(産業の)米づくり環境ご紹介

第3回自作CPUを語る会
R06.06.02 石谷太一

自己紹介

- 石谷太一(イシタニタイチ)
  - GitHub taichi-ishitani
  - Twitter @taichi600730
  - RTLやさん
- PEZY Computing K.K.
  - HP https://pezy.co.jp
  - GitHub https://github.com/pezy-computing
  - 辛うじて生きてきます
  - 担当業務
    - IP 組み込み/周辺回路設計検証
    - 社内バスプロトコル整備
    - 共通モジュール整備
    - 設計環境整備

- 自作 open source hardware/software
  - RgGen
    - 制御レジスタ生成ツール
    - RICE 開発にも導入
  - tvip-axi
    - AMBA AXI VIP
  - tnoc
    - Network on Chip router/fabric
  - RICE
    - RISC-V コア
    - 今日の題目
  - RbJSON5
    - Ruby用JSON5パーサー
  - RuPkl
    - Ruby用Pklパーサー

今日のお題目

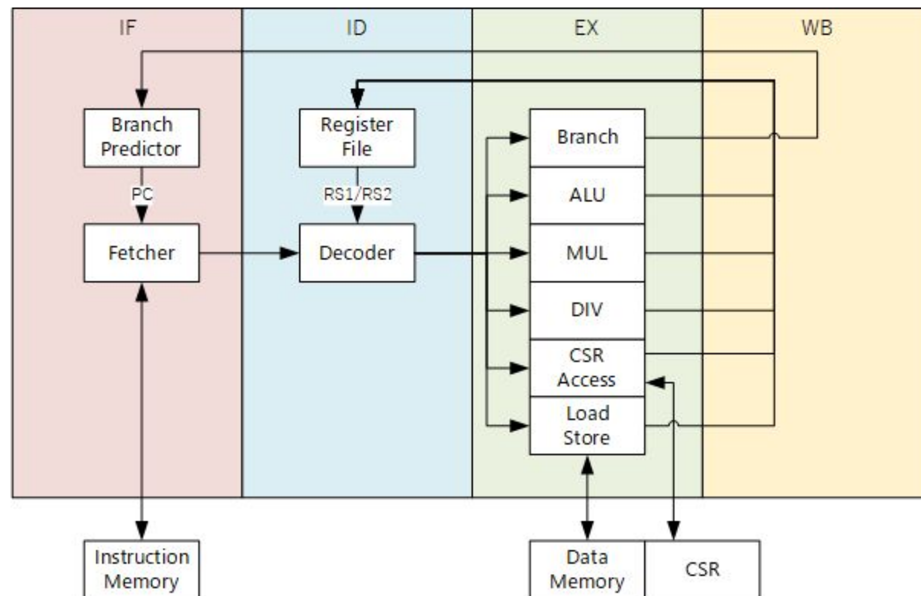
- 自作 RISC-V コア RICE のご紹介ではありません
- RICE **設計環境**のご紹介です
  - コードの自動生成
  - 検証環境

目次

- RICE 概要
- RTL 実装効率化
  - デコーダを（ちょっと）自動生成
  - CSR 自動生成
- 検証環境
  - Konata でパイプラインを良い感じに見る
  - ISS との一致比較
  - riscv-tests の導入

RICE 概要

- 名前の由来
  - マイコン→半導体→産業の米→RICE
- RV32IM
  - 4段パイプライン
    - IF/ID/EX/WB
  - 分岐予測も実装
- SystemVerilog で記述
- リポジトリ
  - https://github.com/taichi-ishitani/rice

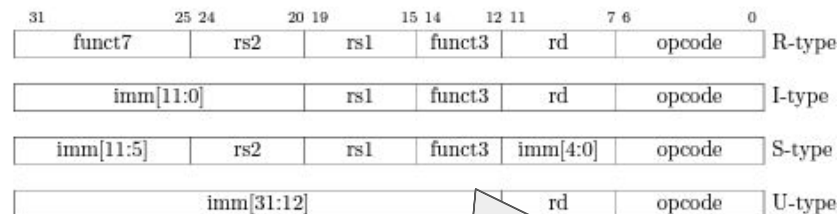


RTL 実装効率化 - デコーダを（ちょっと）自動生成

- デコーダは何してますの？
  - 入力された命令ビット列を解析して、後段に必要な情報を取り出す
    - オペランドの個数と種類（即値、レジスタ）は？
    - 使う演算器は？
    - 結果の格納先は？
- 例：入力命令ビット列が inst = 0x0010_0093 の場合
  - inst[6:0] = 0b001_0011 && inst[14:12] = 0b000
    - 命令は addi 命令
    - 加算器を使用
  - inst[11:7] = 0b000_1
    - 演算結果はレジスタ x1 に書き込み
  - inst[19:15] = 0x0000_0 / inst[31:20] = 0x001
    - オペランドはレジスタ x0 と即値 0x001

