C2RTL高位システム設計検証環境による RISC-V SoCの開発

RISC-V SoC Development on C2RTL System-Level Design Verification Framework

一色　剛 Tsuyoshi Isshiki

一般財団法人　新システムビジョン研究開発機構　代表理事
東京工業大学　工学院　情報通信系　教授

President, New System Vision Research and Development Institute
Professor, Tokyo Institute of Technology

April. 2021

説明概要 : Outline

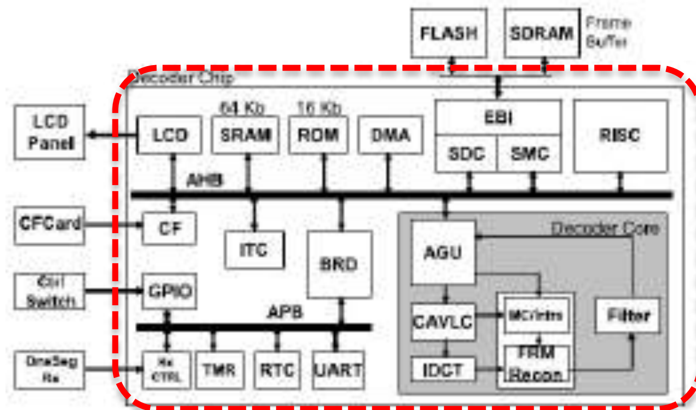
✧ **C2RTL高位システム設計環境のご紹介 :**
C2RTL System-Level Design Framework
- ➢ LLVM-C2RTLツール環境 : LLVM-C2RTL tool environment
- ➢ データフロー記述方式 : C/C++ Data Flow Coding Style
- ➢ 高位合成との相違点 : Comparison with HLS (High-Level Synthesis)

✧ **C2RTL設計開発事例 : C2RTL Design Examples**
- ➢ Deep Neural Network : Super-resolution CNN, Resnet-34
- ➢ RISC-V Processor : MMU/Cache, Linux verification, FPU, DNN extension

C/C++記述による次世代SoC統合設計検証環境

SoC Design Verification based on C/C++ Description

- ✧ **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

SW記述による
SoC全体の見える化

次々世代

SW記述による
SoC設計サインオフ
とHW-IP流通

RTLをSW記述で表現!

C2RTLシステム高位設計技術（東工大発技術）

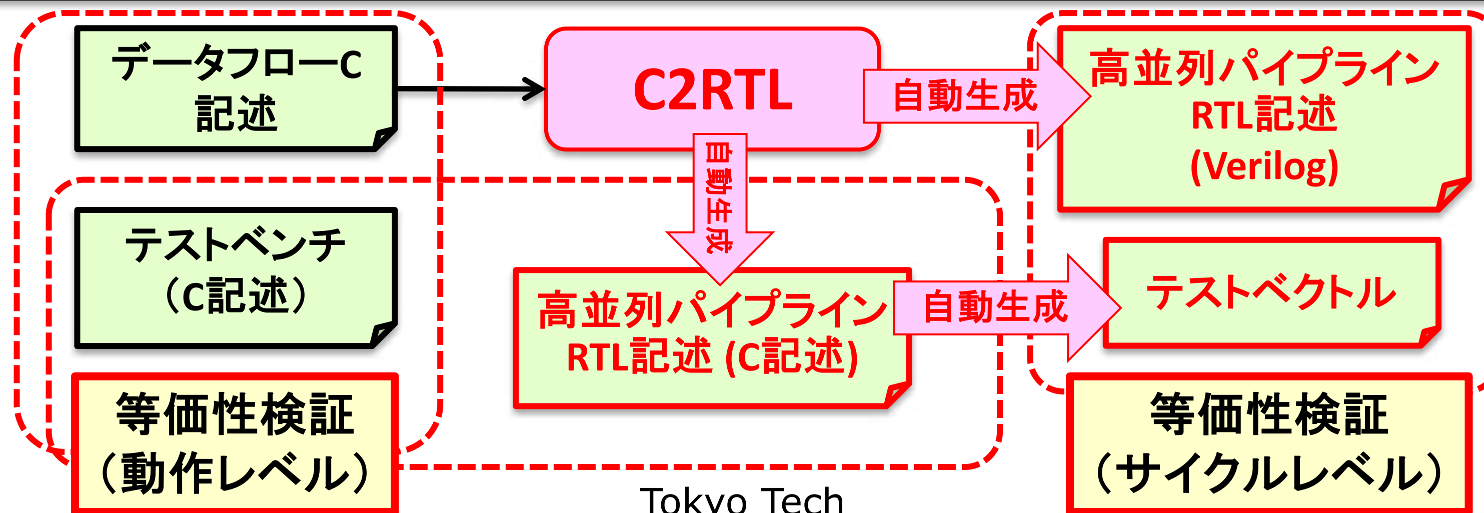
C2RTL System-Level Design Verification Methodology

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）



LLVM-C2RTLツール環境

LLVM-C2RTL Tool Environment

カスタムプロセッサ
GPU eng.
DSP eng.
HW Acc.
bus system
mem. cntl.
cache mem

C
C++
objC

clang

LLVM-IR

LLVM opt

LLVM-IR

LLVM code-gen

x86
ARM
MIPS
…

LLVM-IR変換可能SW言語に将来対応 (java, python, 他)

LLVM-IR (*.bc)

LLVM-C2RTL

データフロー構造生成（LLVM-IR）

LLVM-IR → C2RTL-IR変換

パイプライン回路構造生成

RTL-C
RTL(verilog)
Testbench (verilog)

RTLで記述できるものはすべてC/C++で表現可能

RTL視覚化GUI

ライセンス契約

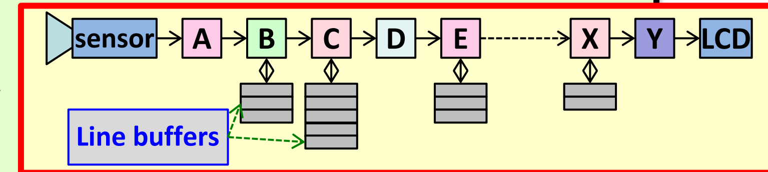
販売契約

株式会社ユーリカ

C/C++データフロー記述方式

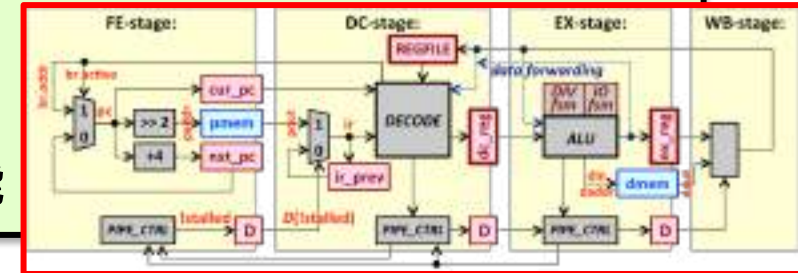
C/C++ Data Flow Coding Style

- **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

Comparison with HLS (High-Level Synthesis)

- **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

What You See Is What You Get

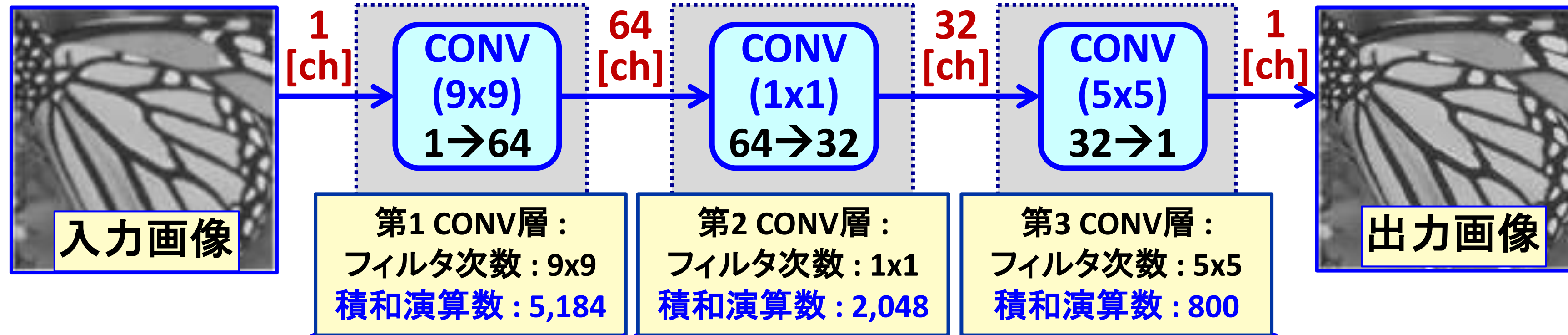
- **C2RTL : RTLアーキテクチャそのものをC/C++で直接記述 (WYSWYG)**
  - ✧ SW記述の動作の抽象度:サイクルレベル(1サイクル記述) → RTLと同じ動作抽象度
  - ✧ SW記述の中間言語変換(clang →llvm-ir) → C2RTL-IR → RTL記述
  - ✧ 単純なRTL変換処理:関数コールの完全展開、ループの完全展開、変数代入MUX合成
  - ✧ 数千～数万演算の超並列処理が簡単に表現可能(Deep Learning、画像信号処理系)
  - ✧ 明示的FSM記述:バスインターフェースなどもC++で直接表現(プロセッサ)
  - ✧ System 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

説明概要 : Outline

- ✧ **C2RTL高位システム設計環境のご紹介 :**
  C2RTL System-Level Design Framework
  - ➢ LLVM-C2RTLツール環境 : LLVM-C2RTL tool environment
  - ➢ データフロー記述方式 : C/C++ Data Flow Coding Style
  - ➢ 高位合成との相違点 : Comparison with HLS (High-Level Synthesis)

- ✧ **C2RTL設計開発事例 : C2RTL Design Examples**
  - ➢ Deep Neural Network : Super-resolution CNN, Resnet-34
  - ➢ RISC-V Processor : MMU/Cache, Linux verification, FPU, DNN extension

Super-Resolution Convolutional Neural Network (SRCNN)

