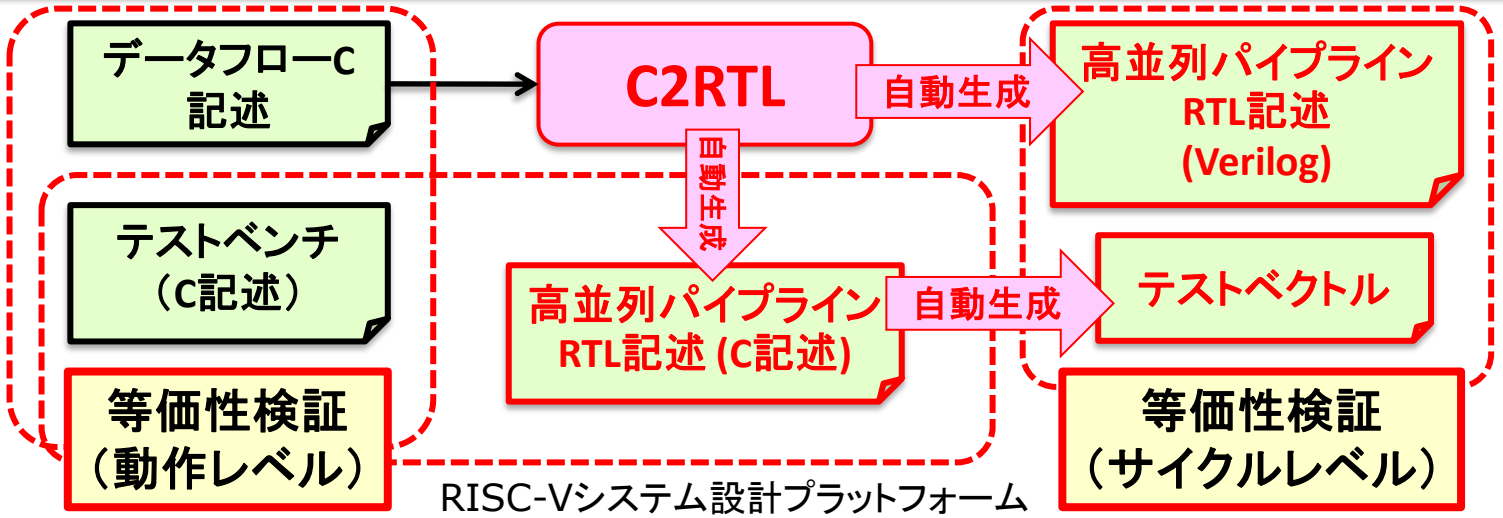
C2RTLシステム高位設計技術（東工大発技術）

C/C++記述によるRTL構造表現（データパス、FSM、サブシステム）

- ✧ データフローC/C++記述 → 1サイクル動作記述（後述）
- ✧ C/C++準拠（言語拡張なし） → 既存C/C++開発環境利用可
- ✧ HW属性記述（pragma/GCC-attribute）：ビット幅、レジスタ、メモリ
- ✧ System-level integration → 複数のHW-IPのシステム結合をC/C++記述のみで表現

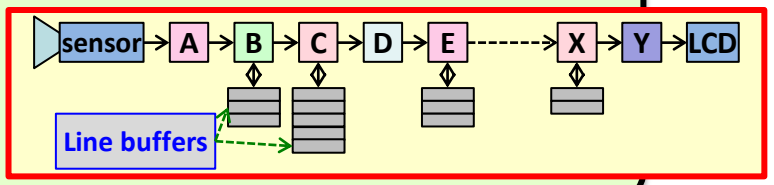
統合化された設計検証環境

- ✧ RTL等価Cモデル自動生成 → 元のC/C++開発環境でRTL検証可能
- ✧ RTL（Verilog）検証環境の自動生成（テストベクトル、テストベンチ）