RTL 実装効率化 - デコーダを（ちょっと）自動生成

- 愚直な実装
  - 命令ビット列の必要箇所を切り出して、パターンに一致するか調べる
    - inst[6:0] = 0b001_0011 && inst[14:12] = 0b000 ならば addi 命令
  - 命令によって切り出す箇所が異なる
    - 命令のフォーマットによって、切り出す箇所は大体同じ
    - けど、たまに違う命令がいる
      - 論理右シフト (SRLI) と算術右シフト (SRAI)
      - inst[6:0] と inst[14:12] は同じ
      - 即値フィールドの未使用部分の値で判別
- 手書きは面倒なうえに、コードの見通しがよろしくない



基本的にはopcodeとfunct3/7でどの命令か判別できる

0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI

opcode/funct3 は同じ値になっていて、即値の未使用部分の値が異なる

RTL 実装効率化 - デコーダを(ちょっと)自動生成

```
case ({inst.opcode, inst.funct3, inst.funct7}) inside
  {RICE_CORE_OPCODE_LUI, 3'b???, 7'b???_????}:    //  lui
    return get_alu_operation(RICE_CORE_ALU_ADD, RICE_CORE_ALU_SOURCE_IMM_0, RICE_CORE_ALU_SOURCE_IMM);
  {RICE_CORE_OPCODE_AUIPC, 3'b???, 7'b???_????}:  //  auipc
    return get_alu_operation(RICE_CORE_ALU_ADD, RICE_CORE_ALU_SOURCE_PC, RICE_CORE_ALU_SOURCE_IMM);
  {RICE_CORE_OPCODE_OP_IMM, 3'b000, 7'b???_????}: //  addi
    return get_alu_operation(RICE_CORE_ALU_ADD, RICE_CORE_ALU_SOURCE_RS, RICE_CORE_ALU_SOURCE_IMM);
  {RICE_CORE_OPCODE_OP_IMM, 3'b010, 7'b???_????}: //  slti
    return get_alu_operation(RICE_CORE_ALU_LT, RICE_CORE_ALU_SOURCE_RS, RICE_CORE_ALU_SOURCE_IMM);
  {RICE_CORE_OPCODE_OP_IMM, 3'b011, 7'b???_????}: //  sltiu
    return get_alu_operation(RICE_CORE_ALU_LTU, RICE_CORE_ALU_SOURCE_RS, RICE_CORE_ALU_SOURCE_IMM);
  {RICE_CORE_OPCODE_OP_IMM, 3'b100, 7'b???_????}: //  xori
    return get_alu_operation(RICE_CORE_ALU_XOR, RICE_CORE_ALU_SOURCE_RS, RICE_CORE_ALU_SOURCE_IMM);
  {RICE_CORE_OPCODE_OP_IMM, 3'b110, 7'b???_????}: //  ori
    return get_alu_operation(RICE_CORE_ALU_OR, RICE_CORE_ALU_SOURCE_RS, RICE_CORE_ALU_SOURCE_IMM);
  {RICE_CORE_OPCODE_OP_IMM, 3'b111, 7'b???_????}: //  andi
    return get_alu_operation(RICE_CORE_ALU_AND, RICE_CORE_ALU_SOURCE_RS, RICE_CORE_ALU_SOURCE_IMM);
  {RICE_CORE_OPCODE_OP_IMM, 3'b001, 7'b000_0000}: //  slli
    return get_alu_operation(RICE_CORE_ALU_SLL, RICE_CORE_ALU_SOURCE_RS, RICE_CORE_ALU_SOURCE_IMM);
  {RICE_CORE_OPCODE_OP_IMM, 3'b101, 7'b000_0000}: //  srli
    return get_alu_operation(RICE_CORE_ALU_SRL, RICE_CORE_ALU_SOURCE_RS, RICE_CORE_ALU_SOURCE_IMM);
  {RICE_CORE_OPCODE_OP_IMM, 3'b101, 7'b010_0000}: //  srai
    return get_alu_operation(RICE_CORE_ALU_SRA, RICE_CORE_ALU_SOURCE_RS, RICE_CORE_ALU_SOURCE_IMM);
  {RICE_CORE_OPCODE_OP, 3'b000, 7'b000_0000}:     //  add
    return get_alu_operation(RICE_CORE_ALU_ADD, RICE_CORE_ALU_SOURCE_RS, RICE_CORE_ALU_SOURCE_RS);
  {RICE_CORE_OPCODE_OP, 3'b000, 7'b010_0000}:     //  sub
    return get_alu_operation(RICE_CORE_ALU_SUB, RICE_CORE_ALU_SOURCE_RS, RICE_CORE_ALU_SOURCE_RS);
```