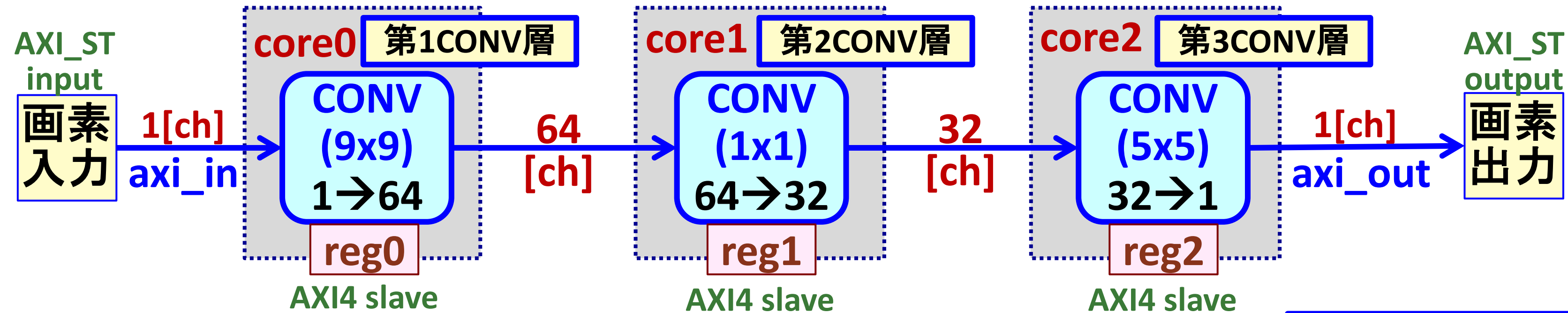


- 8,032 MACs/pixel, 16,064 Ops/pixel
- 2.4 TeraOPS/sec @150MHz (75 FPS @ 2K x 1K pixels)

- 7-stage pipeline : 8,032 MAC units (9b x 11b integer)
- 6.31 Million Gates (概算値) → 1 MAC演算器当り 786 gates

"Image super-resolution using deep convolutional networks", Dong et al (2016)

Super-Resolution CNN（SRCNN）C++ Resource Description




```
template < typename T, typename CT, int ISZ, int F1SZ, int L1SZ, int F2SZ, int L2SZ, int F3SZ,
           int OSZ, int SB1, int SB2, int SB3, int CBW, int BBW, int P1BW, int P2BW >

struct SRCNN_TOP {
  SRCNN_REG_AXI4L <CT, ISZ, F1SZ, L1SZ, CBW, BBW>     reg0; /// AXI4 slave
  SRCNN_REG_AXI4L <CT, L1SZ, F2SZ, L2SZ, CBW, BBW>    reg1; /// AXI4 slave
  SRCNN_REG_AXI4L <CT, L2SZ, F3SZ, OSZ, CBW, BBW>     reg2; /// AXI4 slave

  CNN_CONV_CORE   <T, CT, ISZ,    F1SZ, L1SZ, SB1, CBW, BBW, P1BW>  core0;
  CNN_CONV_CORE   <T, CT, L1SZ, F2SZ, L2SZ, SB2, CBW, BBW, P2BW>  core1;
  CNN_CONV_CORE   <T, CT, L2SZ, F3SZ,  OSZ, SB3, CBW, BBW, P2BW>  core2;

  AXI_ST::SlaveFSM    in_sif;      /// AXI_ST input (slave interface)
  AXI_ST::MasterFSM out_mif;     /// AXI_ST output (master interface)

  void Run(AXI4L::CH *axi_rif0, AXI4L::CH *axi_rif1, AXI4L::CH *axi_rif2,
    AXI_ST::CH *axi_in, AXI_ST::CH *axi_out); /// SRCNN top function
};
```

CNNモデルパラメータ（チャネル数、フィルタ次数、ビット幅）

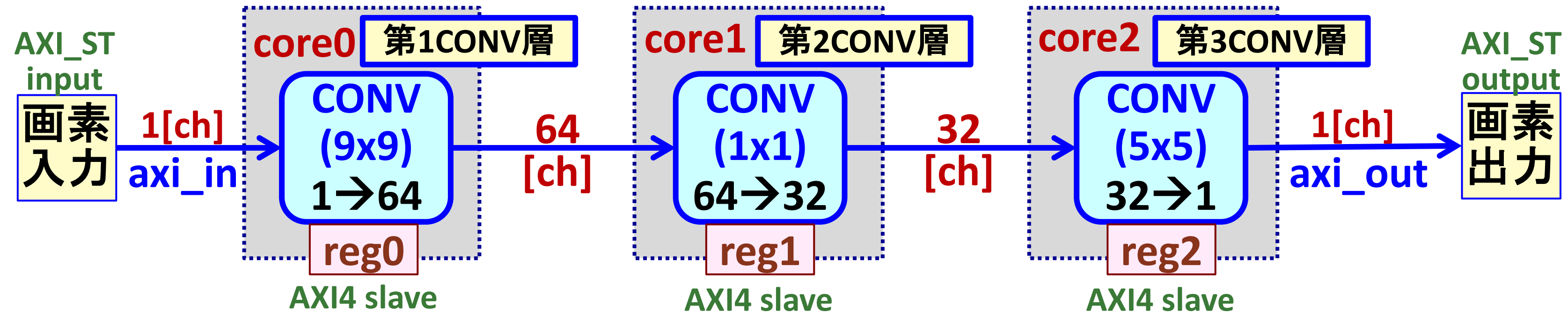
CONV係数レジスタI/F

CONV処理部

AXI-Stream入出力I/F

SRCNN動作記述

SRCNN C++ Data Flow Behavior Description




```
template < . . . >  void SRCNN_TOP< . . . > :: Run (
AXI4L::CH *axi_rif0, AXI4L::CH *axi_rif1, AXI4L::CH *axi_rif2,
AXI_ST::CH *axi_in, AXI_ST::CH *axi_out) {
        int reset = reg0.reset && reg1.reset && reg2.reset;
        int initialized = reg0.initialized && reg1.initialized && reg2.initialized;
        T v0[ISZ], v1[L1SZ], v2[L2SZ], v3[OSZ];
        BIT vld0 = in_sif.read( axi_in, initialized, &v0 );                    // 画素入力
        BIT vld1 = core0.Run( v0, v1, reg0, reset && !initialized, vld0, 0 );  // 第1CONV層
        reg0.fsm(axi_rif0);
        BIT vld2 = core1.Run( v1, v2, reg1, reset && !initialized, vld1, 0 );  // 第2CONV層
        reg1.fsm(axi_rif1);
        BIT vld3 = core2.Run( v2, v3, reg2, reset && !initialized, vld2, 0 );  // 第3CONV層
        reg2.fsm(axi_rif2);
        out_mif.write(axi_out, vld3, v3[0] );                                  // 画素出力
    }
};
```

CONV-Layer C++ Data Flow Behavior Description

```
template < typename T, typename CT, int ISZ, int FSZ, int OSZ, int SBITS,
           int CBW, int BBW, int PBW >   struct CNN_CONV_CORE
{
  VectorLineBuffer <T, ISZ, FSZ, MAX_WIDTH, PBW>  lbuf;
  VectorShiftRegWindow <T, ISZ, FSZ, FSZ, PBW>      srw;
  int Kernel(T *sr, T *weight);
  BIT Run ( T din[], T dout[], SRCNN_REG_AXI4L<CT, ISZ, FSZ, OSZ, CBW, BBW> &reg,
        int init, int invalid ) {
    ...
    lbuf.UpdateData101(din, lb1, reg.width, reg.height, lbvld);
    srw.UpdateData101(lb1, lbvld, inValid, sr1, srvld, px, reg.width);
    outValid = srvld[FSZ*FSZ / 2];
    for (int i = 0; i < OSZ; i++) {   /// 出力チャネル forループ
     if (outValid) {
       T sum = 0;
       for (int j = 0; j < ISZ; j++) {   /// 入力チャネル forループ
        sum += Kernel(&sr1[j*FSZ*FSZ], &reg.weight[(i*ISZ + j)*FSZ*FSZ]);
       }
       sum = DESCALE(sum, SBITS) + reg.bias[i];
       dout[i] = ReLU(sum);  /// ReLU
     }
     else { dout[i] = 0; }
    }
  }
};
```

Line Buffer
Shift Register

CONV
(FSZxFSZ)
ISZ→OSZ

ISZ : Input Channel #
OSZ : Output Channel #
FSZ : Filter size

```
int Kernel(T *sr, T *weight) {
   T sum = 0;
   for (int i = 0; i < FSZ*FSZ; i++) {
      sum += sr[i] * weight[i];
   }
   return sum;
}
```

CONV
Kernel
Coding

ハードウェア合成対象記述では
全てのループが完全展開される
→ ISZ x OSZ個のCONV回路が生成

Resnet-34 C++ Description

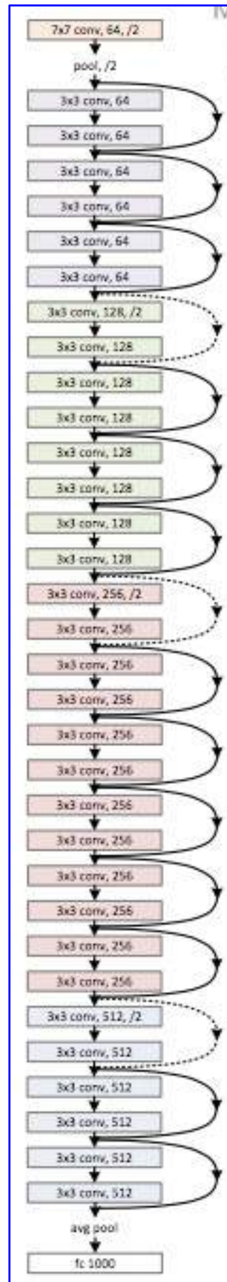
```
//// LayerGroup(0) :
CNN_CONV_Stride2x2  < 4,STy,UTy, 1,   3,7,  64,WB1,WB1,FB0,FB1>      L0(0);
CNN_MAX_POOL_Stride2x2  < 4,STy,UTy, 2,    3,  64,          FB1>      L1(1);
//// LayerGroup(1) :
CNN_CONV_BYPASS     < 4,STy,UTy, 4,  64,3,  64,WB1,WB1,FB1,FB1>    L2(2);
CNN_CONV            < 4,STy,UTy, 4,  64,3,  64,WB1,WB1,FB1,FB1>    L3(3);
. . . . .
//// LayerGroup(16) :
CNN_CONV_BYPASS     < 4,STy,UTy,32, 512,3, 512,WB4,WB4,FB4,FB4> L35(35);
CNN_CONV            < 4,STy,UTy,32, 512,3, 512,WB4,WB4,FB4,FB4> L36(36);
//// LayerGroup(17) :
CNN_AVE_POOL        < 4,STy,UTy,32, 512,           FB4,FB4>         L37(37);
CNN_LINEAR     < 4,STy,UTy,32, 512, 512,1000,1,4,WB5,WB5,FB4,FB5> L38(38);
```

