2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION TAIWAN

Adopts ISA-Formal On High-End Out-Of-Order Execute RISC-V Cores

HUANG YU-TSE, WU SHENG-JHAN, HSIAO YUNG-CHING

Agenda

What's an ISA-Formal?

- ISA-Formal
  - An end-to-end framework to verify the core conform to the Instruction-Set Architecture (ISA) spec.
  - bounded model checking (BMC) to explore different sequences of instructions.
- Pros and Cons

Pros

- Single Instruction
  - Error in decode
  - Error in data path
- Multiple Instruction
  - Errors in forwarding logic
  - Errors in register renaming and OoO execution
  - Errors in exception trigger

Cons

- FP computation
- Mul/Div operation
- Vector operation
- Crypto operation

ISA view

- In the ISA view, instructions are seen in a very straightforward manner
- Instruction is considered a state transformation function.

$$\begin{tikzcd}
State_0 \arrow[r, "Instr_0"] & State_1 \arrow[r, "Instr_1"] & State_2 \arrow[r, "Instr_2"] & \ldots\ldots \arrow[r, "Instr_n"] & State_{n+1}
\end{tikzcd}$$

- Each instruction contains an execution body.
  - E.g. ADDI x2, x1, 10

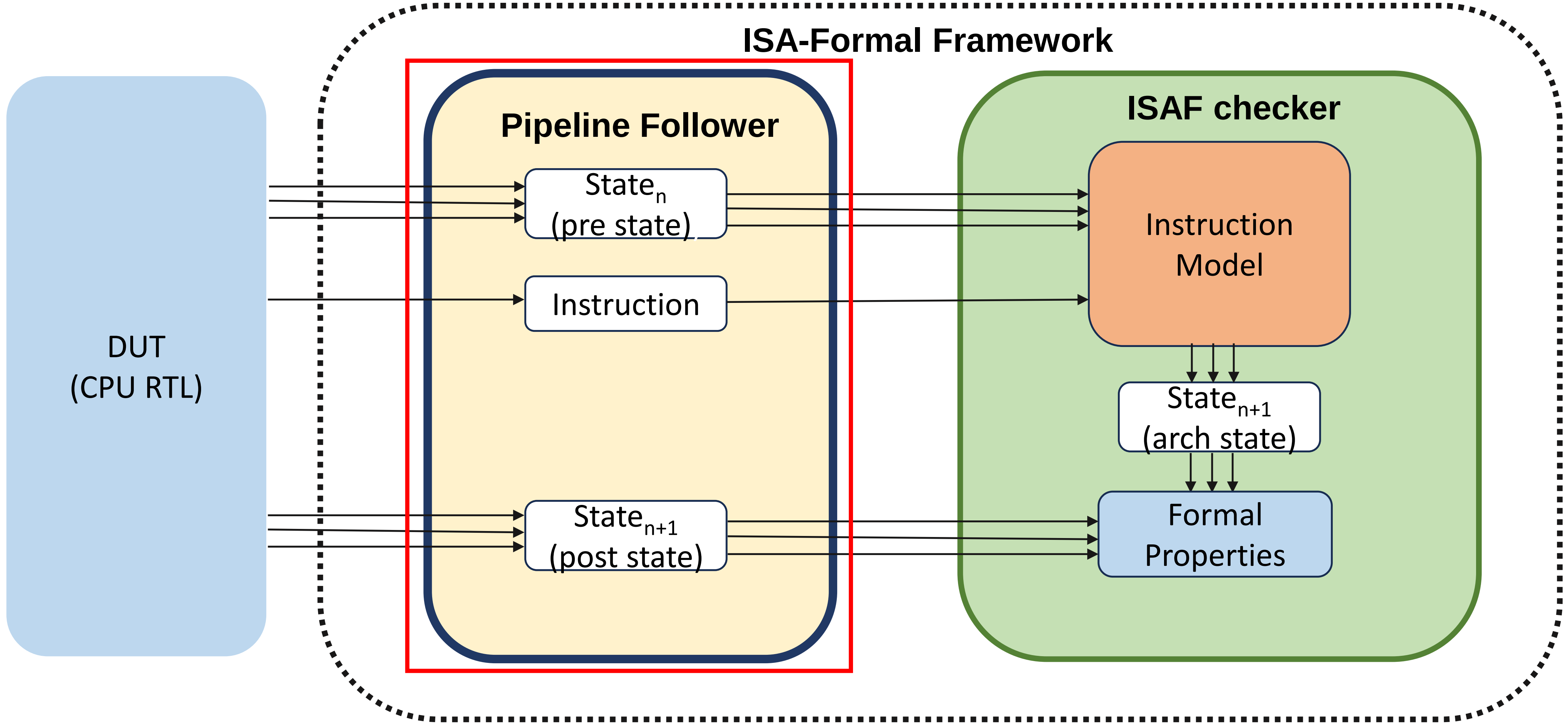
$$\begin{tikzcd}
State_0 \arrow[r, "\textit{ADDI x2,x1, 10}"] & State_1
\end{tikzcd}$$

$State_1.x2 = State_0.\ x1+10$

ISA-Formal Key Component

- Golden instruction Model
  - Calculate the expected result.
  - Must be synthesizable.
- Pipeline follower
  - Collect necessary CPU information for building architectural state.
  - Determine when to activate Instruction model and check.
- Formal Properties
  - Assertion properties to verify the instruction result correctness.
    - E.g. when an addi instruction retired implies GPRs of DUT should be same to GPRs of architectural state.
  - Constraint properties to remove unsolvable scenarios in ISA-Formal.
    - E.g. asynchronous interrupts.