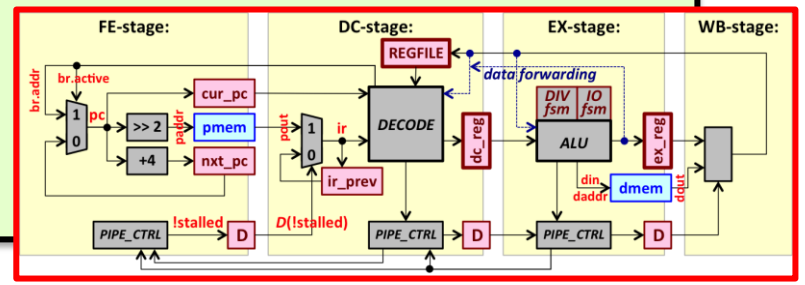
C/C++データフロー記述方式

- **データフロー型単純パイプライン記述方式 : 画像信号処理系、Deep Learning**
  - ✧ 高並列・パイプライン処理 → TeraOPS性能の小面積回路実装
  - ✧ パイプラインストール制御なし → 単純なC/C++データフロー記述
  - ✧ パイプライン段数調整・論理遅延平滑化 → ツールで自動化



ISP (Image Signal Processing) System

- **フロー制御付きパイプライン記述方式 : プロセッサ、制御系、通信系**
  - ✧ 複雑なフロー制御を「逆方向信号参照」で詳細に記述
  - ✧ プロセッサC++モデルで必須（ストール制御、data forwarding）
  - ✧ 詳細なパイプライン構造をC/C++データフロー記述で表現可能



- **いずれの記述方式 でもFSM記述は明示的に表現: 1サイクル記述**
  - ✧ 合成対象トップ関数を1回コールと、1サイクルのシステム動作が等価
  - ✧ 1サイクル内の状態更新（レジスタ、メモリ）は高々1回 → SW記述とRTL動作が1対1に対応