特殊対応が不要な命令も、一致比較が不要なことを書く必要がある

シフト命令の特殊対応

RTL 実装効率化 - デコーダを(ちょっと)自動生成

- ちょっと自動化
  - 命令ごとの各フィールドの値を YAML で記述
    - https://github.com/taichi-ishitani/rice/blob/master/inst/riscv_inst.yaml
    - フィールド名での指定なので、ビット位置でより分かりやすい(と思う)
    - 特殊対応が必要な命令のみ、当該フィールドの指定を行う
  - YAML から命令の判別を行う関数定義を生成する
  - 判別を行う箇所では、生成した関数を呼び出す

```yaml
addi:
  { type: i, opcode: OP_IMM, funct3: 0b000 }
slti:
  { type: i, opcode: OP_IMM, funct3: 0b010 }
sltiu:
  { type: i, opcode: OP_IMM, funct3: 0b011 }
xori:
  { type: i, opcode: OP_IMM, funct3: 0b100 }
ori:
  { type: i, opcode: OP_IMM, funct3: 0b110 }
andi:
  { type: i, opcode: OP_IMM, funct3: 0b111 }
slli:
  { type: i, opcode: OP_IMM, funct3: 0b001, imm: '000000xxxxxx' }
srli:
  { type: i, opcode: OP_IMM, funct3: 0b101, imm: '000000xxxxxx' }
srai:
  { type: i, opcode: OP_IMM, funct3: 0b101, imm: '010000xxxxxx' }
```

特殊対応が必要な命令だけ個別対応する

RTL 実装効率化 - デコーダを(ちょっと)自動生成

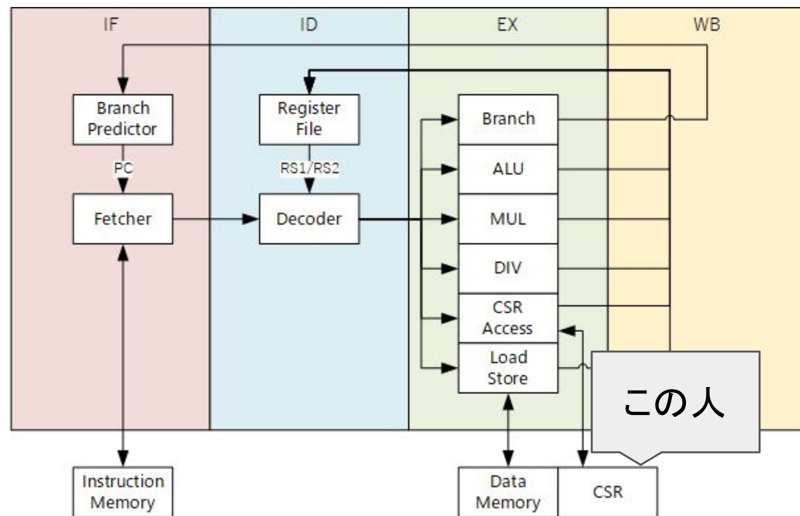
```
function automatic logic match_xori(rice_riscv_inst inst_bits);
  return inst_bits ==? {25'bxxxxxxxxxxxxxxxxx100xxxxx, RICE_RISCV_OPCODE_OP_IMM};
endfunction

function automatic logic match_ori(rice_riscv_inst inst_bits);
  return inst_bits ==? {25'bxxxxxxxxxxxxxxxxx110xxxxx, RICE_RISCV_OPCODE_OP_IMM};
endfunction

function automatic logic match_andi(rice_riscv_inst inst_bits);
  return inst_bits ==? {25'bxxxxxxxxxxxxxxxxx111xxxxx, RICE_RISCV_OPCODE_OP_IMM};
endfunction

function automatic logic match_slli(rice_riscv_inst inst_bits);
  return inst_bits ==? {25'b000000xxxxxxxxxxx001xxxxx, RICE_RISCV_OPCODE_OP_IMM};
endfunction

function automatic logic match_srli(rice_riscv_inst inst_bits);
  return inst_bits ==? {25'b000000xxxxxxxxxxx101xxxxx, RICE_RISCV_OPCODE_OP_IMM};
endfunction

function automatic logic match_srai(rice_riscv_inst inst_bits);
  return inst_bits ==? {25'b010000xxxxxxxxxxx101xxxxx, RICE_RISCV_OPCODE_OP_IMM};
endfunction
```