ISA-Formal Overview



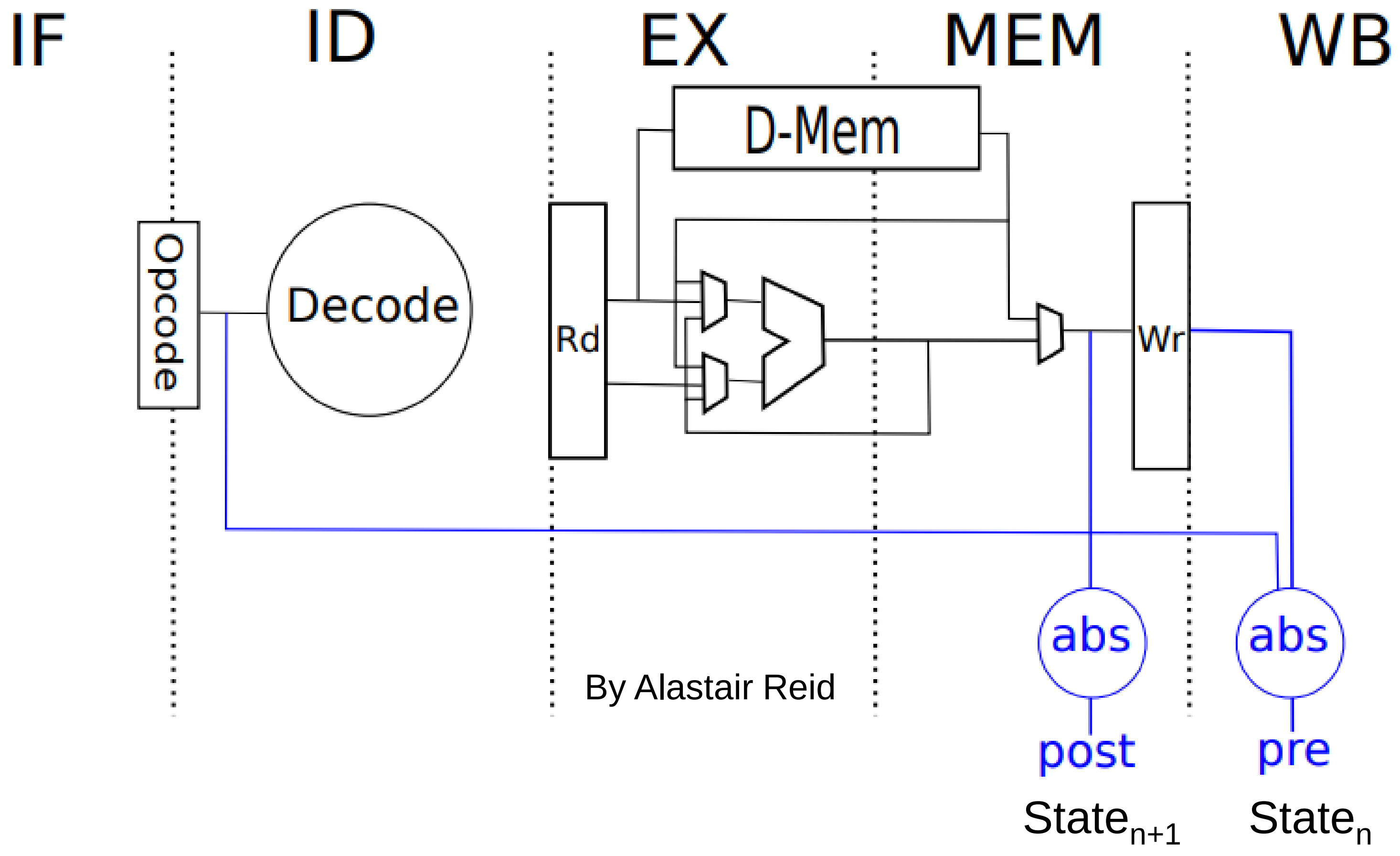
Structure of CPU state

- General purpose registers(GPRs)
- Floating point registers
- Vector registers
- Control and status registers(CSRs)
- Privilege
- Virtual mode
- Trap

Pipeline-Follower

What do we need for verifying ISA architecture?

- Instruction opcode
- State before the instruction execution(pre_state)
- State after the instruction execution by DUT(post_state)



By Alastair Reid

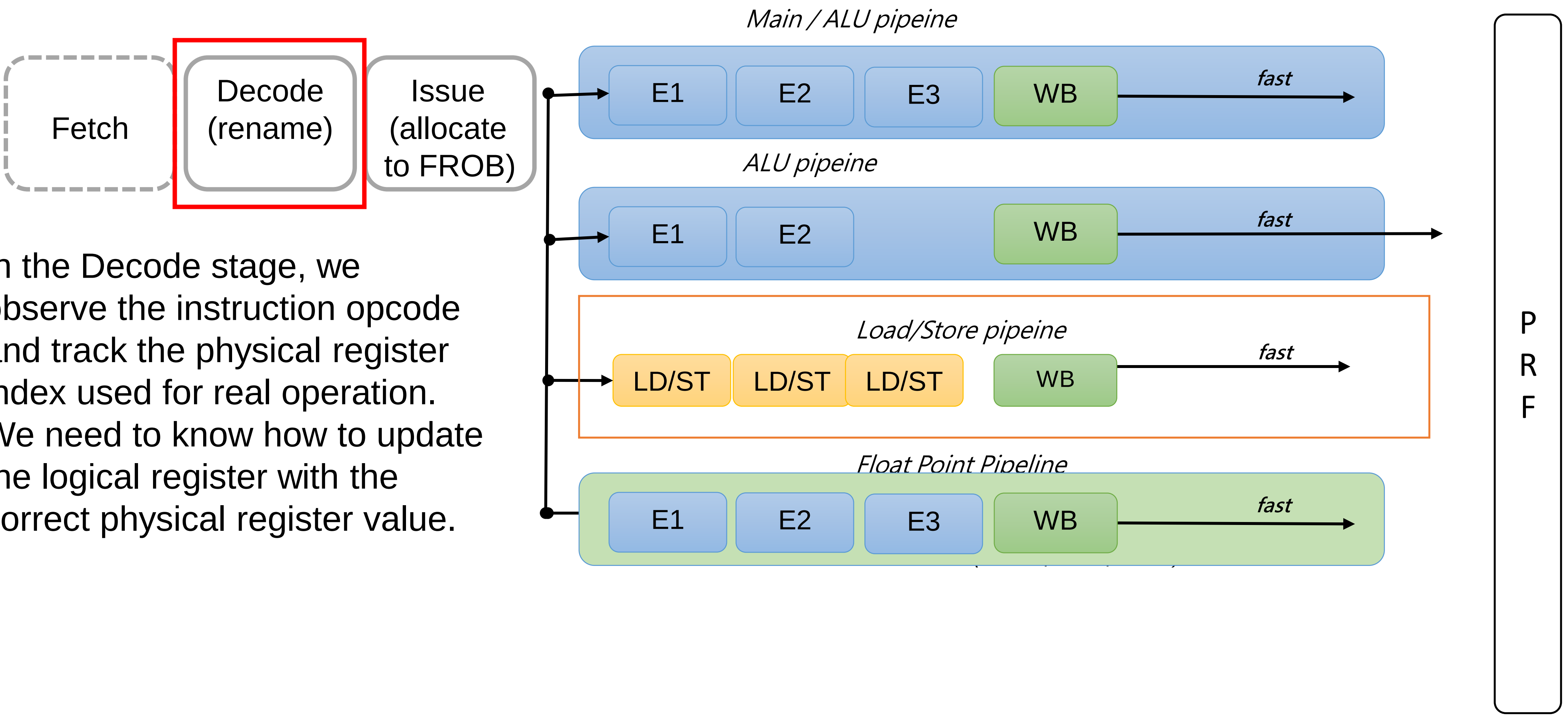
OoO CPU state abstraction

- Decode and Register rename stage
  - Collect instruction operand information
    - Rd, Rs1, Rs2, Immediate value
  - Map the logic register to physical register.
  - Mark Instruction with instruction id.
- Formal Reorder Buffer (FROB)
  - Ensure the instruction committed in order.
  - Observe issued instruction status.
    - Instruction age
    - completed
    - committed
    - Trap
    - Physical register index