従来のHLS（高位合成）とC2RTLの違い

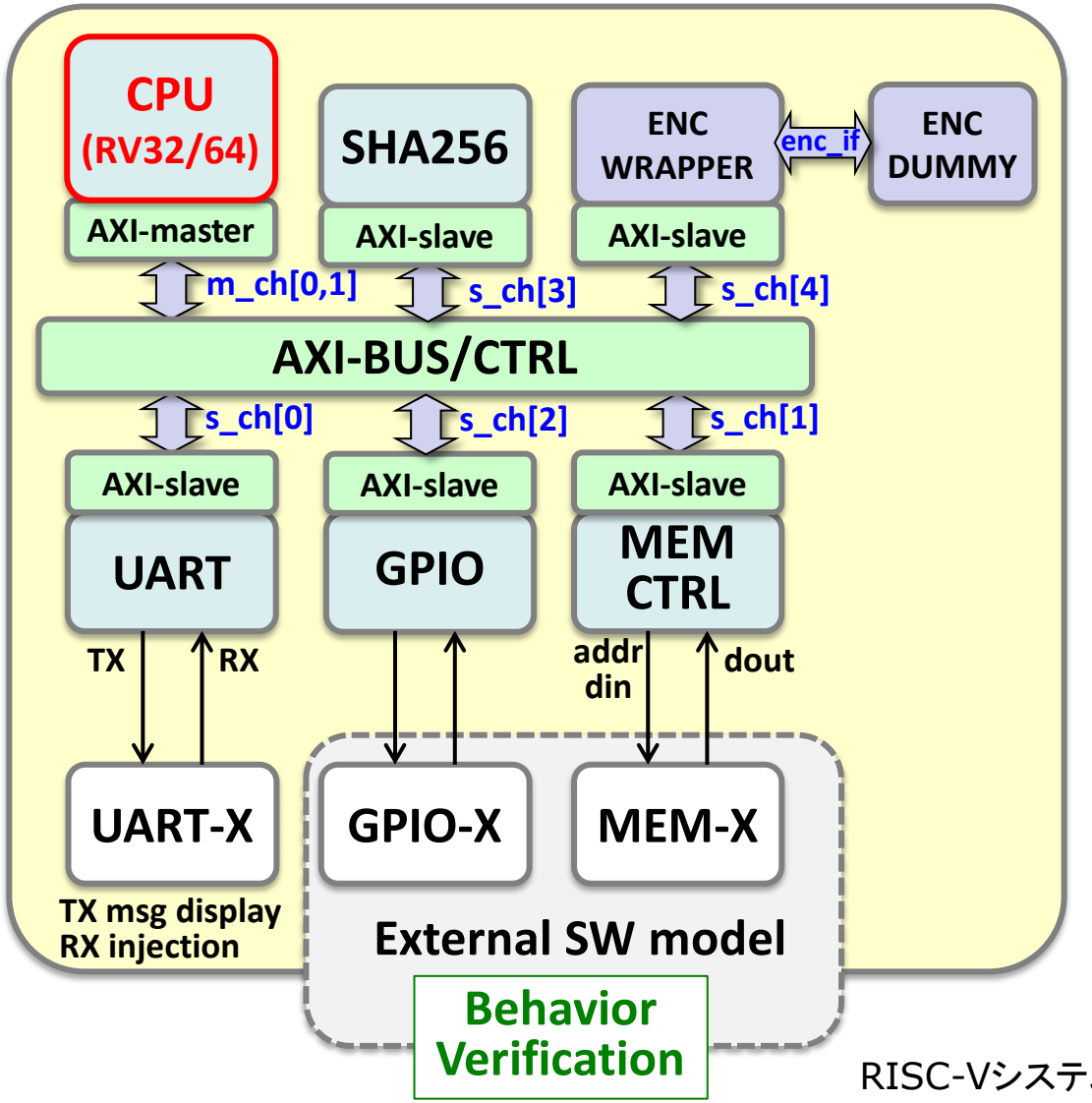
- **High-Level Synthesis : SW記述から多様なRTLアーキテクチャを自動合成**
  - ✧ SW記述の中間言語変換（CDFG等）→ Scheduling → Resource Binding → Datapath+FSM合成
  - ✧ リソース制約等により、回路規模/処理性能トレードオフに応じたアーキテクチャ合成
  - ✧ 記述スタイル、プラグマ指定子、高位合成内部処理、など個別HLSツールで大きく異なる
  - ✧ 高品質なRTL合成のためには、それなりのHLSツール熟練度・ノウハウが必要
  - ✧ System level integration : HLSで個別RTL合成後、RTLでシステム結合 → システムレベルRTL検証

- **C2RTL : RTLアーキテクチャそのものをC/C++で直接記述 (WYSWYG)**

What You See Is What You Get

  - ✧ SW記述の動作の抽象度：サイクルレベル（1サイクル記述）→ RTLと同じ動作抽象度
  - ✧ SW記述の中間言語変換（clang →llvm-ir）→ C2RTL-IR → RTL記述
  - ✧ 単純なRTL変換処理：関数コールの完全展開、ループの完全展開、変数代入MUX合成
  - ✧ 数千～数万演算の超並列処理が簡単に表現可能（Deep Learning、画像信号処理系）
  - ✧ 明示的FSM記述：バスインターフェースなどもC++で直接表現（プロセッサ）
  - ✧ System level integration : C++で直接システム結合 → システムレベル検証もC++で可能