	入力サイズ	出力サイズ	層数	Kernel
①	224^2 x 3	112^2 x 64	1	7x7 CONV
	112^2 x 64	56^2 x 64		3x3 max pool
②	56^2 x 64	56^2 x 64	2 x 3	3x3 CONV
③	28^2 x 128	28^2 x 128	2 x 4	3x3 CONV
④	14^2 x 256	14^2 x 256	2 x 6	3x3 CONV
⑤	7^2 x 512	7^2 x 512	2 x 3	3x3 CONV
⑥	512	1000	1	Linear (fully-connnected)
Total			34	

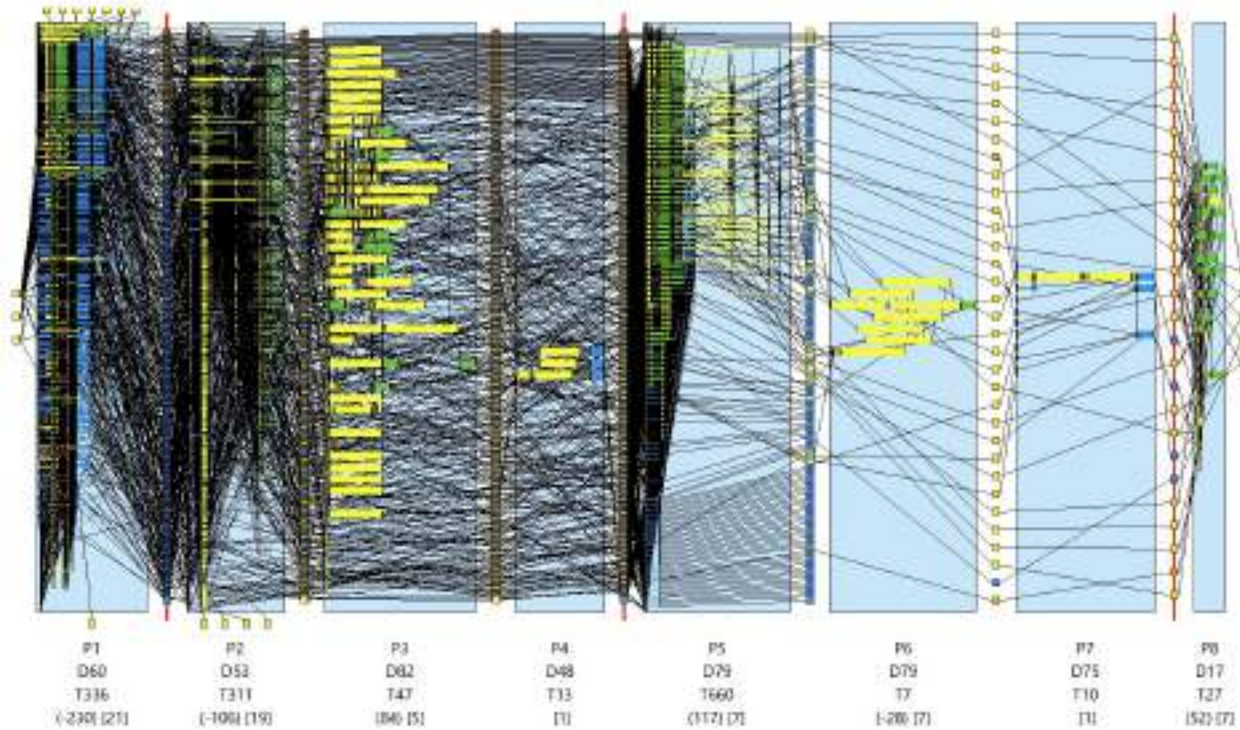
```
_C2R_MODULE_
void cnn_c2ri(C2RI::BUS<C2RI_T, RIF_COUNT> *c2ri_bus, CNN_Channels *src) {
/// LayerGroup(0) :
  L0.FW(&c2ri_bus->s_ch[0], &src->ch0, &src->ch1);
  L1.FW(&c2ri_bus->s_ch[1], &src->ch1, &src->ch2);
/// LayerGroup(1) :
  L2.FW_RES_OUT(&c2ri_bus->s_ch[2], &src->ch2, &src->ch3, &src->ch2_res);
  L3.FW_RES_IN (&c2ri_bus->s_ch[3], &src->ch3, &src->ch2_res, &src->ch4);
. . .
/// LayerGroup(16) :
  L35.FW_RES_OUT(&c2ri_bus->s_ch[35], &src->ch35, &src->ch36, &src->ch35_res);
  L36.FW_RES_IN (&c2ri_bus->s_ch[36], &src->ch36, &src->ch35_res, &src->ch37);
/// LayerGroup(17) :
  L37.FW(&c2ri_bus->s_ch[37], &src->ch37, &src->ch38);
  L38.FW(&c2ri_bus->s_ch[38], &src->ch38, &src->ch39);

  c2ri_ctrl.ConnectChannel(c2ri_bus);
}
```

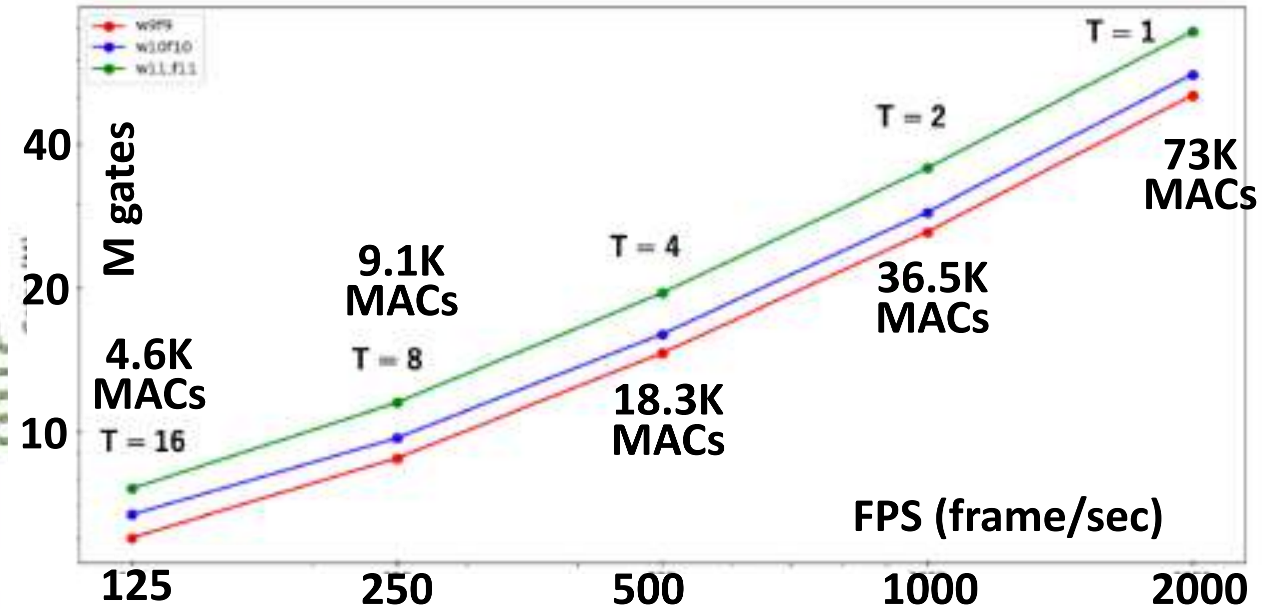
ネットワーク構成情報から
C2RTL用C++コードを自動生成
（C2RTLとは別のプログラムで）

・34層 + 補助層を個別にRTL変換
・層間の結合までをC++で表現 →
System level integration



Resnet-34 RTL Generation Results





Resnet CONV層(3x3 + 1x1)(Stride 2)
64 Input Channels/128 Output Channels

- 各層内で時分割処理を導入することで、gate/FPSトレードオフを実現
- 係数・画素のビット幅とゲート規模の比較
- クロック周波数:100MHzを想定
- 最大性能:50M gates, 73K MAC units, 2000 FPS

説明概要 : Outline

- ✧ **C2RTL高位システム設計環境のご紹介 :**
  C2RTL System-Level Design Framework
  - ➢ LLVM-C2RTLツール環境 : LLVM-C2RTL tool environment
  - ➢ データフロー記述方式 : C/C++ Data Flow Coding Style
  - ➢ 高位合成との相違点 : Comparison with HLS (High-Level Synthesis)
- ✧ **C2RTL設計開発事例 : C2RTL Design Examples**
  - ➢ Deep Neural Network : Super-resolution CNN, Resnet-34
  - ➢ RISC-V Processor : MMU/Cache, Linux verification, FPU, DNN extension

RISC-V Processor C++ Resource Description

GCC-attribute形式でHW属性記述

```
#define _BW(N)  __attribute__((C2R_bit_width(N)))
#define _T(N)   __attribute__((C2R_type(N)))

typedef uint8_t      BIT _BW(1), UINT2 _BW(2);
typedef uint8_t      UINT4 _BW(4), UINT8 _BW(8);
typedef BIT          ST_BIT _T(state);
typedef uint32_t     M_UINT32 _T(memory);
typedef uint32_t     ST_UINT32 _T(state);
```

typedef/変数のHW属性指定

- ビット幅指定 → _BW(N)
- _T(state) → レジスタ
- _T(memory) → メモリ
- その他 → ワイヤ

```
struct CPU {
  ST_UINT32      gpr[GPR_COUNT];
  M_UINT32       pmem[PM_SIZE];
  M_UINT32       dmem[DM_SIZE];
  ST_BIT         halted;
  ST_UINT32      cycle, ir_prev;
  UINT32    ir;
  Insn      insn;
  FESig     fe_sig;      FEreg   fe_reg _T(state);
  DCSig     dc_sig;      DCreg   dc_reg _T(state);
  EXSig     ex_sig;      EXreg   ex_reg _T(state);
  WBSig     wb_sig;      MDreg   md     _T(state);
  IO        io;
};
```