ROB field	Note
completed	Instruction complete
commited	Instruction committed

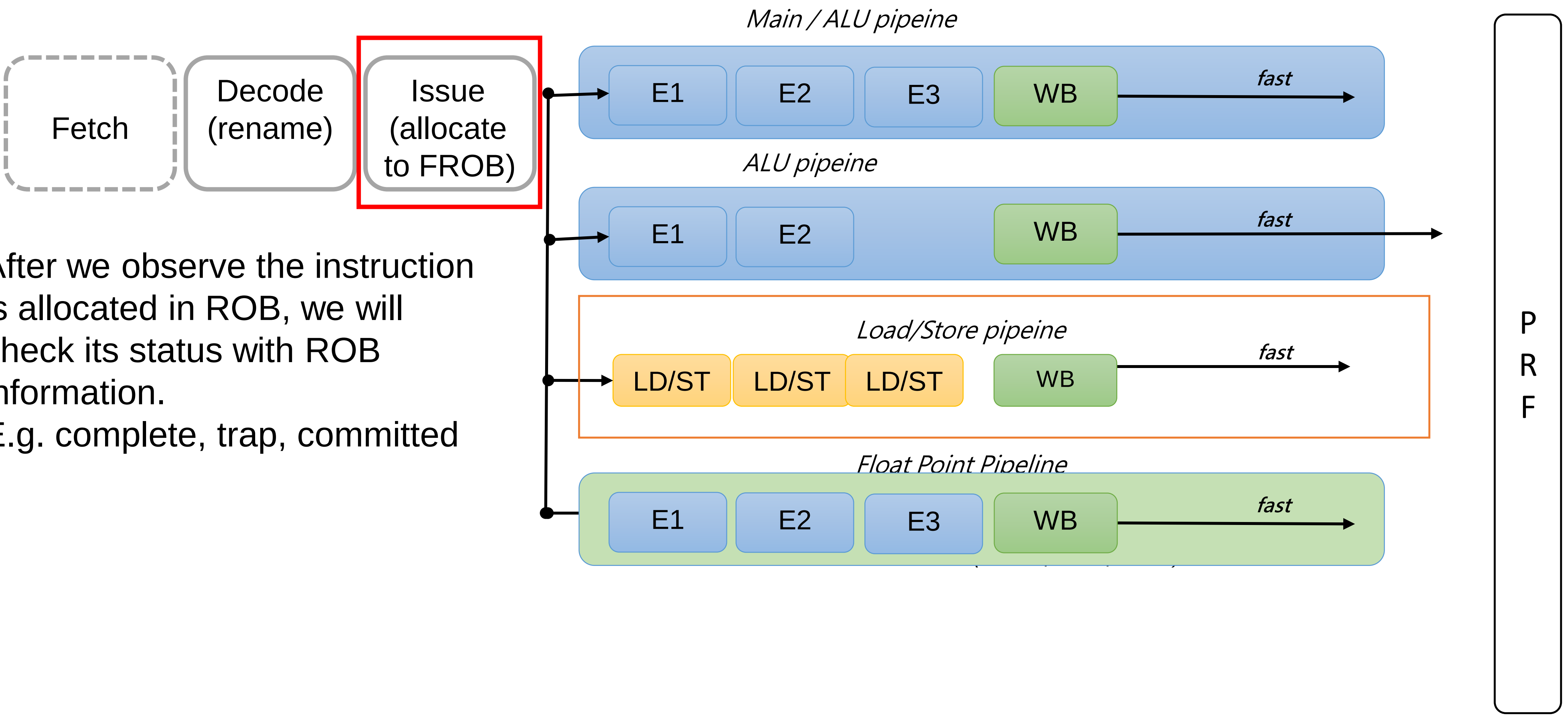
ROB field	Note
age	Instruction Age
pc	PC counter
ird	Logical register Index
irs1	
Irs2	
Prd	Physical register Index.
Prs2	
prs2	

OoO Pipeline-Follower



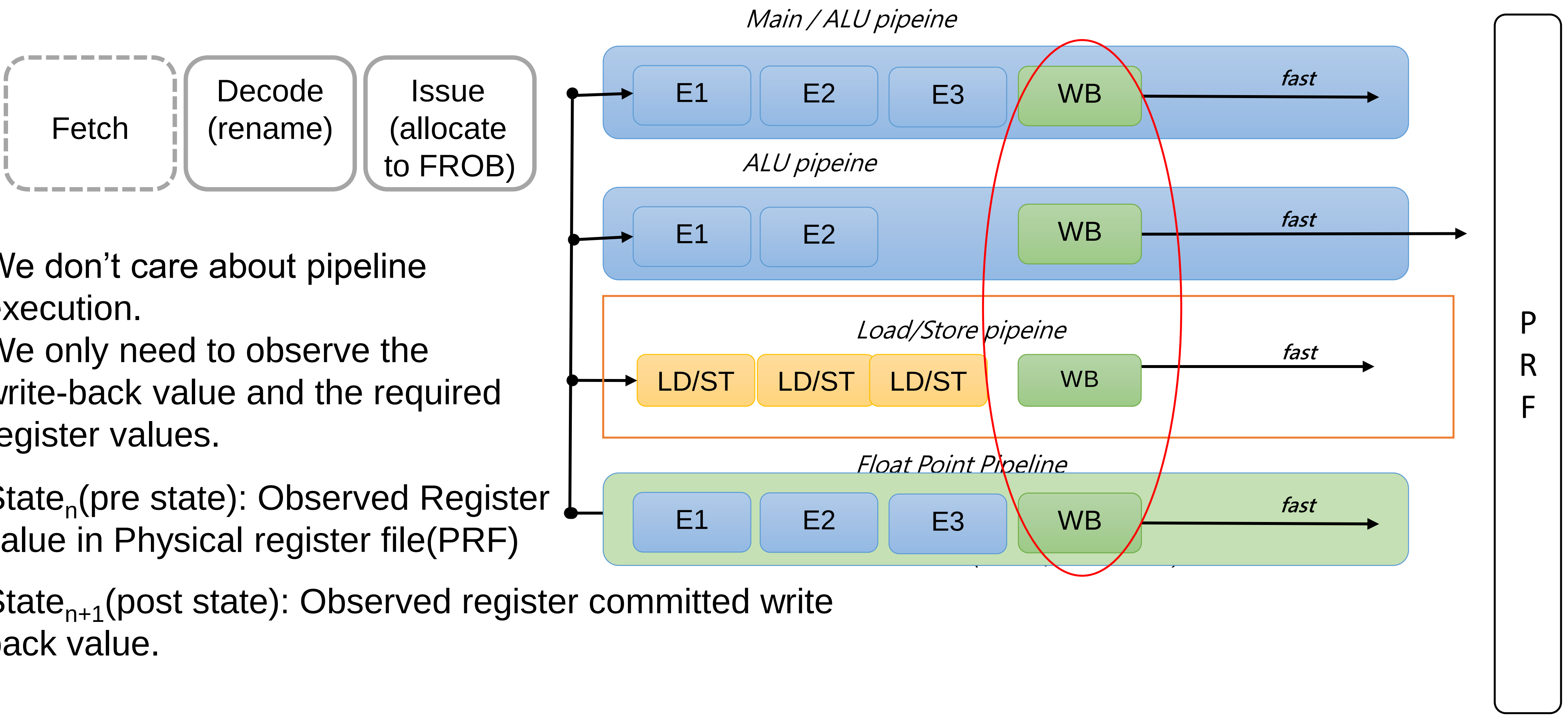
In the Decode stage, we observe the instruction opcode and track the physical register index used for real operation. We need to know how to update the logical register with the correct physical register value.

OoO Pipeline-Follower



After we observe the instruction is allocated in ROB, we will check its status with ROB information.
E.g. complete, trap, committed