RISC-V SoC モデリング階層（C++記述 → RTL自動合成）



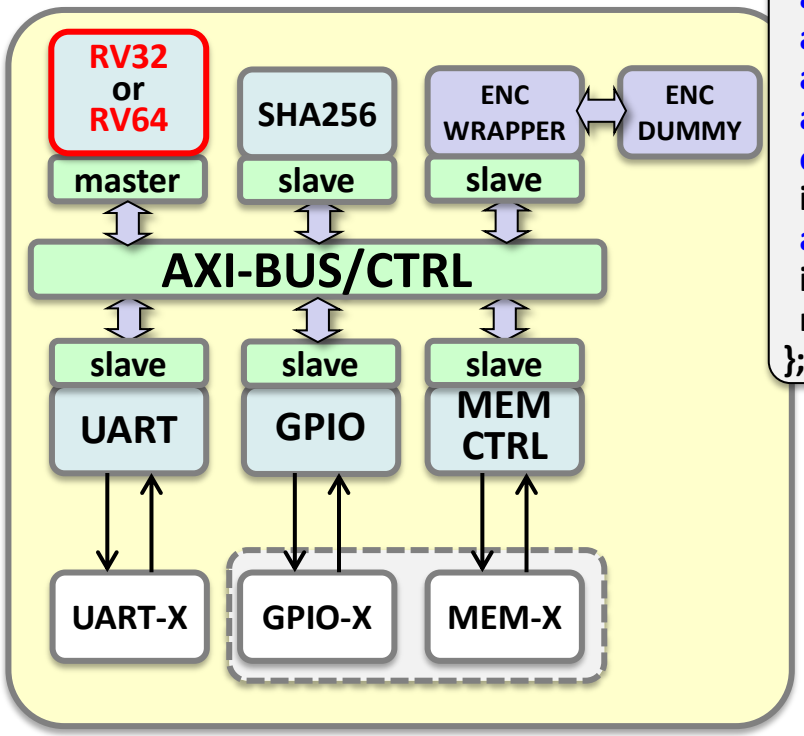
SoCのモデリング階層

- IP-レベル：個別RTL合成トップ関数
- SoCレベル：IP間接続

IPモデル群

- RISC-V(RV32/64)：キャッシュ/MMU付き、AXI-Master Port x2
- AXI-Slave devices
  - UART
  - GPIO
  - MEMCTRL
  - SHA256 HWアクセラレータ
  - ENC_WRAPPER/ENC_DUMMY→ 耐量子暗号モジュール統合用（マルチクロック設計）
- AXI-BUS/CTRL : 2-Master, 5-Slave

RV-SoC 最上位階層 C++ 記述




```
#define _C2R_MODULE  __attribute__((C2R_module))
_C2R_MODULE_
int RVProcAXI(IO_PINS* io_pins, AXI4L::BUS< 2, 5 >* axi_bus) {
  axi_uart.step (&axi_bus->s_ch[AXI4L::DI_UART], &io_pins->uart);
  axi_memctl.step (&axi_bus->s_ch[AXI4L::DI_EXT_MEM], &io_pins->mpin);
  axi_gpio.step (&axi_bus->s_ch[AXI4L::DI_GPIO], &io_pins->gpio);
  axi_sha256.step (&axi_bus->s_ch[AXI4L::DI_SHA256]);
  axi_enc_wrapper.step (&axi_bus->s_ch[AXI4L::DI_ENC], &io_pins->enc_if);
  enc_dummy.step (&io_pins->enc_if);
  int val = cpu1.step (&axi_bus->m_ch[0], &axi_bus->m_ch[1], axi_bus->s_ch[AXI4L::DI_UART].intr);
  axi_bus_ctrl.connectChannel (axi_bus);
  io_pins->uart.rx = uart_ext (io_pins->uart.tx);
  return val;
};
struct IO_PINS {
  UARTPin      uart;
  MEMCTLPin    mpin;
  GPIOPin      gpio;
  ENC_IF       enc_if;
} io_pins;
AXI4L::BUS< 2, 5 > axi_bus ;
int main (int argc, char * argv[]) {
   /// initialization codes here...
   while ( !RVProcAXI (&io_pins, &axi_bus)) {
     gpio_update (&io_pins.gpio) ;
     mem_ext.update (&io_pins.mpin) ;
   }
}
```