レジスタファイル

メモリ

パイプラインレジスタ

パイプライン
内部信号（ワイヤ）

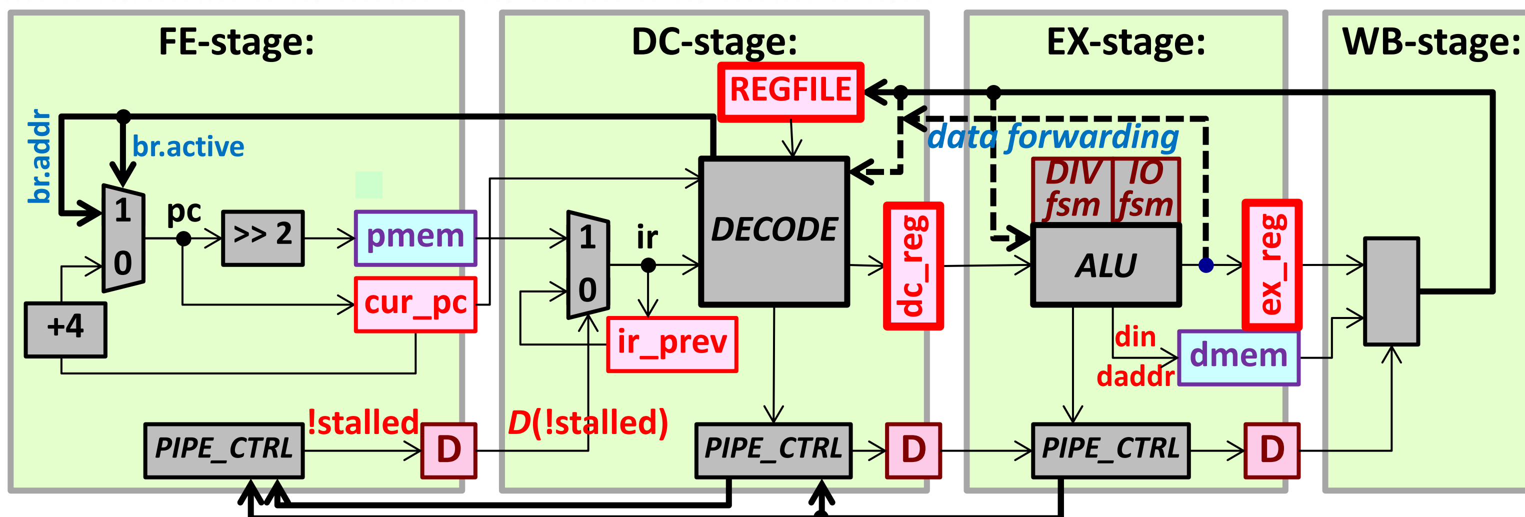
- ✧ 標準C++開発環境で実行可能
- ✧ 言語拡張（独自予約語など）なし
- ✧ ビルトインクラスなし

RISC-V Processor Pipeline Behavior Description

```
int CPU :: step (D_FIFO_PORT *in_fifo, D_FIFO_PORT *out_fifo)
{
  set_fifo_input_ports(&io.fifo, in_fifo, out_fifo);
  fetch();
  decode();
  execute();
  writeback();
  set_fifo_output_ports(&io.fifo, in_fifo, out_fifo);
  return (cpu.halted == 1);
}
```

RTL合成トップ関数：
全システムの
1サイクル動作を定義

C++記述制約：トップ関数内で各状態更新は「1回以下」
（状態更新 → レジスタ・メモリへの書込み）
→　任意のRTL構造をC++データフロー記述で表現可能

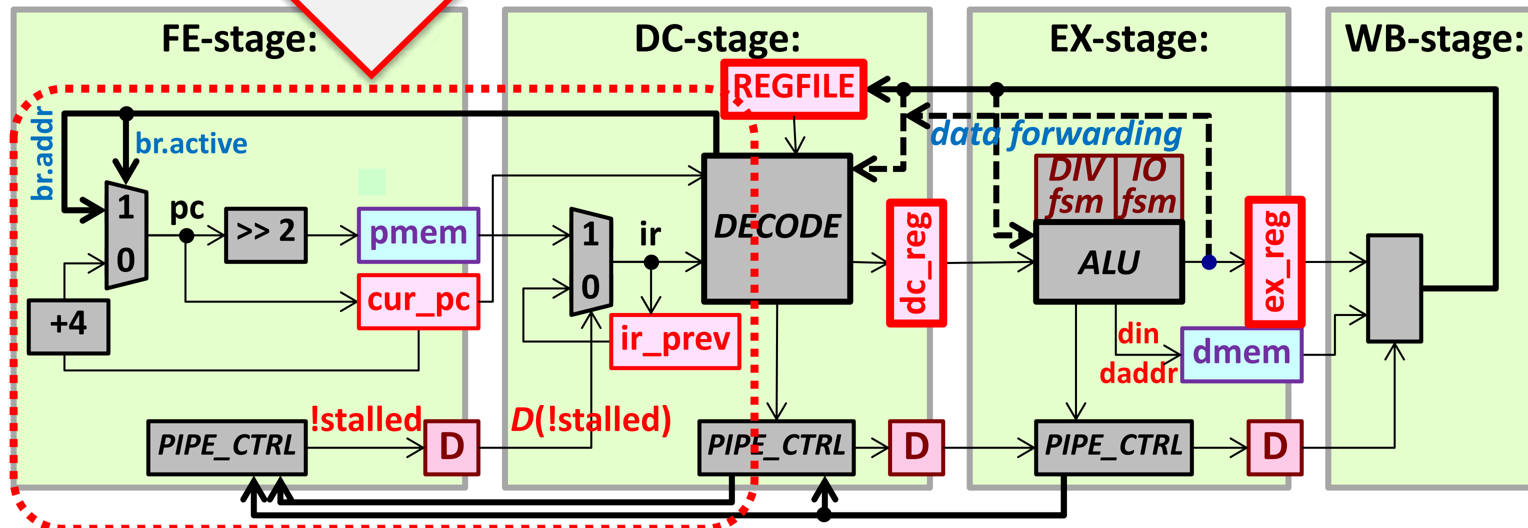


Instruction Fetch Stage C++ Description

```
void CPU :: fetch() {
  fe_sig.pctl.stalled  =  dc_sig.pctl.stall | ex_sig.pctl.stall;
  if ( ! fe_sig.pctl.stalled ) {
    UINT32 pc = (dc_sig.br.active) ? dc_sig.br.addr : fe_reg.cur_pc + 4;
    fe_reg.cur_pc = pc;
    ir = pmem[pc >> 2];
    ir_prev = ir;
  } else { ir = ir_prev;  }
}
```

メモリ・レジスタ出力参照 → 1クロック遅延

- ✧ メモリ　：pmem[ ]
- ✧ レジスタ：cur_pc, ir_prev
- ✧ ワイヤ　：pc, ir



RISC-V Instruction Decode Stage C++ Description

```
enum RVFields {
    RVF_funct7,
    RVF_funct3,
    RVF_funct2,
    RVF_rs3,
    RVF_rs2,
    RVF_rs1,
    RVF_rd,
    RVF_opc,
    RVF_imm12H,
    RVF_imm5L,
    RVF_imm20H,
    RVF_count,
};
```

```
struct Field { SINT32 msb, bits; };
Field  FLD[ RVF_count ]  =  {
    { 31, 7 },  /// RVF_funct7
    { 14, 3 },  /// RVF_funct3
    { 26, 2 },  /// RVF_funct2
    { 31, 5 },  /// RVF_rs3
    { 24, 5 },  /// RVF_rs2
    { 19, 5 },  /// RVF_rs1
    { 11, 5 },  /// RVF_rd
    {  6, 7 },  /// RVF_opc
    { 31, 12 }, /// RVF_imm12H
    { 11, 5 },  /// RVF_imm5L
    { 31, 20 }, /// RVF_imm20H
};
```

```
enum RVInsnOpCode {
    /// opc[6:0]
    RVI_lui     = 0x37,
    RVI_auipc   = 0x17,
    RVI_jal     = 0x6f,
    RVI_jalr    = 0x67,
    RVI_br      = 0x63,
    RVI_ld      = 0x03,
    RVI_st      = 0x23,
    RVI_compi   = 0x13,
    RVI_comp    = 0x33,
    RVI_fence   = 0x0f,
    RVI_sys     = 0x73,
};
```

RISC-V命令セット定義
- 命令フィールド
- オペコード

```
void CPU :: decode() { . . . .
  DEC_FLD(opc);
  switch (insn.opc) {
    case RVI_lui:    dec_U(); . . .
    case RVI_auipc: dec_U(); . . .
    case RVI_jal:    dec_UJ(); br_type = 1; . .
    case RVI_jalr:   dec_I();  br_type = 2; . .
    case RVI_br:     dec_SB(); br_type = 3; . .
    case RVI_ld:     dec_I(); . . .
    case RVI_st:     dec_S(); . . .
    case RVI_compi: dec_I(); . . .
    case RVI_comp:  dec_R(); . . .
    . . .
```

```
#define GET_BITS(d, m, b)   (((d) >> ((m) - (b) + 1)) & ((1 << (b)) - 1))
#define GET_FLD(fid)        (GET_BITS (ir, FLD[fid].msb, FLD[fid].bits))
#define DEC_FLD(fid)        insn.fid = GET_FLD (RVF_##fid)

/// R : funct7[31:25], rs2[24:20], rs1[19:15], funct3[14:12], rd[11:7], opc[6:0]
void CPU :: dec_R ()
{  DEC_FLD(funct7); DEC_FLD(rs2); DEC_FLD(rs1); DEC_FLD(funct3); DEC_FLD(rd); }

/// S : imm11_5[31:25], rs2[24:20], rs1[19:15], funct3[14:12], imm4_0[11:7], opc[6:0]
void CPU :: dec_S () {
  DEC_FLD(funct7); DEC_FLD(rs2); DEC_FLD(rs1); DEC_FLD(funct3); DEC_FLD(imm5L);
  SINT32 i11 = ir & 0x80000000;     /// sign bit
  UINT32 i10_5 = (insn.funct7) & 0x3f; /// 6 bits : funct7 --> same as imm11_5[31:25]
  insn.imm = (i11 >> 20) | (i10_5 << 5) | insn.imm5L;
}
```

RISC-V命令デコード記述
- オペコード別デコード関数

RISC-V EX-stage/WB stage C++ Description

EX-stage : 命令別動作記述

```
void CPU::execute() { . . . .
 switch (dc_reg.op) { . . . .
  case RVO_ld:  /// LOAD命令
    wb_sig.dout = format_rd (dmem[daddr], byte_pos); /// shift-right, sign/zero extend
    break;
  case RVO_st:  /// STORE命令
    st_data = format_wd (src2, byte_pos);                /// shift-left
    write_mem (&dmem[daddr], byte_en (byte_pos), st_data); /// write with byte-enable
    break;
  case RVO_comp:   /// 演算命令
    BIT sign0 = (src0 >> 31), sign1 = (src1 >> 31);
    BIT ultFlag = (src0 < src1);
    BIT sraFlag = (dc_reg.sextFlag && sign0);
    UINT32 shamt = (src1 & 0x1f);
    switch (dc_reg.funct3) {
      case RVF3_add:  ex_out = add_out;                  break;
      case RVF3_slt:   ex_out = sign0 ^ sign1 ^ ultFlag;  break;
      case RVF3_sltu:  ex_out = ultFlag;                 break;
      case RVF3_shl:   ex_out = src0 << shamt;           break;
      case RVF3_shr:  ex_out = (src0 >> shamt) | ((sraFlag) ? ~(0xffffffffu >> shamt) : 0); break;
      case RVF3_xor:  ex_out = src0 ^ src1;              break;
      case RVF3_or:    ex_out = src0 | src1;             break;
      case RVF3_and: ex_out = src0 & src1;               break;
    }
    break;
. . . .
```

```
void write_mem (UINT32 *mem, UINT4 be, UINT32 din) {
  UINT8 *cmem = (UINT8 *)mem;
  UINT8 *cdin = (UINT8 *)&din;
  if (be & 0x1) { cmem[0] = cdin[0]; }
  if (be & 0x2) { cmem[1] = cdin[1]; }
  if (be & 0x4) { cmem[2] = cdin[2]; }
  if (be & 0x8) { cmem[3] = cdin[3]; }
}
```