OoO Pipeline-Follower

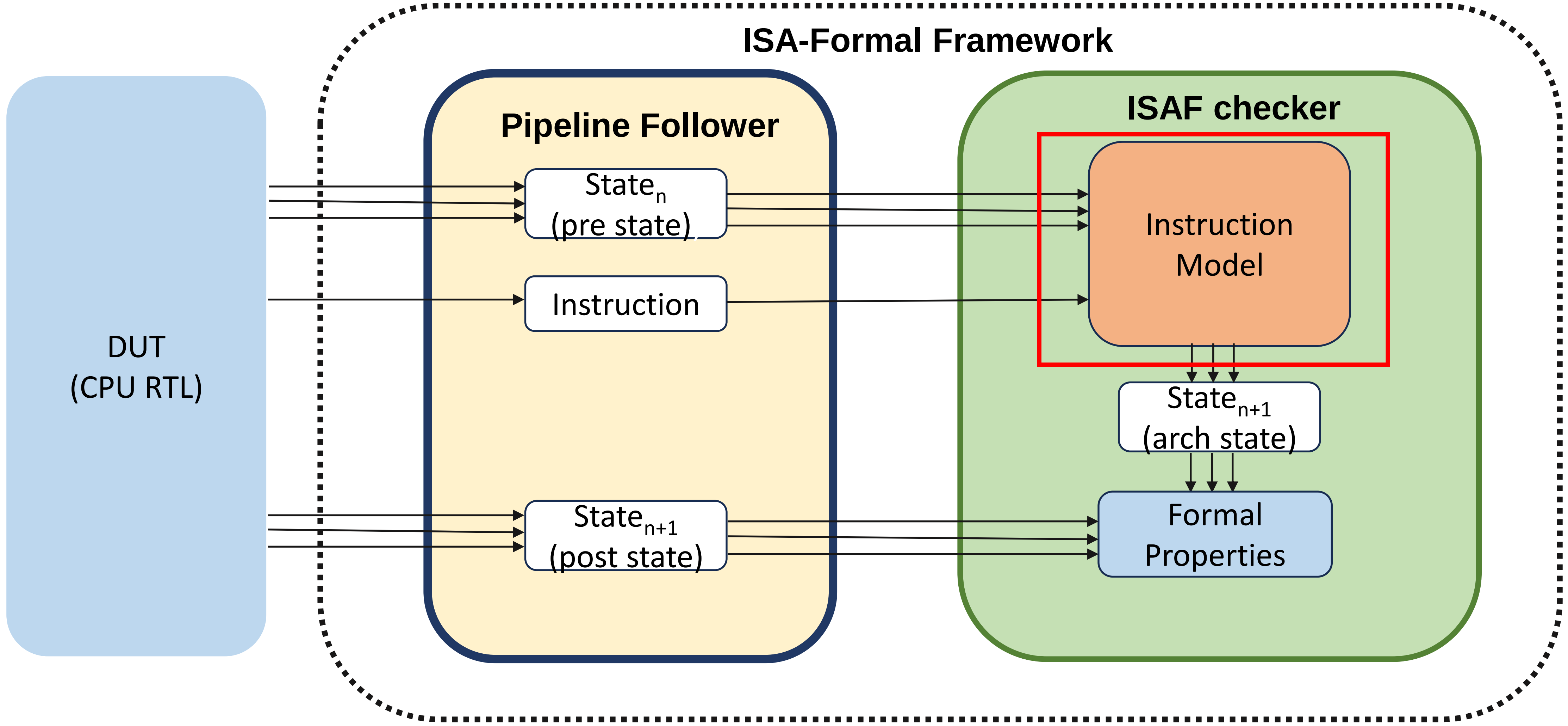


We don’t care about pipeline execution.
We only need to observe the write-back value and the required register values.

State_n(pre state): Observed Register value in Physical register file(PRF)

State_{n+1}(post state): Observed register committed write back value.

ISA-Formal Overview



Golden Instruction model

- ADDI Instruction model by handwriting

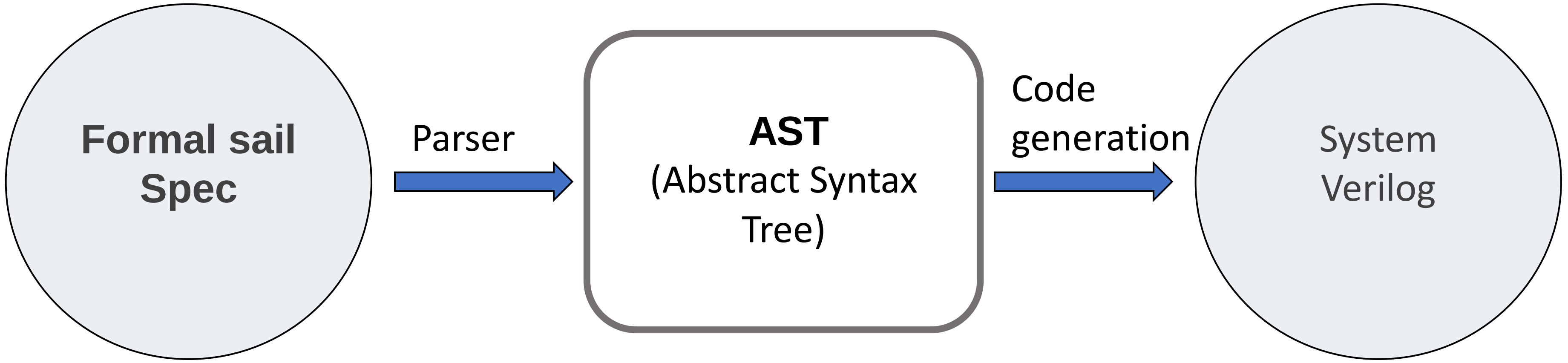
31………20	19……15	14…12	11……..7	6………. 0
imm12	**rs1**	000	rd	0010011

```
assign ADDI_retiring = ( instr & 32’h707F ) == 32’h13;
assign ADDI_result =  preState.GPR[instr[19:15]] + instr[31:20];
assign ADDI_rd = instr[11:7];


assert property (@(posedge clk) disable iff (~reset_n)
 ADDI_retiring |-> (ADDI_result == postState.GPR[ADDI_rd]))
```

Too many !!

Golden Instruction model

- How to efficiently generate enormous amount instructions?
  - Utilized the existed open-source Sail-riscv project to transform the Sail specifications into Verilog, creating an accurate instruction model.



Sail

- A language for defining the instruction-set architecture semantics.
  - engineer-friendly
  - Numerous type system

```
register PC : xlenbits
register nextPC : xlenbits

register Xs : vector(32, dec, xlenbits)

val rX : regbits -> xlenbits
function rX(r) =
  match r {
    0b00000 => EXTZ(0x0),
    _ => Xs[unsigned(r)]
  }
```

```
type xlen       : Int = 64
type xlen_bytes : Int = 8
type xlenbits         = bits(xlen)
```

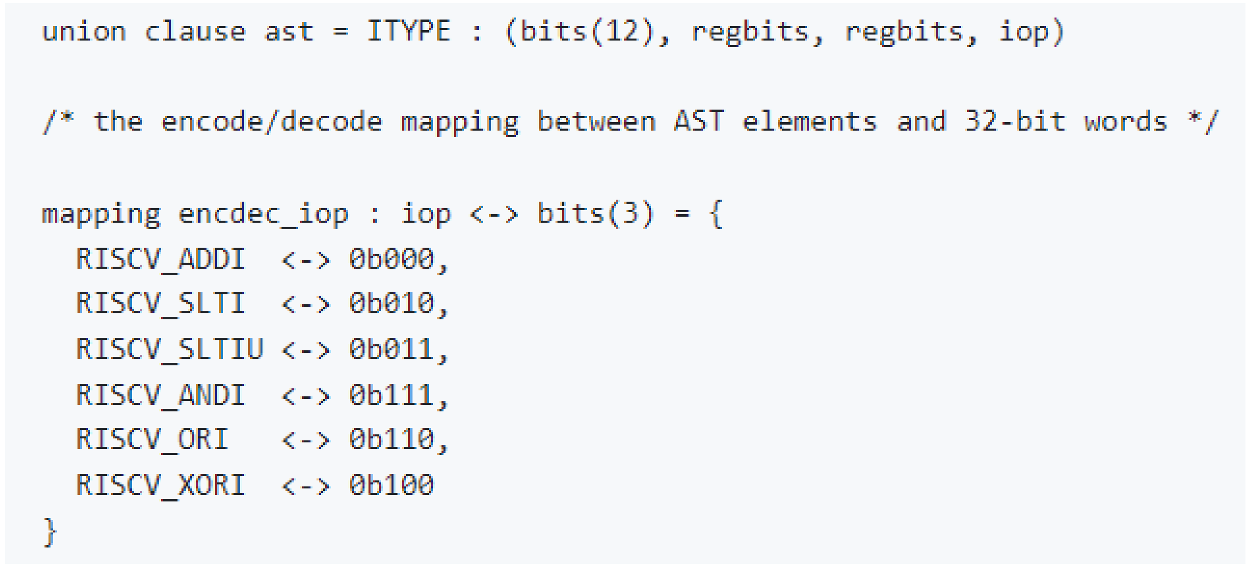
- Given a Sail ISA specification, the tool can:
  - Conduct type-check
  - Generate executable emulators in C or Ocaml
  - Generate a reference ISA model in SystemVerilog

Sail-riscv

- Formal specification of the RISC-V architecture, written in Sail.
- This project Specifies RISC-V ratified extensions in detail.
    - Instruction behavior
    - Control and status registers
    - General purpose registers
    - Machine Privilege
    - Virtual address translation
    - Parameterized over platform-specific options