RTL 実装効率化 - デコーダを(ちょっと)自動生成

```
case (1'b1)
  match_lui(inst_bits):
    return get_alu_operation(RICE_CORE_ALU_ADD, RICE_CORE_ALU_SOURCE_IMM_0, RICE_CORE_ALU_SOURCE_IMM);
  match_auipc(inst_bits):
    return get_alu_operation(RICE_CORE_ALU_ADD, RICE_CORE_ALU_SOURCE_PC, RICE_CORE_ALU_SOURCE_IMM);
  match_jal(inst_bits):
    return get_alu_operation(RICE_CORE_ALU_ADD, RICE_CORE_ALU_SOURCE_PC, RICE_CORE_ALU_SOURCE_IMM_4);
  match_jalr(inst_bits):
    return get_alu_operation(RICE_CORE_ALU_ADD, RICE_CORE_ALU_SOURCE_PC, RICE_CORE_ALU_SOURCE_IMM_4);
  match_beq(inst_bits):
    return get_alu_operation(RICE_CORE_ALU_XOR, RICE_CORE_ALU_SOURCE_RS, RICE_CORE_ALU_SOURCE_RS);
  match_bne(inst_bits):
    return get_alu_operation(RICE_CORE_ALU_XOR, RICE_CORE_ALU_SOURCE_RS, RICE_CORE_ALU_SOURCE_RS);
  match_blt(inst_bits):
    return get_alu_operation(RICE_CORE_ALU_LT, RICE_CORE_ALU_SOURCE_RS, RICE_CORE_ALU_SOURCE_RS);
  match_bge(inst_bits):
    return get_alu_operation(RICE_CORE_ALU_LT, RICE_CORE_ALU_SOURCE_RS, RICE_CORE_ALU_SOURCE_RS);
  match_bltu(inst_bits):
    return get_alu_operation(RICE_CORE_ALU_LTU, RICE_CORE_ALU_SOURCE_RS, RICE_CORE_ALU_SOURCE_RS);
  match_bgeu(inst_bits):
    return get_alu_operation(RICE_CORE_ALU_LTU, RICE_CORE_ALU_SOURCE_RS, RICE_CORE_ALU_SOURCE_RS);
  match_addi(inst_bits):
    return get_alu_operation(RICE_CORE_ALU_ADD, RICE_CORE_ALU_SOURCE_RS, RICE_CORE_ALU_SOURCE_IMM);
```

RTL 実装効率化 - CSR 自動生成

- CSR とは何ぞや？
  - Control and Status Registers
  - CPU の動作に必要な情報や、ステータスを保持するレジスタ
    - トラップハンドラのベースアドレス
    - 起こった例外の種類



RTL 実装効率化 - CSR 自動生成

- CSR の仕様
  - レジスタ
    - アドレス
    - アクセス属性
  - ビットフィールド
    - レジスタ内を更に細分化
    - 設定値やステータスが保持される実体

Number	Privilege	Name	Description
			Machine Information Registers
0xF11	MRO	mvendorid	Vendor ID.
0xF12	MRO	marchid	Architecture ID.
0xF13	MRO	mimpid	Implementation ID.
0xF14	MRO	mhartid	Hardware thread ID.
0xF15	MRO	mconfigptr	Pointer to configuration data structure.
			Machine Trap Setup
0x300	MRW	mstatus	Machine status register.
0x301	MRW	misa	ISA and extensions
0x302	MRW	medeleg	Machine exception delegation register.
0x303	MRW	mideleg	Machine interrupt delegation register.
0x304	MRW	mie	Machine interrupt-enable register.
0x305	MRW	mtvec	Machine trap-handler base address.
0x306	MRW	mcounteren	Machine counter enable.
0x310	MRW	mstatush	Additional machine status register, RV32 only.
			Machine Trap Handling
0x340	MRW	mscratch	Scratch register for machine trap handlers.

The mstatus register is an MXLEN-bit read/write register formatted as shown in Figure 3.6 for RV32 and Figure 3.7 for RV64. The mstatus register keeps track of and controls the hart's current operating state. A restricted view of mstatus appears as the sstatus register in the S-level ISA.

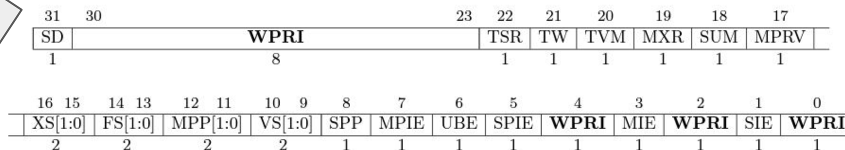


Figure 3.6: Machine-mode status register (mstatus) for RV32.

RTL 実装効率化 - CSR 自動生成

- 手書き実装どうする？
  - やってることはごく簡単
    - アドレスをデコード
    - 書き込みデータを各ビットフィールドの分配
    - 各フィールドから読み出しデータを収集
- ただ、面倒で注意力がいる作業
  - 単純に数が多い
  - アドレスは合っているか？
  - ビット位置が間違っていないか？
  - アクセス属性が合っているか？
- 自動生成するのが幸せ

```
always @(posedge i_clk) begin
  case (i_address)
    // repeat similar code
    'h0000: o_read_data <= foo_setting;
    'h0010: o_read_data <= bar_status;
  endcase
end

// repeat similar code
always @(posedge i_clk) begin
  if (i_address == 'h0000) begin
    foo_setting <= i_write_data;
  end
end
```