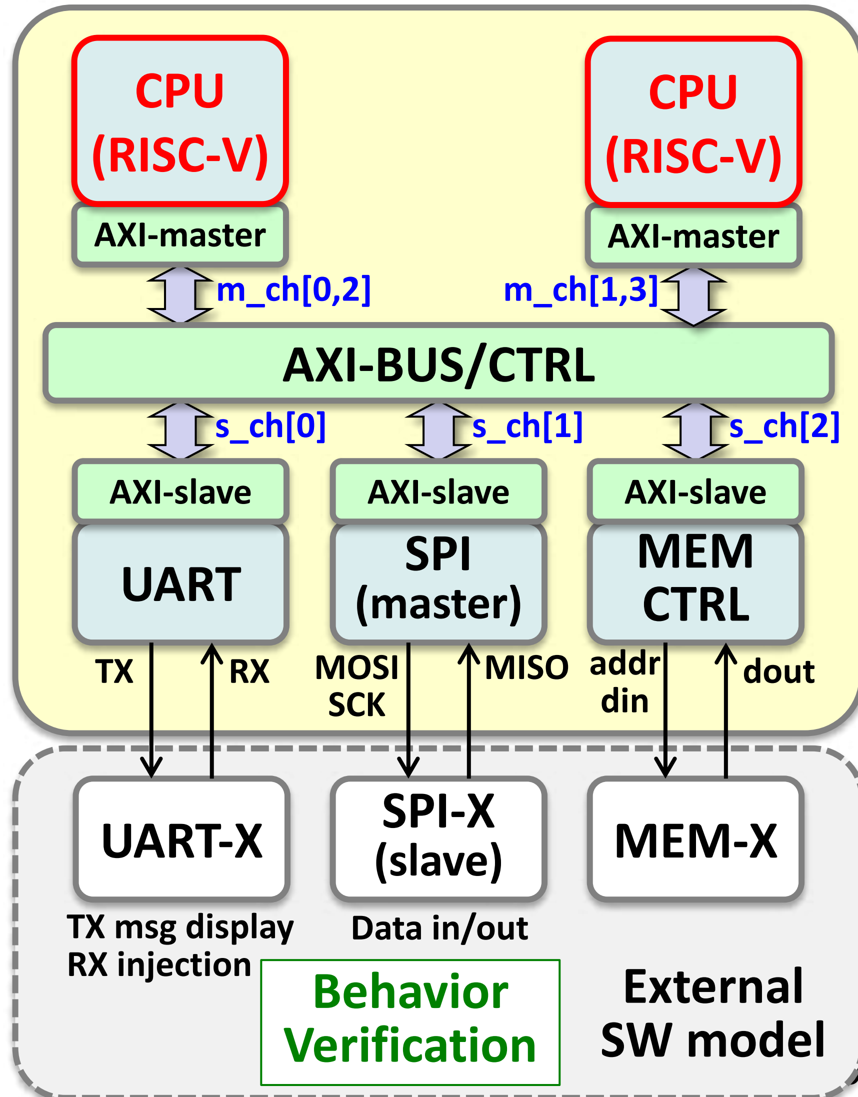
32-bitワードを8-bitに分割してアクセス

Memory-Write with byte-enable

WB-stage : REG-FILE書込み

```
void CPU::writeback() { . . . .
  UINT32 wb_data = (ex_reg.wbflag == W_dout) ? wb_stt.dout : ex_reg.out;
  wb_sig.pctl.stalled  = ex_sig.pctl.stall_fw;
  if (!wb_stt.pctl.stalled_flag && ex_reg.dst_f) {
    gpr [ ex_reg.dst_id ] = wb_data;
  }
  . . . .
```

RISC-V SoC Modeling



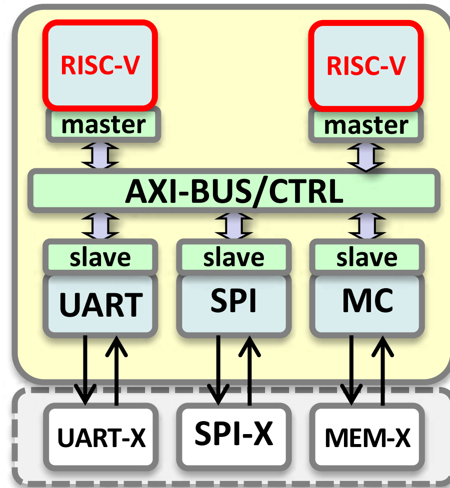
SoCのモデリング階層

- ✧ **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 x2 : キャッシュ付き、AXI-Master x2/コア**
- ✧ **UART : AXI-Slave**
- ✧ **SPI : AXI-Slave**
- ✧ **MEMCTRL : AXI-Slave**
- ✧ **AXI-BUS : 4-Master, 3-Slave**

RV-SoC Top Level C++ Description




```
#define _C2R_MODULE __attribute__((C2R_module))
```

SoC階層属性

```
_C2R_MODULE_
int RVProcAXI (D_FIFO_PORT *in_fifo, D_FIFO_PORT *out_fifo,
  D_FIFO_PORT *in_fifo2, D_FIFO_PORT *out_fifo2,
  MEMCTLPin *mpin, UARTPin *uart, SPIPin *spi, AXI4L::BUS<4,3> *axib) {
  axi_uart.step (&axib->s_ch[0], uart);
  axi_spim.step (&axib->s_ch[1], spi);
  axi_memctl.step (&axib->s_ch[2], mpin);
  int val = cpu1.step (in_fifo, out_fifo, &axib->m_ch[0], &axib->m_ch[2]);
  val &= cpu2.step (in_fifo2, out_fifo2, &axib->m_ch[1], &axib->m_ch[3]);
  axi_bus_ctrl.connectChannel (axib);
  return val;
}
```

SoCレベル記述：IPトップ関数の呼出し → IP接続記述

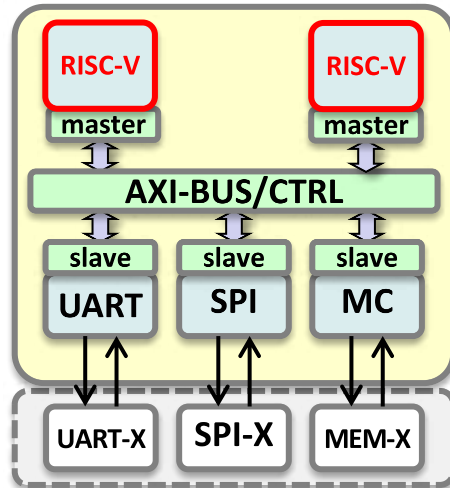
```
FIFO_PORT in_fifo[2], out_fifo[2];
MEMCTRLPin mpin;
UARTPin uart ;
SPIPin spi ;
AXI4L::BUS<4, 3> axi_bus = { 0 } ;
int main (int argc, char * argv[]) {
  /// initialization codes here...
  while ( !RVProcAXI (&in_fifo[0], &out_fifo[0], &in_fifo[1], &out_fifo[1], &mpin,&uart, &spi, &axi_bus)) {
    spi_ext_slave (&spi) ;
    uart.rx = uart_ext(uart.tx) ;
    mem_ext.update (&mpin) ;
  }
}
```

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

外部
SW モデル

RV-SoC IP-Level C++ Description




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

IP階層属性（個別RTL合成関数）

CPU top : 5段パイプライン

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

D-cache実装のため、1段追加

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-UART top

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

AXI-SPI top

```
_C2R_FUNC(1)
void SPIM_AXI4L::step(AXI4L::CH *axi,
   SPIPin *spi_pin) {
   spim.set_pin(spi_pin);
   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 BUS Resource Description

AXI Channels

```
struct AXI4L {
  //////////// AXI-channels ////////////
  struct CH {
    struct ADDR {  /// read-addr, write-addr
      struct MA { UINT32 addr ; BIT valid ; UINT4 len; } m ;
      struct SL  { BIT ready ; } s ;
    }  raddr , waddr ;
    struct RDAT {   /// read-data
      struct MA { BIT ready ; } m ;
      struct SL  { UINT32 data ; UINT2 resp ; BIT valid, last ; } s ;
    }  rdat ;
    struct WDAT {  /// write-data
      struct MA { UINT32 data ; UINT4 strobe ; BIT valid, last ; } m ;
      struct SL  { BIT ready ; } s ;
    }  wdat ;
    struct WRES {  /// write-resp
      struct MA { BIT ready ; } m ;
      struct SL  { UINT2 resp ; BIT valid ; } s ;
    }  wres ;
  } _T(direct_signal) ; /// all signals are unlatched wires
  /// continued…
```