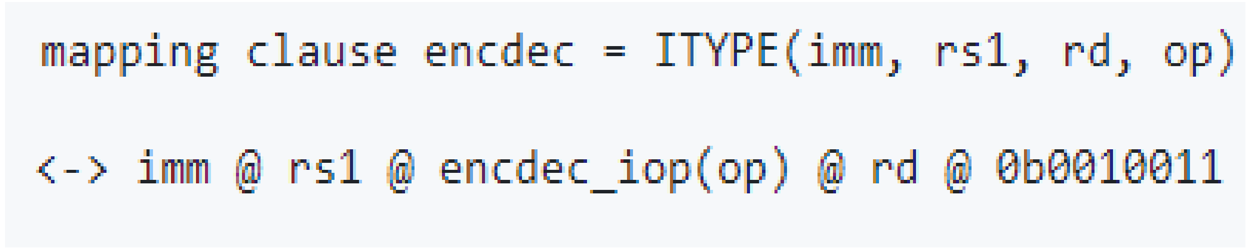
Sail-riscv

Abstract syntax tree

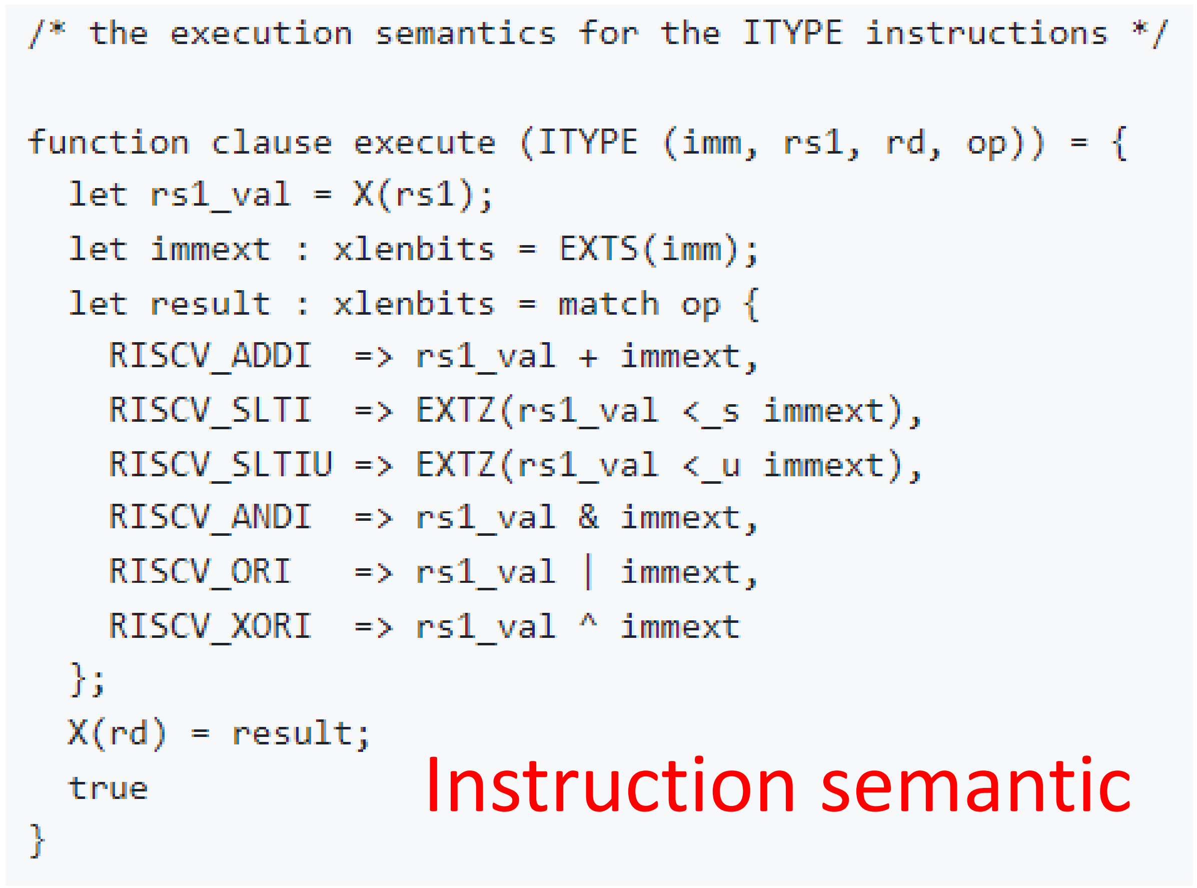


```
union clause ast = ITYPE : (bits(12), regbits, regbits, iop)

/* the encode/decode mapping between AST elements and 32-bit words */

mapping encdec_iop : iop <-> bits(3) = {
  RISCV_ADDI  <-> 0b000,
  RISCV_SLTI  <-> 0b010,
  RISCV_SLTIU <-> 0b011,
  RISCV_ANDI  <-> 0b111,
  RISCV_ORI   <-> 0b110,
  RISCV_XORI  <-> 0b100
}
```

Encoding/Decoding



```
mapping clause encdec = ITYPE(imm, rs1, rd, op)

<-> imm @ rs1 @ encdec_iop(op) @ rd @ 0b0010011
```

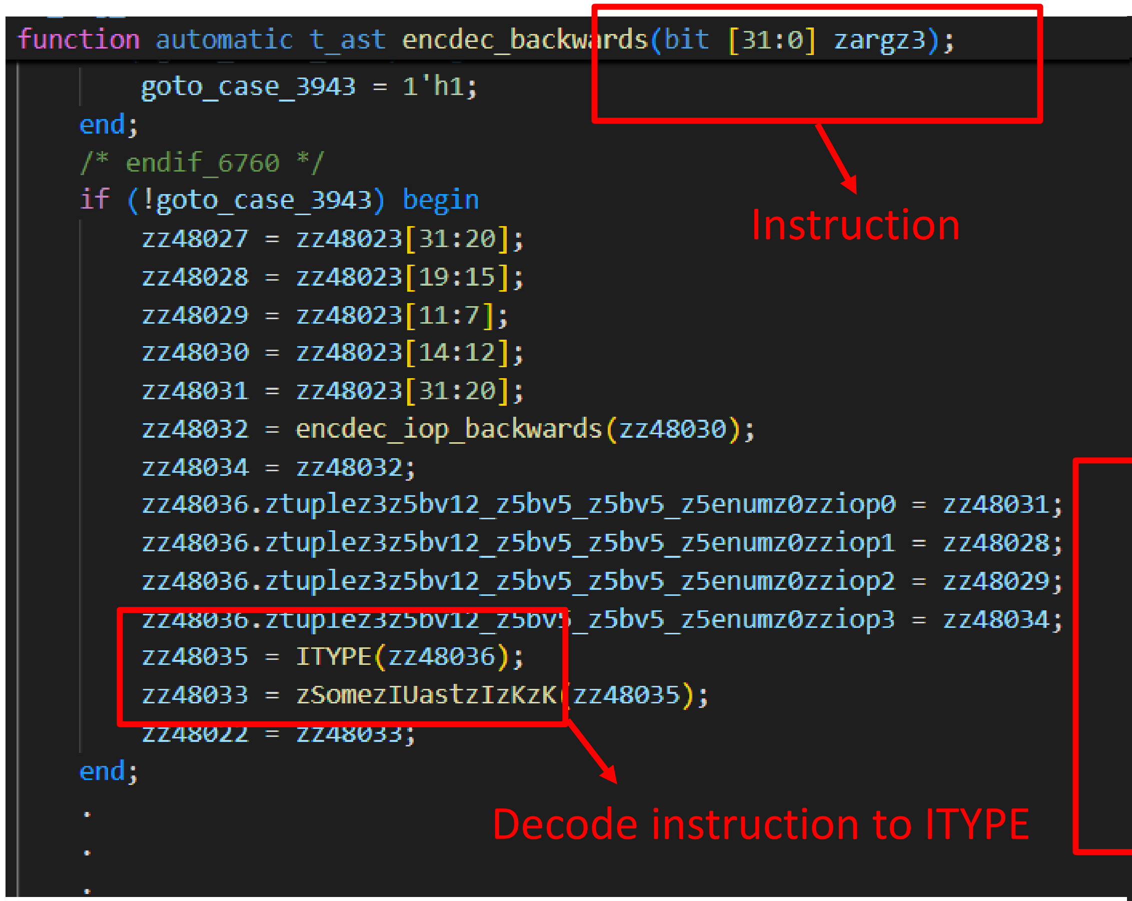


```
/* the execution semantics for the ITYPE instructions */

function clause execute (ITYPE (imm, rs1, rd, op)) = {
  let rs1_val = X(rs1);
  let immext : xlenbits = EXTS(imm);
  let result : xlenbits = match op {
    RISCV_ADDI  => rs1_val + immext,
    RISCV_SLTI  => EXTZ(rs1_val <_s immext),
    RISCV_SLTIU => EXTZ(rs1_val <_u immext),
    RISCV_ANDI  => rs1_val & immext,
    RISCV_ORI   => rs1_val | immext,
    RISCV_XORI  => rs1_val ^ immext
  };
  X(rd) = result;
  true
}
```