SoC階層属性

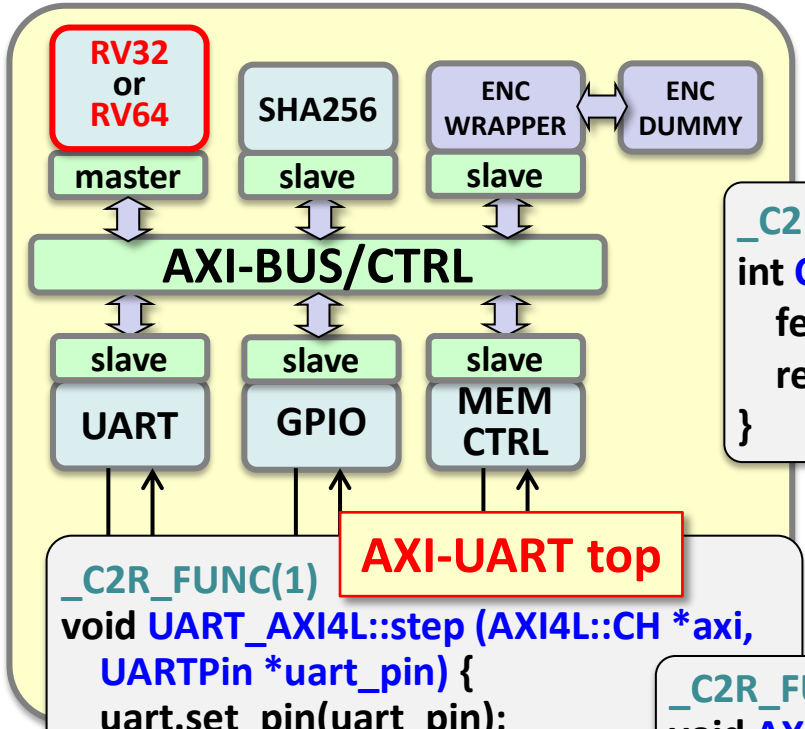
SoCレベル記述 : IPトップ関数の呼出し → IP接続記述

SoC IOピン定義・AXIバス定義 :
ユーザ定義クラスのみ
(Built-inクラスなし)

RV-SoCテストベンチ記述
（RV-SoCシミュレータ）

外部
SW モデル

RV-SoC IP階層 C++ 記述




```
#define _C2R_FUNC(N)  __attribute__((C2R_function(N)))
```

IP階層属性（個別RTL合成関数）

CPU top : RISC-V 5段パイプライン

```
_C2R_FUNC(5)
int CPU::step (AXI4L::CH *axi_d, AXI4L::CH *axi_i, unsigned intr) {
   fetch() ; decode(axi_i) ; execute(axi_d) ;  data_mem() ; writeback() ;
   return (cpu.halted == 1) ;
}
```

AXI-UART top

```
_C2R_FUNC(1)
void UART_AXI4L::step (AXI4L::CH *axi,
   UARTPin *uart_pin) {
   uart.set_pin(uart_pin);
   fsm(axi);
}
```

AXI-SHA256 top

```
_C2R_FUNC(1)
void SHA256_AXI4L::step (AXI4L::CH *axi) {
   fsm(axi);
}
```

AXI-BUS_CTRL top

```
_C2R_FUNC(1) template <int MC, int SC>
void AXI4L::CTRL<MC,SC>::
   connectChannel(BUS<MC,SC> *bus){ . . . }
```

AXIバス制御

AXI-MEMCTL top

```
_C2R_FUNC(1)
void MEMCTL_AXI4L::step (AXI4L::CH *axi,
   D_MEMCTLPin *mem_pin) {
   din = mem_pin->dout;
   mpin.set_outpin(mem_pin);
   fsm(axi);
}
```

AXI-GPIO top

```
_C2R_FUNC(1)
void GPIO_AXI4L::step(AXI4L::CH *axi,
   GPIOPin *gpio_pin) {
   gpim.set_pin(gpio_pin);
   fsm(axi);
}
```