```
///struct AXI4L {
 template <int MC, int SC> struct BUS
 {   CH  m_ch[MC], s_ch[SC];   };
```

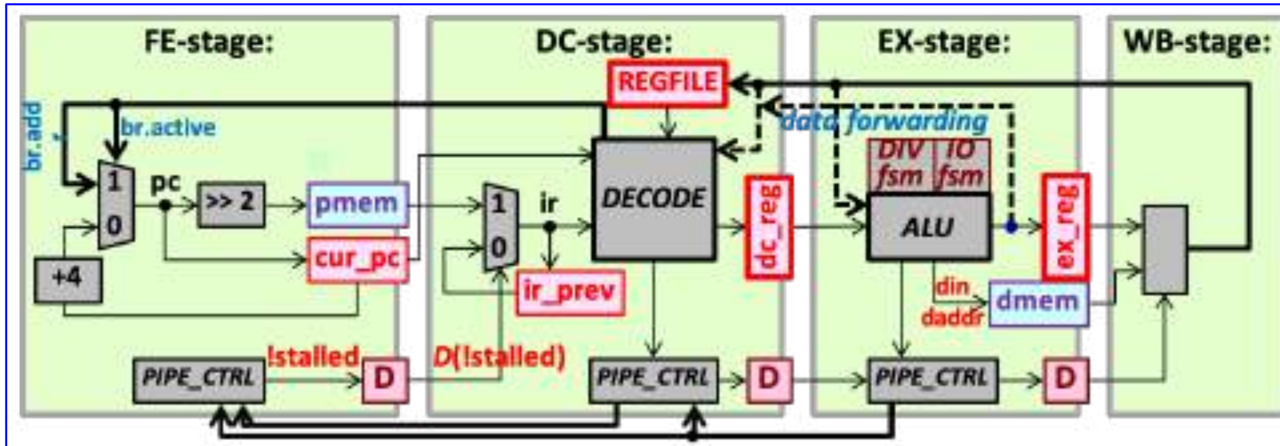
AXI BUS (MC masters, SC slaves)

チャネル信号設定用メソッド関数が用意されている

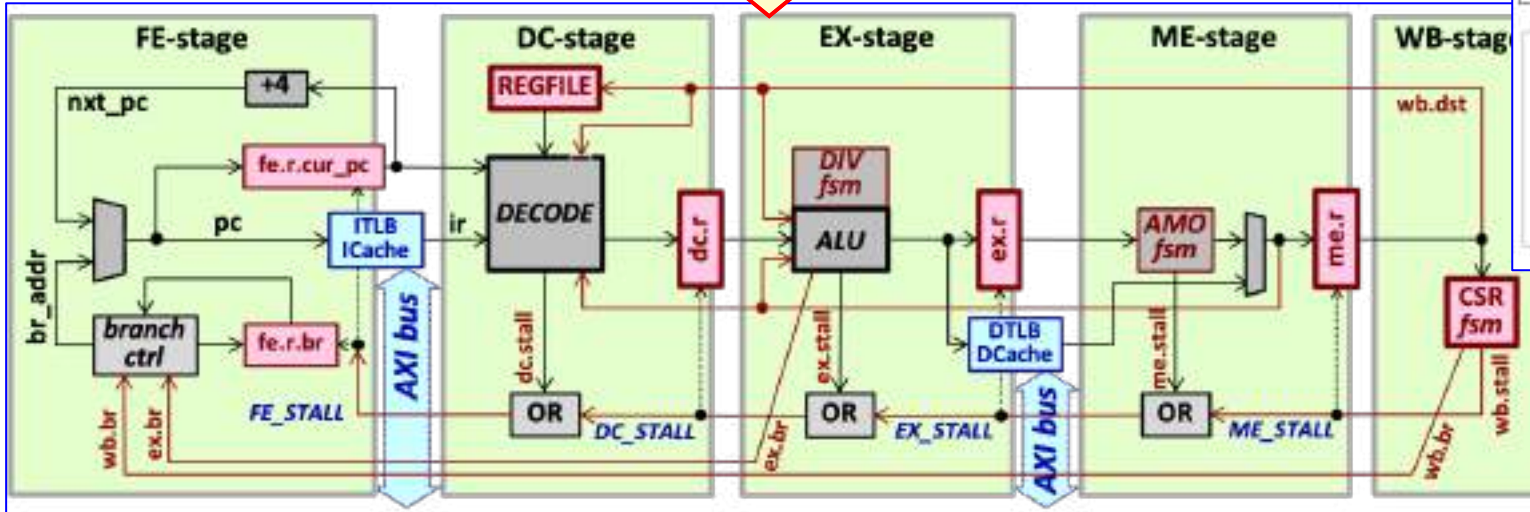
AXI-BUS_CTRL top

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

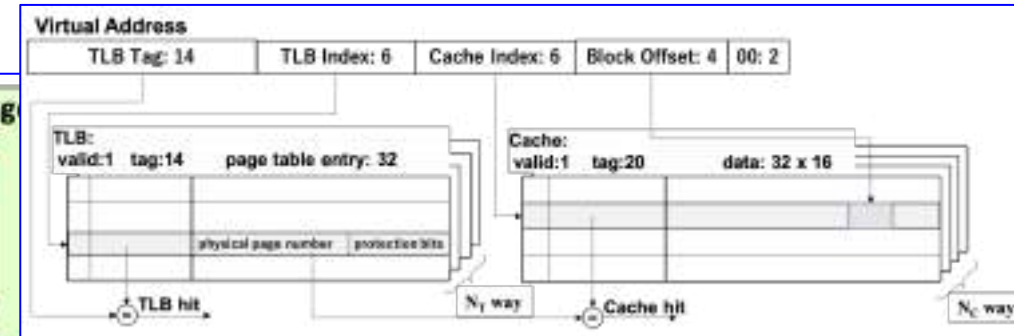
RISC-V Processor Cache/MMU Design



4-stage → 5-stage

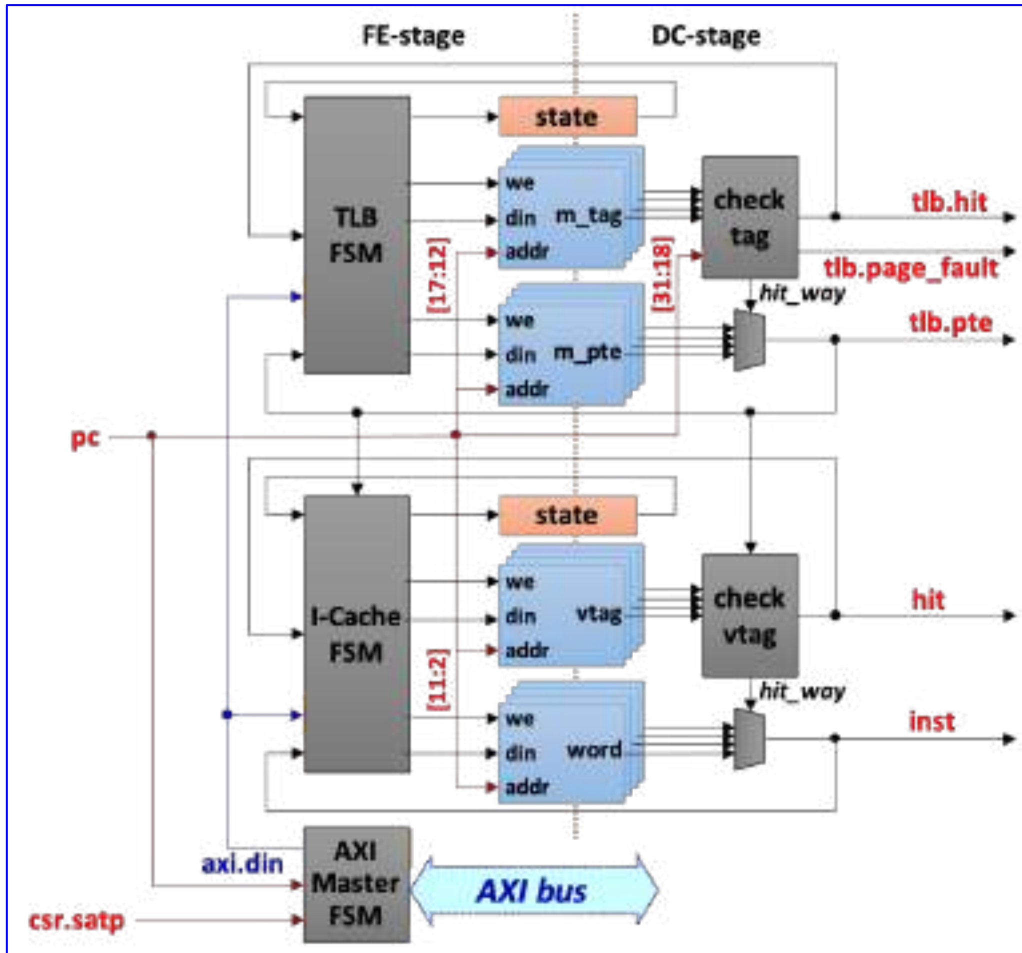


- ・MMU : Linux-OS等で必須
- ・MMU = TLB + page-walk logic
- ・Virtually-indexed physically tagged (VIPT)
- ・D-Cache/D-TLBのために 1 stage追加
- ・I-Cache/I-TLBとD-Cache/D-TLBそれぞれにAXI-Masterポートを接続



Virtually-indexed physically tagged (VIPT)

RISC-V Processor Cache/MMU Design



I-Cache/I-TLB/AXI FSM構成

```
template <..> void Cache::check_vtag() {
  tstt.reset();
  for (int w = 0; w < N_WAY; w++) {
    if (adf.tag == (out_vtag[w] & ~DIRTY_BIT)) {
        tstt.dirty = (out_vtag[w] & DIRTY_BIT) != 0;
        tstt.hit |= 1;
        tstt.hit_way = w;
    }
  }
}
```

```
#define DIRTY_BIT  (1 << (BT + 1))
#define VALID_BIT  (1 << (BT))
```

```
template<..> UINT32 ICache::fsm(BIT mmu_en, UINT32 pc,
  AXI4L::CH *axi, unsigned ptbr, DCState &dc, FEState &fe) {
  this->extract_addr_fields(pc); /// pc => adf
  unsigned vp = this->adf.tag;
  access_axi(mmu_en, axi, pc, ptbr);

  update_state(mmu_en, dc.iflush);

  tlb.i_fsm(mmu_en, dc.itflush, vp, pc, &this->axim);

  if (tlb.state != TLB_ST_INIT && mmu_en) {
    this->adf.tag = (tlb.pte >> 10) | VALID_BIT;
  }
  this->check_vtag();

  if (tlb.state == TLB_ST_INIT || !tlb.hit) {
    this->busy = 1;                    /// TLB miss
  } else if (tlb.page_fault) {
    this->busy = 0;                    /// page fault : raise exception
  } else {
    this->busy = !this->tstt.hit;      /// cache miss
  }
  UINT32 inst = RV_NOP_INST;
  if (!this->busy && !tlb.page_fault) { /// read cache word
    inst = this->out_word[this->tstt.hit_way];
    this->lru.update(this->adf.idx, this->tstt.hit_way); /// LRU-update
  }
  if (!fe.r.ready || fe.r.sw_mode) { tlb.page_fault = 0; }
  if (!fe.r.ready) { inst = prev_inst;}
  prev_inst = inst;
  return (fe.r.sw_mode) ? RV_NOP_INST : inst;
}
```