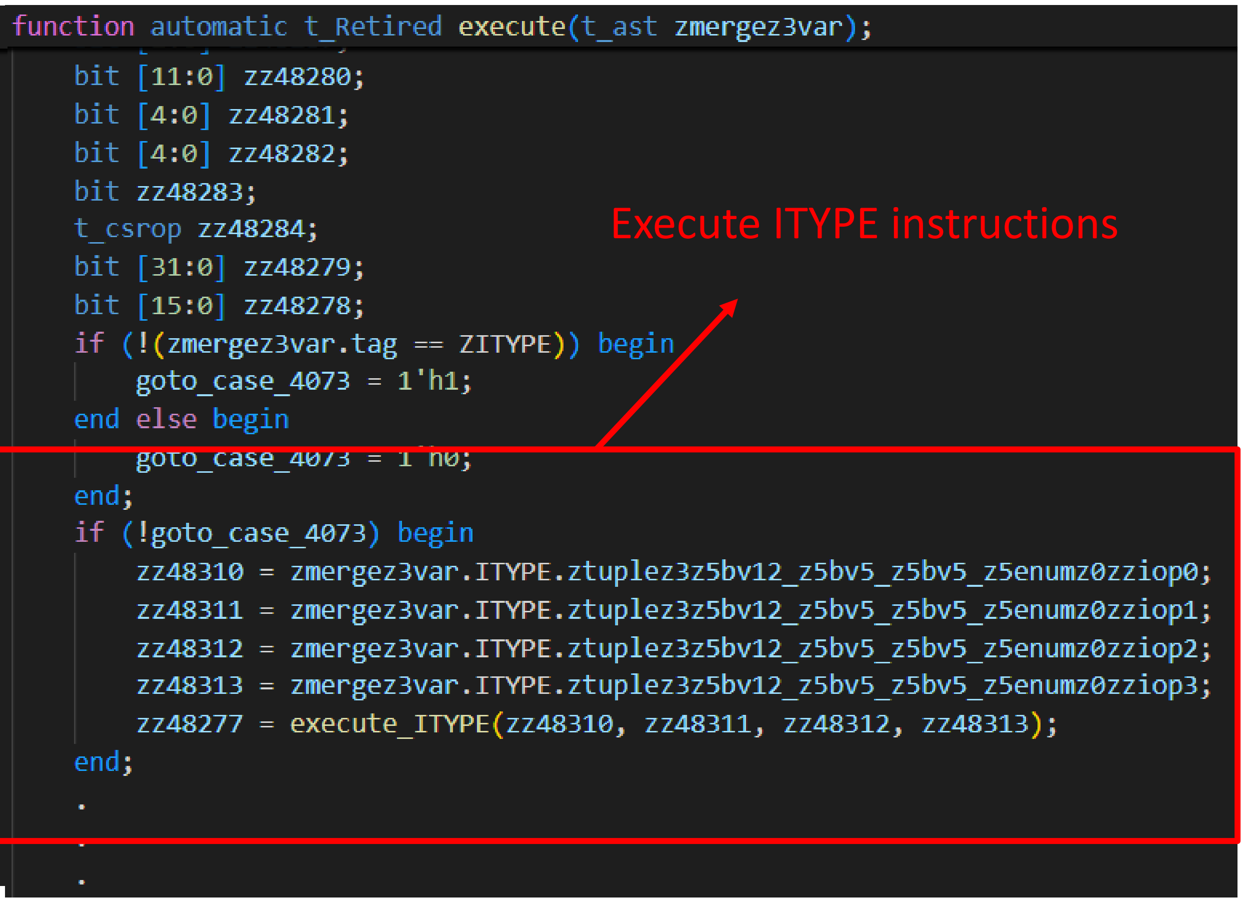
Instruction semantic

Sail to SystemVerilog

- Transform formal specification to Combinational SystemVerilog.