RTL 実装効率化 - CSR 自動生成

- CSR 自動生成ツールの導入
  - 拙作の RgGen を使用
- 入力形式として Ruby を採用
  - https://github.com/taichi-ishitani/rice/blob/master/csr/rice_csr_m_level.rb
  - レジスタマップ内で Ruby の構文が使える
    - RV32 と RV64 で初期値やビット位置が異なるビットフィールドがある
    - if 式などRuby の構文を駆使して、差分を吸収
- このレジスタマップから SystemVerilog RTL を生成
  - https://github.com/taichi-ishitani/rice/blob/master/rtl/csr/rice_csr_m_level_xlen32.sv

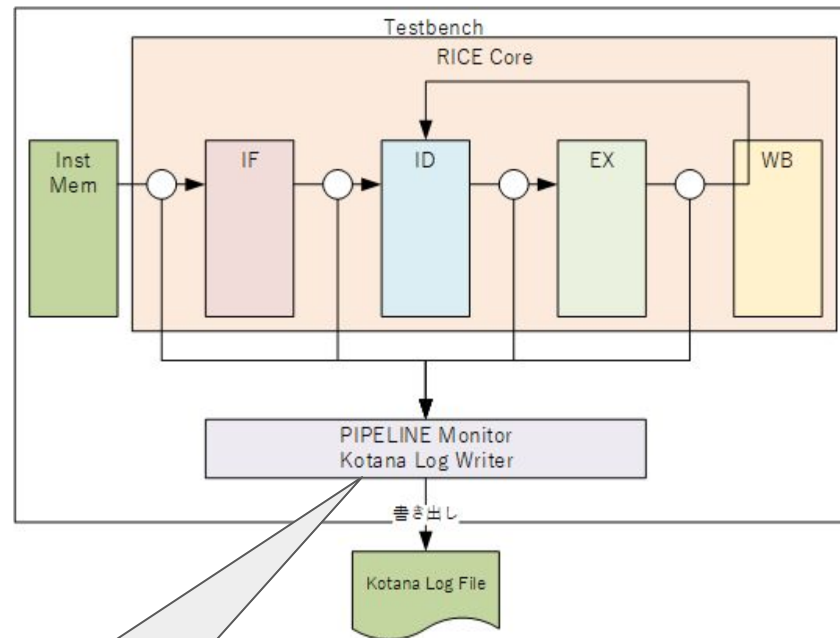
```
register {
  name 'misa'
  offset_address byte_address(0x301)
  type :rw
  bit_field {
    name 'support_e'
    bit_assignment lsb: 4, width: 1; type :ro; reference 'misa.support_i'
  }
  bit_field {
    name 'support_i'
    bit_assignment lsb: 8, width: 1; type :rof; initial_value 1
  }
  bit_field {
    name 'support_m'
    bit_assignment lsb: 12, width: 1; type :rof; initial_value 1
  }
  bit_field {
    name 'user_mode'
    bit_assignment lsb: 20, width: 1; type :rof; initial_value 1
  }
  bit_field {
    name 'mxl'
    bit_assignment lsb: xlen - 2, width: 2; type :rof
    initial_value case xlen
                  when 32 then 0b01
                  else 0b00
                  end
  }
}
```

検証環境 - Konata でパイプラインを良い感じに見る

- Konata とはなんぞや？
  - 東大の塩谷先生開発のパイプラインビューワー
    - https://github.com/shioyadan/Konata
  - パイプラインの実行ログを与えると、良い感じに表示してくれる

検証環境 - Konata でパイプラインを良い感じに見る

- Konata でパイプラインを可視化してみる
  - インストール
    - コンパイル済みバイナリを展開するだけ
      https://github.com/shioyadan/Konata/releases
  - パイプラインモニターをテストベンチに組み込む
    - 各ステージ間を流れる情報を取り込む
      - 掛かったサイクル数
      - PC
      - 命令ビット列
      - デコード結果
      - などなど
    - 取り込んだ情報を Konata 用のログフォーマットで書き出す
  - 出力したログを Konata で開く