```
#define RV_NOP_INST  0x00000013
```

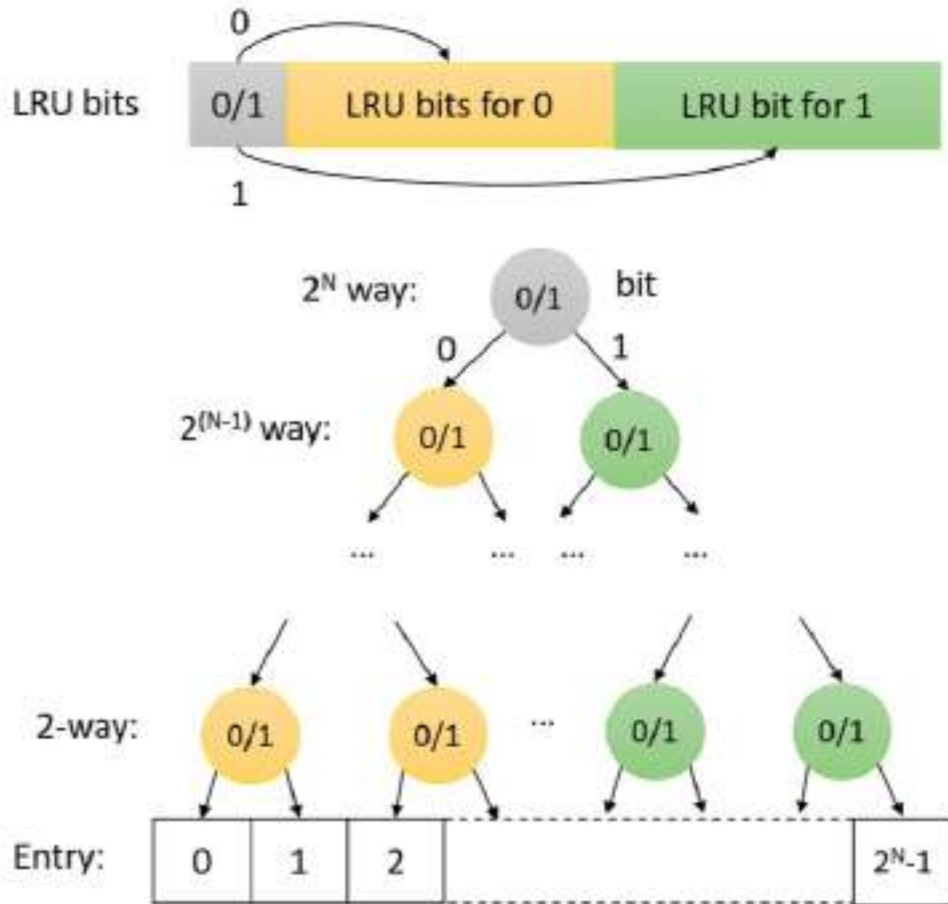
AXI-master FSM

I-Cache core FSM

I-TLB FSM

I-Cache tag check on physical tag

RISC-V Processor Cache/MMU Design




```
template<int BW> unsigned select_lru(unsigned bits) {
  /// parameters for sub-tree
  const unsigned BW2 = BW - 1;
  const unsigned BL2 = (1 << BW2) - 1;
  const unsigned MASK2 = (1 << BL2) - 1;
  unsigned victim;
  /// check MSB
  if (bits & (1 << (BL2 << 1))) {
    unsigned bits2 = bits & MASK2;
    victim = select_lru<BW2>(bits2) + (1<<BW2);
  } else {
    unsigned bits = (bits >> BL2) & MASK2;
    victim = select_lru<BW2>(bits2);
  }
  return victim;
}
template<> unsigned select_lru<1>(unsigned bits) {
  return bits;
}
```

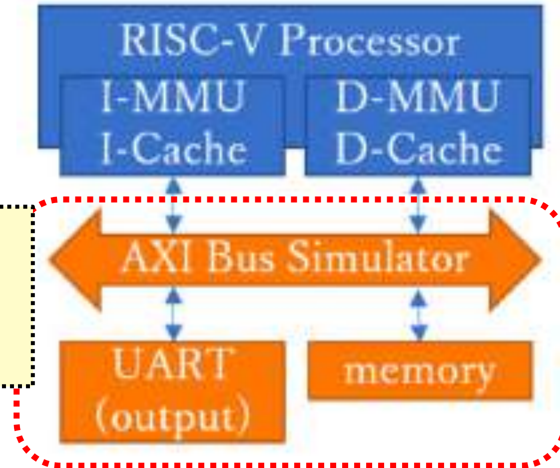
再帰関数
呼出による
2分岐探索
RTL表現

再帰型pseudo-LRUキャッシュ置換

RISC-V Processor Cache/MMU Design : Simulation

RISC-Vテストベンチ

AXIバス、UART、メモリ
→SWモデルによる
高速シミュレーション




```
struct {AXI::CH m_ch[2];} bus; /// bus instance
CPU cpu1;                      /// CPU instance
struct AXI_SIM {               /// AXI bus simulator
  AXI::CH::ADDR:MA ra; /// Latched master read ch
  ....
  Read(AXI::CH *io)  {     /// Read request
    if (io.raddr.m.valid) {ra = io.raddr.m; ...}
    if (ra.valid && io.rdat.m.ready) {
      io.rdat.s.data = mem_read(ra.addr);
      ....
      io.rdat.s.valid = 1;    /// Transfer data
    } else { io.rdat.s.valid = 0; }
  }
  Write(AXI::CH *io) { ... } /// Write request
} ch_sim[2];