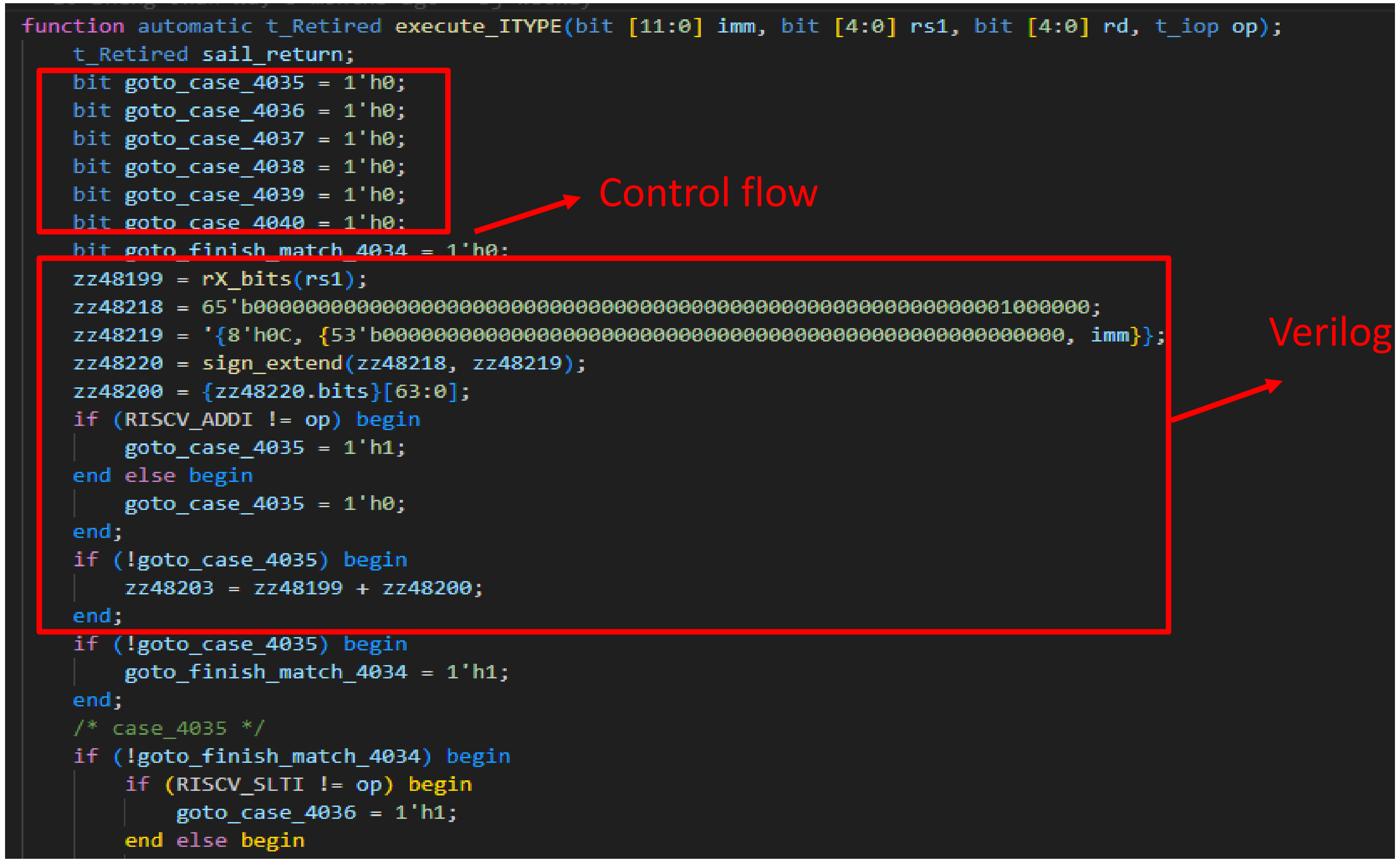
```
function automatic t_ast encdec_backwards(bit [31:0] zargz3);
            goto_case_3943 = 1'h1;
        end;
        /* endif_6760 */
        if (!goto_case_3943) begin
            zz48027 = zz48023[31:20];
            zz48028 = zz48023[19:15];
            zz48029 = zz48023[11:7];
            zz48030 = zz48023[14:12];
            zz48031 = zz48023[31:20];
            zz48032 = encdec_iop_backwards(zz48030);
            zz48034 = zz48032;
            zz48036.ztuplez3z5bv12_z5bv5_z5bv5_z5enumz0zziop0 = zz48031;
            zz48036.ztuplez3z5bv12_z5bv5_z5bv5_z5enumz0zziop1 = zz48028;
            zz48036.ztuplez3z5bv12_z5bv5_z5bv5_z5enumz0zziop2 = zz48029;
            zz48036.ztuplez3z5bv12_z5bv5_z5bv5_z5enumz0zziop3 = zz48034;
            zz48035 = ITYPE(zz48036);
            zz48033 = zSomezIUastzIzKzK(zz48035);
            zz48022 = zz48033;
        end;
        .
        .
        .
```



```
function automatic t_Retired execute(t_ast zmergez3var);
        bit [11:0] zz48280;
        bit [4:0] zz48281;
        bit [4:0] zz48282;
        bit zz48283;
        t_csrop zz48284;
        bit [31:0] zz48279;
        bit [15:0] zz48278;
        if (!(zmergez3var.tag == ZITYPE)) begin
            goto_case_4073 = 1'h1;
        end else begin
            goto_case_4073 = 1'h0;
        end;
        if (!goto_case_4073) begin
            zz48310 = zmergez3var.ITYPE.ztuplez3z5bv12_z5bv5_z5bv5_z5enumz0zziop0;
            zz48311 = zmergez3var.ITYPE.ztuplez3z5bv12_z5bv5_z5bv5_z5enumz0zziop1;
            zz48312 = zmergez3var.ITYPE.ztuplez3z5bv12_z5bv5_z5bv5_z5enumz0zziop2;
            zz48313 = zmergez3var.ITYPE.ztuplez3z5bv12_z5bv5_z5bv5_z5enumz0zziop3;
            zz48277 = execute_ITYPE(zz48310, zz48311, zz48312, zz48313);
        end;
        .
        .
        .
```

Sail to SystemVerilog

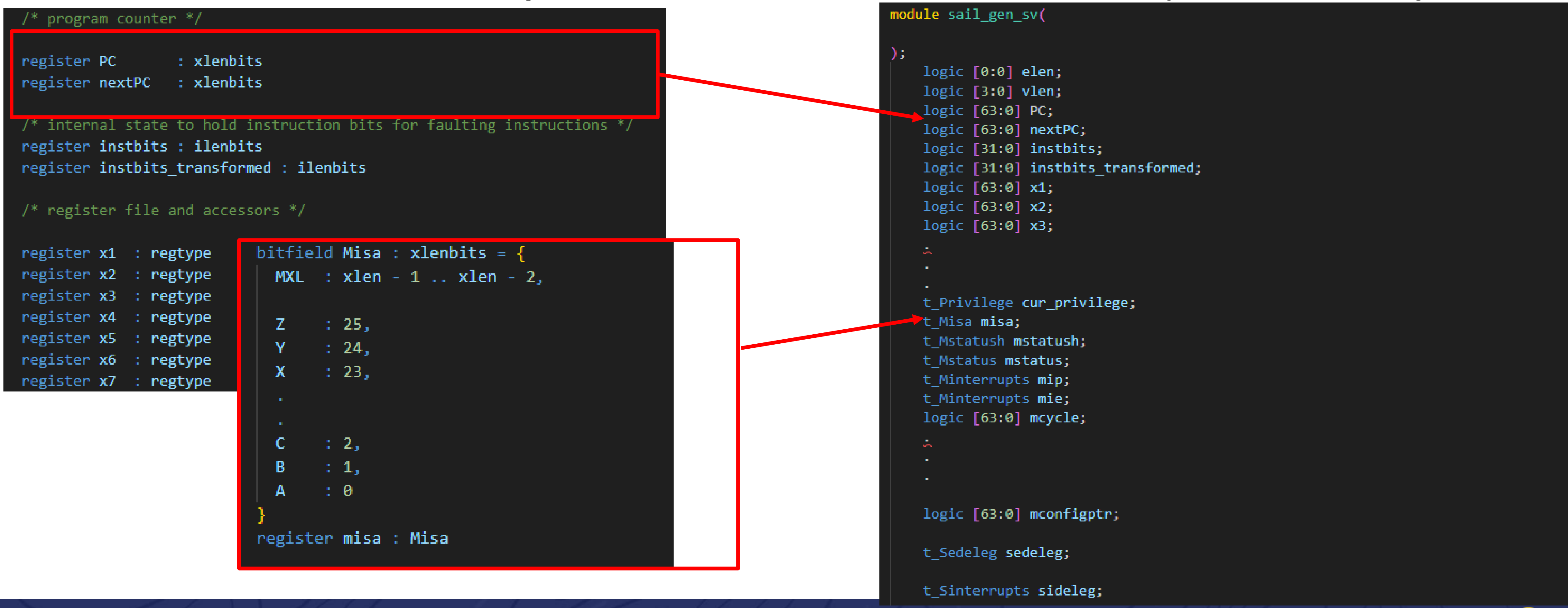
- Transform formal specification to Combinational SystemVerilog.



```
function automatic t_Retired execute_ITYPE(bit [11:0] imm, bit [4:0] rs1, bit [4:0] rd, t_iop op);
    t_Retired sail_return;
    bit goto_case_4035 = 1'h0;
    bit goto_case_4036 = 1'h0;
    bit goto_case_4037 = 1'h0;
    bit goto_case_4038 = 1'h0;
    bit goto_case_4039 = 1'h0;
    bit goto_case_4040 = 1'h0;
    bit goto_finish_match_4034 = 1'h0;
    zz48199 = rX_bits(rs1);
    zz48218 = 65'b00000000000000000000000000000000000000000000000000000000001000000;
    zz48219 = '{8'h0C, {53'b00000000000000000000000000000000000000000000000000000, imm}};
    zz48220 = sign_extend(zz48218, zz48219);
    zz48200 = {zz48220.bits}[63:0];
    if (RISCV_ADDI != op) begin
        goto_case_4035 = 1'h1;
    end else begin
        goto_case_4035 = 1'h0;
    end;
    if (!goto_case_4035) begin
        zz48203 = zz48199 + zz48200;
    end;
    if (!goto_case_4035) begin
        goto_finish_match_4034 = 1'h1;
    end;
    /* case_4035 */
    if (!goto_finish_match_4034) begin
        if (RISCV_SLTI != op) begin
            goto_case_4036 = 1'h1;
        end else begin
```

Sail to SystemVerilog

- Transform formal specification to Combinational SystemVerilog.



```
/* program counter */

register PC       : xlenbits
register nextPC   : xlenbits

/* internal state to hold instruction bits for faulting instructions */
register instbits : ilenbits
register instbits_transformed : ilenbits

/* register file and accessors */

register x1  : regtype
register x2  : regtype
register x3  : regtype
register x4  : regtype
register x5  : regtype
register x6  : regtype
register x7  : regtype
```