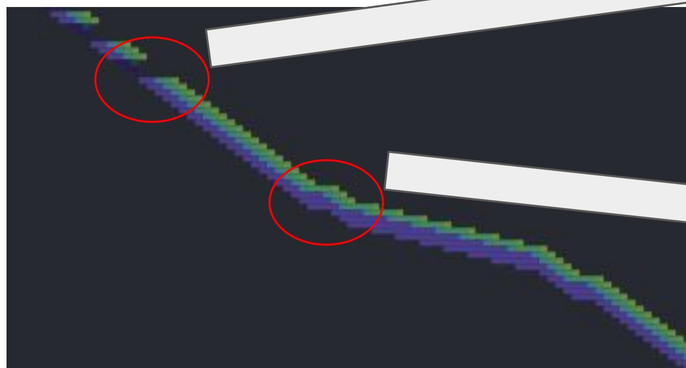
検証環境 - Konata でパイプラインを良い感じに見る

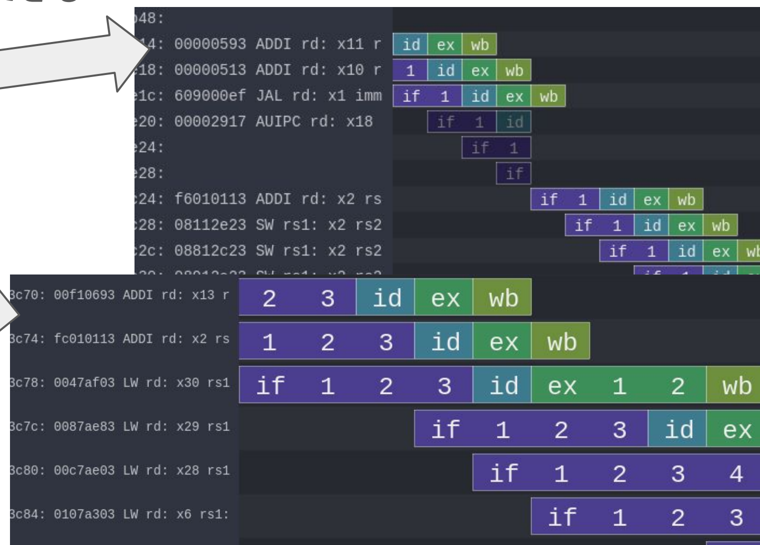
- I
  命令開始
- S
  パイプラインステージ開始
- R
  命令終了
- C
  経過サイクル数
- L
  ラベル

```
Kanata  0004
C=      0
C       13
I       0       0       0
S       0       0       if
C       1
I       1       1       0
S       1       0       if
C       1
I       2       2       0
S       2       0       if
S       0       0       id
C       1
I       3       3       0
S       3       0       if
S       1       0       id
S       0       0       ex
C       1
I       4       4       0
S       4       0       if
S       2       0       id
S       1       0       ex
S       0       0       wb
C       1
I       5       5       0
S       5       0       if
S       3       0       id
S       2       0       ex
S       1       0       wb
L       0       0       80000000: 00000093 ADDI rd: x1 rs1: x0 imm: 00000000
R       0       0       0
C       1
I       6       6       0
S       6       0       if
S       4       0       id
S       3       0       ex
S       2       0       wb
L       1       0       80000004: 00000113 ADDI rd: x2 rs1: x0 imm: 00000000
R       1       0       0
C       1
```

検証環境 - Konata でパイプラインを良い感じに見る

- Konata 導入で嬉しい事
  - 全体を俯瞰することで、パイプラインの流れ具合を目視できる
    - 歯抜けでパイプラインがフラッシュされたことが分かる
    - パイプラインが詰まると、傾きが緩くなる
  - 拡大することで、原因の命令を特定できる



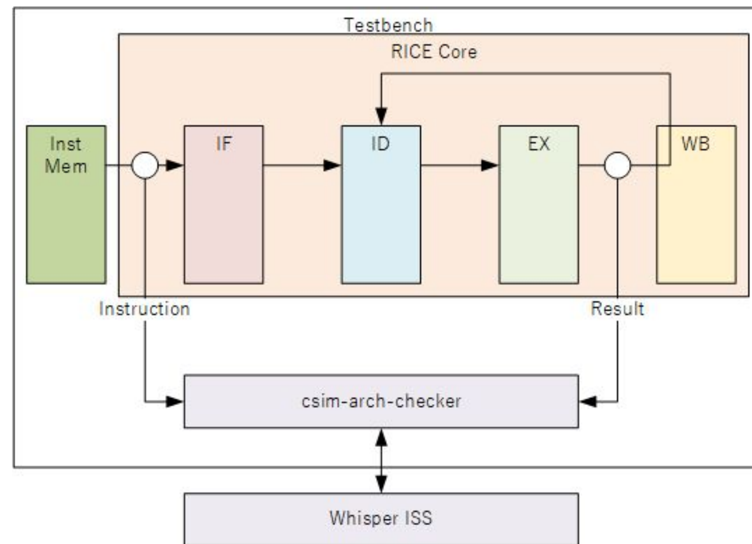


検証環境 - ISS との一致比較

- ISS とはなんぞや
  - Instruction Set Simulator
  - 命令ビット列を与えると、その結果を返してくれるモデル
    - RISC-Vの例
      - SPIKE
        公式様謹製 ISS
      - Whisper
        Tenstorrent 謹製ISS
- ISS を神様にして、作ったCPUが正しく動作しているか確認する

検証環境 - ISS との一致比較

- ISS をテストベンチに組み込む
  - [Whisper](https://github.com/chipsalliance/VeeR-ISS)を神様として採用
  - SystemVerilogとのブリッジ環境が既に用意されていたため
    - https://github.com/tenstorrent/cosim-arch-checker
    - 既存のAPIを叩くだけのお手軽実装
  - シミュレーションの実行中に一致比較を行える




```
function void end_wb(longint cycles, tb_rice_core_env_pipeline
  if (no_error(item)) begin
    cosim_proxy.gpr(cycles, item.rd, item.rd_value);
    if (is_csr_access(item)) begin
      monitor_csr(cycles, item);
    end
  end
  cosim_proxy.instr(cycles, 0, item.pc, item.inst_bits, 0);
endfunction
```