int main(...) { ..... /// main simulation loop
  while(!cpu1.step(&bus.m_ch[0],&bus.m_ch[1],0)){
    ch_sim[0].Read(&bus.m_ch[0]);  /// Data Read
    ch_sim[0].Write(&bus.m_ch[0]); /// Data Write
    ch_sim[1].Read(&bus.m_ch[1]);  /// Inst Read
  } ....
}
```

model	cycles	time (sec)	cycles/sec	speedup
C++ data flow	108,142,423	17.343	6,235,508	234.894
RTL-C [1]	110,580,327	75.520	1,464,252	55.157
RTL-C [2]	110,580,327	103.056	1,073,012	40.420
Verilog (VCS)	110,580,327	4165.520	26,546	1.000

[1] without test-vector generation. [2] with test vector generation.

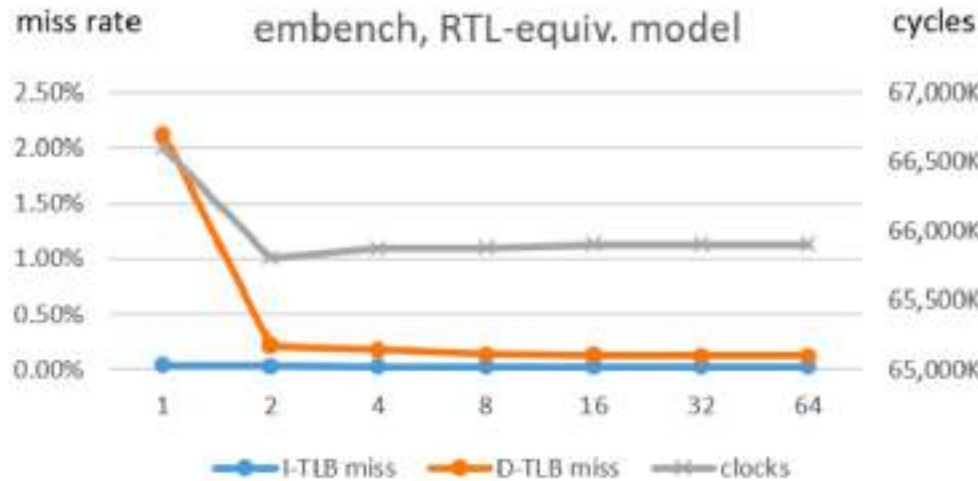
シミュレーション時間 (Linux boot)

シミュレーション速度（対 VCS）:
・C++ データフロー記述 : 234倍
・RTL等価Cモデル（自動生成）: 55倍

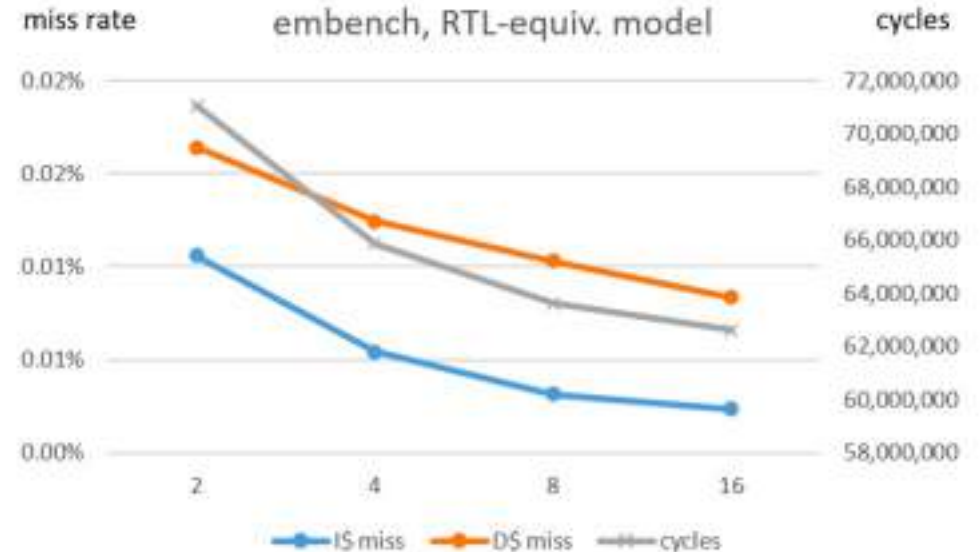
RISC-V Processor Cache/MMU Design : RTL Synthesis

# TLB ways	1	2	4	8	16	32	64
# CLBs	7,694	8,096	8,412	9,380	11,577	16,124	14,537
# FFs	4,129	4,292	4,496	4,888	5,598	7,135	10,144
freq.(MHz)	142.85	133.33	127.22	114.91	104.71	100.00	99.00
comb. gates (C2RTL)	35,317	35,988	37,203	39,633	44,678	54,957	64,605

TLB way数と回路規模 (KU-3P-I1 FPGA)

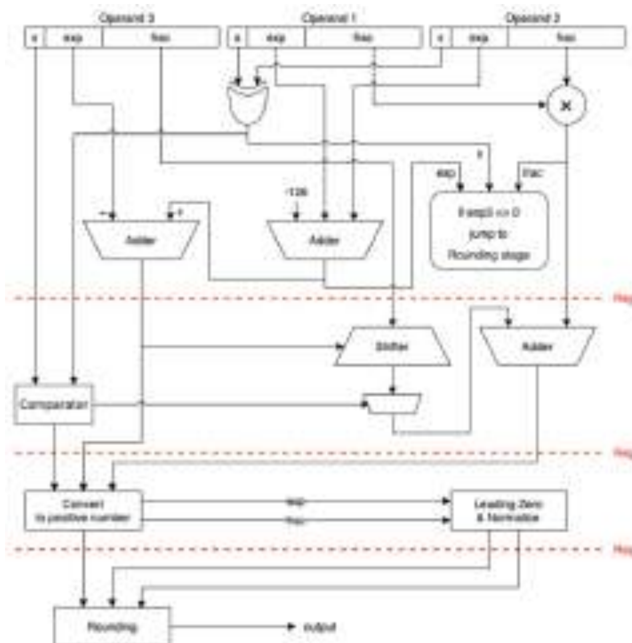


TLB way数とTLBミス率・実行サイクル数



cache way数とTLBミス率・実行サイクル数

RISC-V Processor FPU Implementation



FPU積和演算器

命令セット	# LUTs	#FF	#DSP
RV32-IMA	11,139	6,593	4
RV32-IMAF	15,546	8,078	14
RV32-IMAF +CORDICx	16,683	8,037	6

XC7Z020 FPGA合成結果

標準命令
- 四則演算
- 積和演算
- 平方根
- 比較演算
- 整数変換

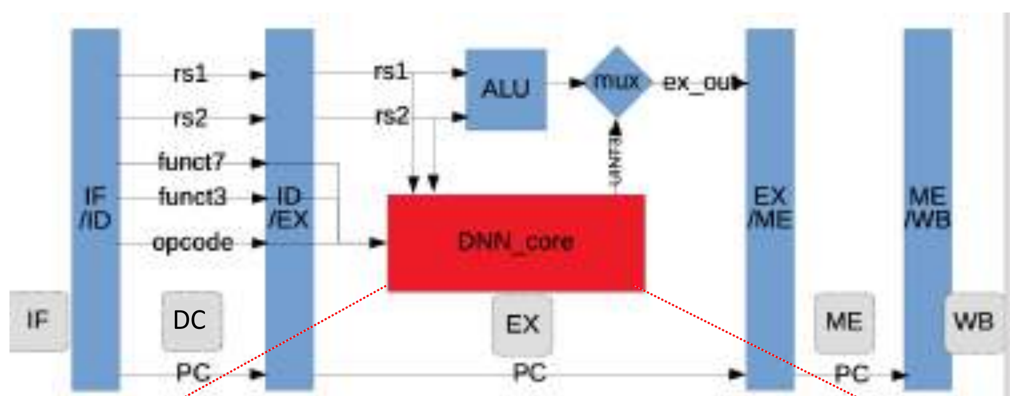
CORDIC 拡張命令
- log
- cos/sin

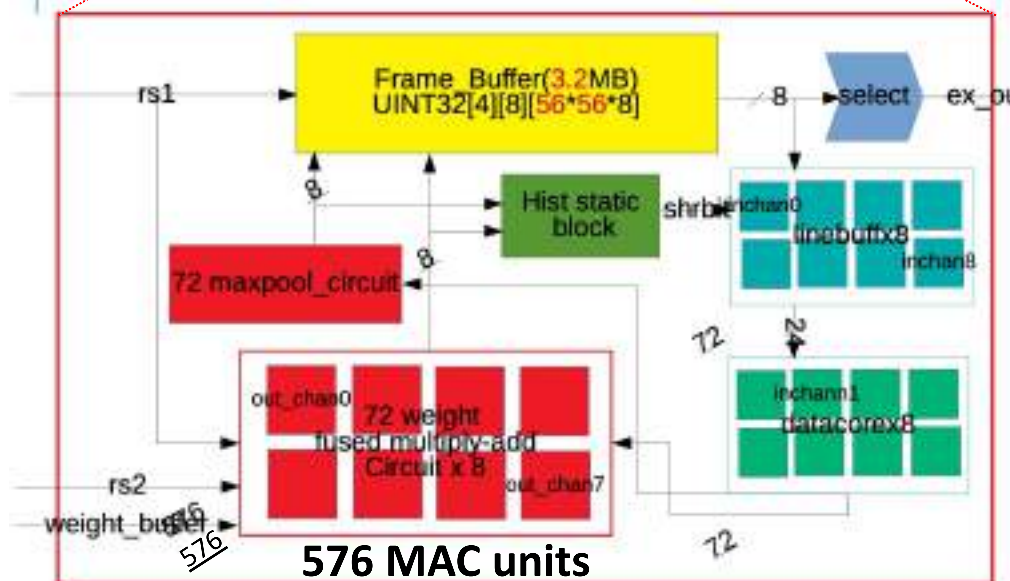
(平方根、除算も実装)

minver : 3x3逆行列計算ベンチマーク		
命令セット	# Cycles	Speed-up
RV32-IMA	6,225,860	1.00
RV32-IMAF	906,200	6.87

math.h : logf(), sinf()		
命令セット	logf()	sinf()
RV32-IMAF	~180 cycles	100~2,400 cycles
RV32-IMAF +CORDICx	18 cycles	34 cycles

RISC-V Processor Deep Neural Network Extension





メモリ階層

- 外部DDR
- Cache/TLB
- Frame Buffer (8-ch)
- Line Buffer (8-ch x 3)
- Shift Register (8-ch x 3 x 3)

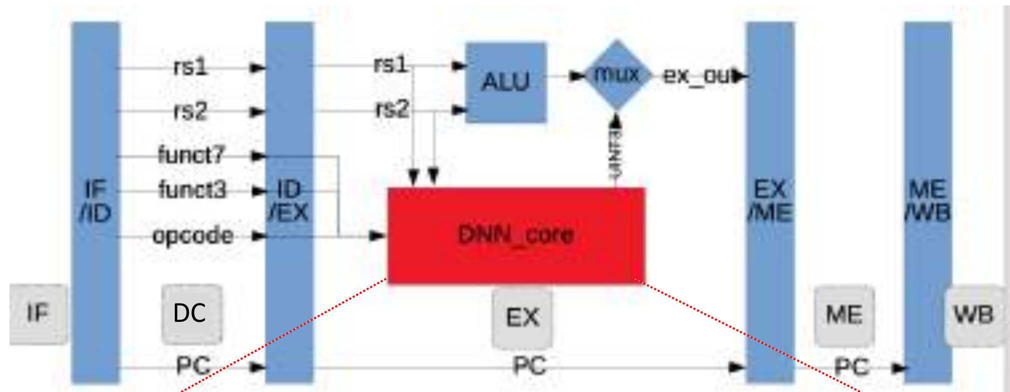
拡張命令

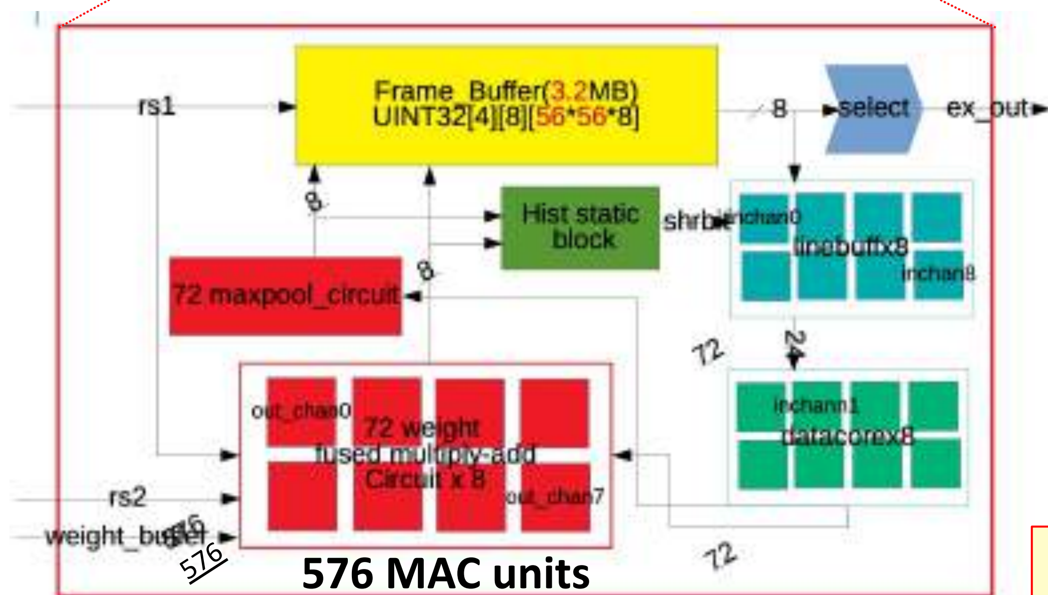
- Frame Buffer転送
- Weight係数転送
- CONV, MAXPOOL, LINEAR
- Zero-overhead loop

Datapath : INT8 + global-exp

- 8-ch x 3 x 3 x 8 CONV (576 MAC units)
- 8-ch x 3 x 3 MAX-POOL
- global-exp 正規化 (feature-map毎に1つの指数部)

RISC-V Processor Deep Neural Network Extension





命令セット	# gates	# LUTs	#FF	#DSP	#BRAM
RV32-IMA	69,556	8,895	4,691	5	8
RV32-IMA +DNNx	620,431 (8.91x)	104,704 (11.77x)	15,581 (3.32x)	10 (2.0x)	808 (101.0x)

A-U200-P64G FPGA合成結果

DNN処理	RV32-IMA (cycles)	RV32-IMA +DNNx (cycles)	Speedup
CONV	71,840,771,062	28,493,941	2521.26
MAXPOOL	46,511,122	101,006	460.47
AVEPOOL	289,443	3,171	91.27
LINEAR	5,052,818	519,342	9.72
Data Transfer	-----	5,698,432	-----
TOTAL	71,892,624,445	34,815,892	2064.93
FPS @ 100MHz	0.00139 FPS	2.872 FPS	

ResNet-34実行サイクル数

(P14) DNN-HW：50M gates, 73K MAC units, 2000 FPS @ 100MHz

(695.37x FPS, 80.58x gates, 126.73x MAC units)

まとめ : Conclusion

- ✧ **C2RTL : System-Level Design Verification Framework using C/C++**
  - ➢ C/C++データフロー記述方式 : 1サイクル動作記述によるRTL構造表現
  - ➢ RTL合成とRTL等価Cモデル自動生成 → C/C++開発環境でRTL検証可能
  - ➢ System integration : C++記述による一気通貫SoC合成フロー
    - ◆ IPレベル : RTL合成単位、ユニット検証
    - ◆ SoCレベル : IP統合、システム検証
- ✧ **C2RTL Design Examples**
  - ➢ Deep Neural Network (Resnet-34): 50M gates, 73K MAC units, 2000FPS
  - ➢ RISC-V Processor + MMU/Cache : Linux + EmBench simulation, TLB/CacheのN-way RTL Synthesis
  - ➢ RISC-V Processor Extension : FPU(32-bit float), DNN Acceleration