結果の登録

命令を与えて、
結果を一致比較

```
<49> DutGprWr: [Hart=0, Reg=X5, Data=0xfffe2000]
<49> DutRetire: [Hart=0, InstrTag=0x0 (0,0), PC=0x8000007c, Opcode=0x1e2b7]
<49> Whisper Step #31: [Hart=0, InstrTag=0x0, ChangeCount=1, PC=0x8000007c, Opcode=0x1e2b7, lui      x5, 0x1e]
<49> Whisper Step #31: [Hart=0, InstrTag=0x0, Resource=r, Addr=0x5, Data=0x1e000]

Register Mismatch
Step: 31
                X5                   DUT:[Data:00000000fffe2000]
                                     SIM:[Data:000000000001e000]
                PC                   DUT:[Data:000000008000007c]
                                     SIM:[Data:000000008000007c]

Error: Core Arch Checker Mismatch
```

検証環境 - riscv-tests の導入

- 公式様提供の動作確認用のプログラム集
  - https://github.com/riscv-software-src/riscv-tests
- ユニットテスト
  - ある命令が実装されているか確認するためのプログラム集
  - 大まかには、対象命令の実行と、結果と期待値の比較を行うプログラムになっている
    - プログラムがエラー無く完走すれば、致命的な実装ミスがないことが分かる
- ベンチマークテスト
  - dhrystone など性能を測るためのテスト集
    - 分岐予測が効いているなとか、結果を見てムフムフできる

分岐予測適応前

```
syswrite: Microseconds for one run through Dhrystone: 953

syswrite: Dhrystones per Second:                      1049

syswrite: mcycle = 476547
```

分岐予測適応後

```
syswrite: Microseconds for one run through Dhrystone: 704

syswrite: Dhrystones per Second:                      1420

syswrite: mcycle = 352086
```

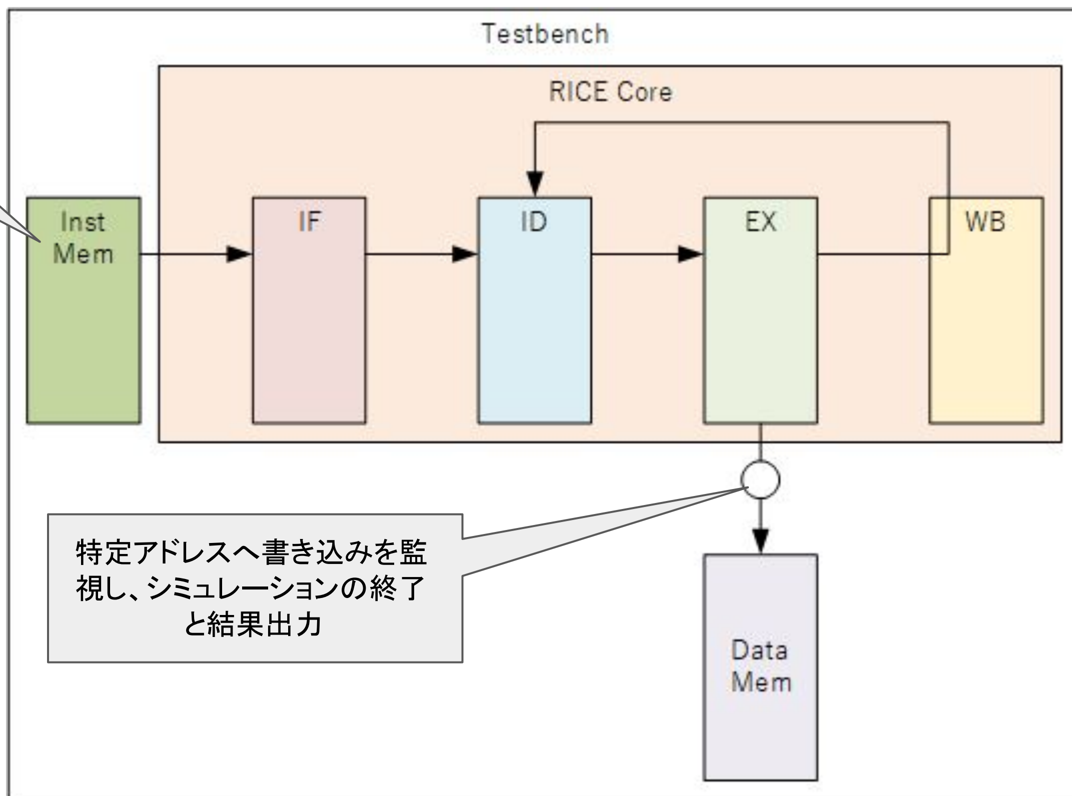
検証環境 - riscv-tests の導入

- riscv-tests を使えるようにしてみる
  - プログラムのコンパイル
    - RISC-V 対応の GCC でコンパイル
    - SystemVerilog 上から読めるようにする為に、objcopy でバイナリ形式に変換
  - テストベンチ側の対応
    - プログラム終了監視
      - 終了時に、結果が特定メモリアドレスに書き込まれる
        - データメモリへの書き込みを監視
        - 当該アドレスへの書き込みがあれば、シミュレーションを終了
      - 書き込みデータから成否を判定し、シミュレーションログに出力

```
0.000ns: uvm_test_top [LOAD_INST_DATA] load inst data from rv32ui-p-add.bin
625.000ns: uvm_test_top [CHECK_RESULT] all tests are passed
```

検証環境 - riscv-tests の導入

コンパイルしたプログラム
を格納する



検証環境 - riscv-tests の導入

- RICEでの適応状況
  - ユニットテスト・ベンチマークの計 69 本のテストが通ることを確認済み

```
taichi@LAPTOP-TVTKLNFD:core
$ ls -d riscv-tests_*
riscv-tests_benchmarks_dhrystone        riscv-tests_isa_rv32ui-p-and        riscv-tests_isa_rv32ui-p-simple
riscv-tests_benchmarks_median           riscv-tests_isa_rv32ui-p-andi       riscv-tests_isa_rv32ui-p-sll
riscv-tests_benchmarks_memcpy           riscv-tests_isa_rv32ui-p-auipc      riscv-tests_isa_rv32ui-p-slli
riscv-tests_benchmarks_multiply         riscv-tests_isa_rv32ui-p-beq        riscv-tests_isa_rv32ui-p-slt
riscv-tests_benchmarks_qsort            riscv-tests_isa_rv32ui-p-bge        riscv-tests_isa_rv32ui-p-slti
riscv-tests_benchmarks_rsort            riscv-tests_isa_rv32ui-p-bgeu       riscv-tests_isa_rv32ui-p-sltiu
riscv-tests_benchmarks_spmv             riscv-tests_isa_rv32ui-p-blt        riscv-tests_isa_rv32ui-p-sltu
riscv-tests_benchmarks_towers           riscv-tests_isa_rv32ui-p-bltu       riscv-tests_isa_rv32ui-p-sra
riscv-tests_benchmarks_vvadd            riscv-tests_isa_rv32ui-p-bne        riscv-tests_isa_rv32ui-p-srai
riscv-tests_isa_rv32mi-p-csr            riscv-tests_isa_rv32ui-p-fence_i    riscv-tests_isa_rv32ui-p-srl
riscv-tests_isa_rv32mi-p-lh-misaligned  riscv-tests_isa_rv32ui-p-jal        riscv-tests_isa_rv32ui-p-srli
riscv-tests_isa_rv32mi-p-lw-misaligned  riscv-tests_isa_rv32ui-p-jalr       riscv-tests_isa_rv32ui-p-sub
riscv-tests_isa_rv32mi-p-ma_addr        riscv-tests_isa_rv32ui-p-lb         riscv-tests_isa_rv32ui-p-sw
riscv-tests_isa_rv32mi-p-ma_fetch       riscv-tests_isa_rv32ui-p-lbu        riscv-tests_isa_rv32ui-p-xor
riscv-tests_isa_rv32mi-p-mcsr           riscv-tests_isa_rv32ui-p-lh         riscv-tests_isa_rv32ui-p-xori
riscv-tests_isa_rv32mi-p-sbreak         riscv-tests_isa_rv32ui-p-lhu        riscv-tests_isa_rv32um-p-div
riscv-tests_isa_rv32mi-p-scall          riscv-tests_isa_rv32ui-p-lui        riscv-tests_isa_rv32um-p-divu
riscv-tests_isa_rv32mi-p-sh-misaligned  riscv-tests_isa_rv32ui-p-lw         riscv-tests_isa_rv32um-p-mul
riscv-tests_isa_rv32mi-p-shamt          riscv-tests_isa_rv32ui-p-ma_data    riscv-tests_isa_rv32um-p-mulh
riscv-tests_isa_rv32mi-p-sw-misaligned  riscv-tests_isa_rv32ui-p-or         riscv-tests_isa_rv32um-p-mulhsu
riscv-tests_isa_rv32mi-p-zicntr         riscv-tests_isa_rv32ui-p-ori        riscv-tests_isa_rv32um-p-mulhu
riscv-tests_isa_rv32ui-p-add            riscv-tests_isa_rv32ui-p-sb         riscv-tests_isa_rv32um-p-rem
riscv-tests_isa_rv32ui-p-addi           riscv-tests_isa_rv32ui-p-sh         riscv-tests_isa_rv32um-p-remu
```

おしまい