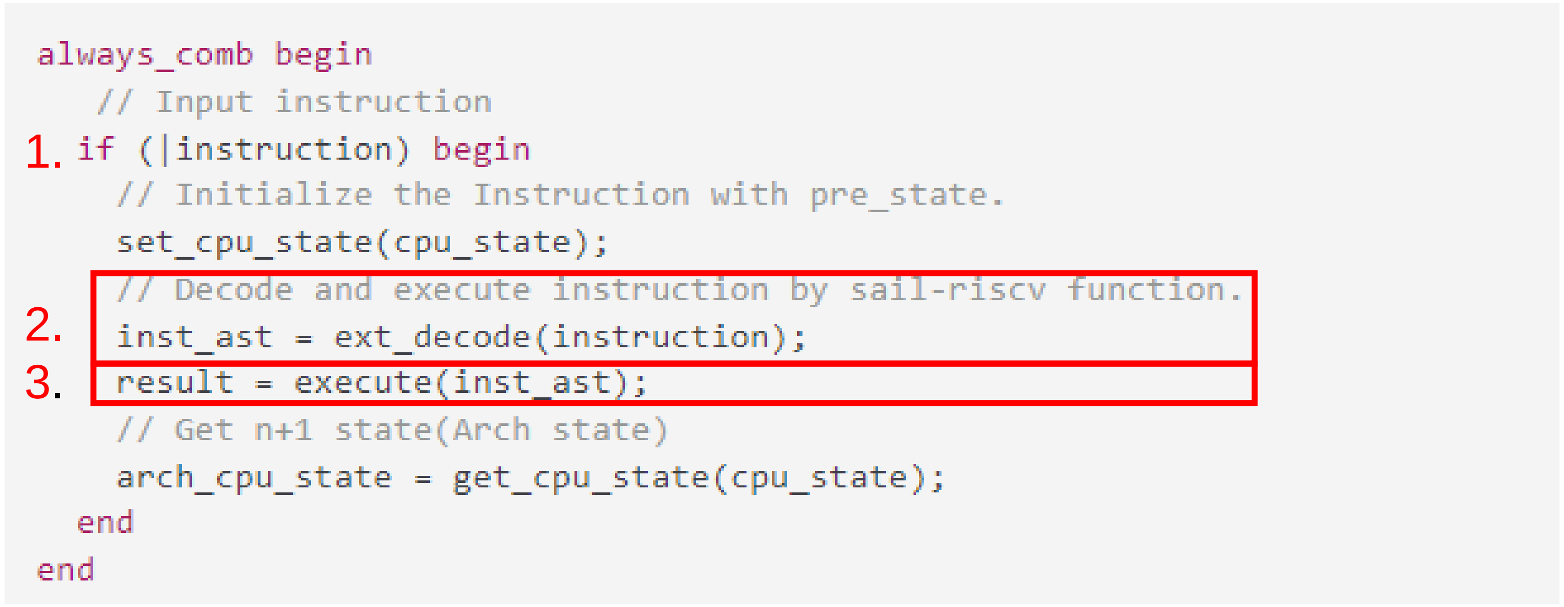
```
bitfield Misa : xlenbits = {
  MXL  : xlen - 1 .. xlen - 2,

  Z    : 25,
  Y    : 24,
  X    : 23,
  .
  .
  C    : 2,
  B    : 1,
  A    : 0
}
register misa : Misa
```

```
module sail_gen_sv(

);
    logic [0:0] elen;
    logic [3:0] vlen;
    logic [63:0] PC;
    logic [63:0] nextPC;
    logic [31:0] instbits;
    logic [31:0] instbits_transformed;
    logic [63:0] x1;
    logic [63:0] x2;
    logic [63:0] x3;
    .
    .
    .
    t_Privilege cur_privilege;
    t_Misa misa;
    t_Mstatush mstatush;
    t_Mstatus mstatus;
    t_Minterrupts mip;
    t_Minterrupts mie;
    logic [63:0] mcycle;
    .
    .
    .

    logic [63:0] mconfigptr;

    t_Sedeleg sedeleg;

    t_Sinterrupts sideleg;
```

Instruction model

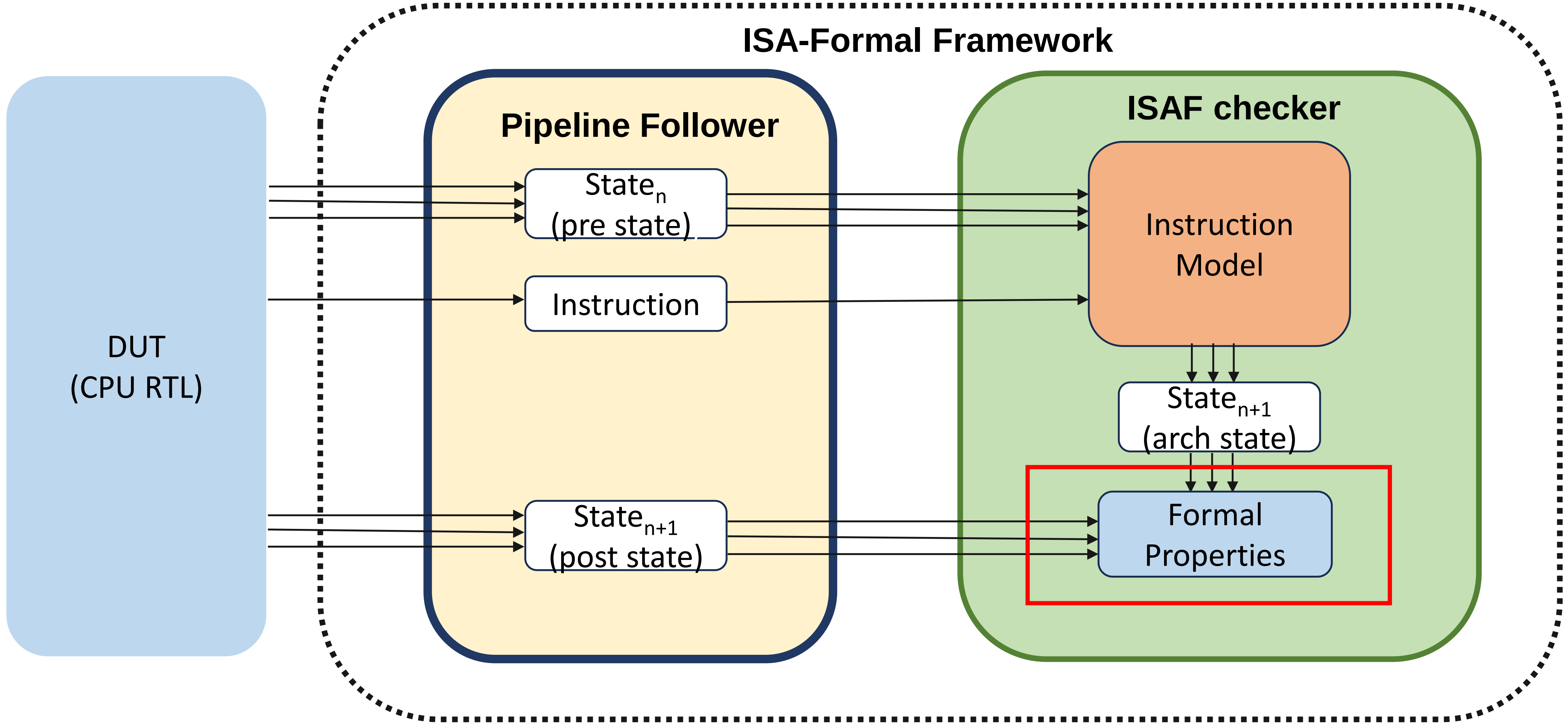
After the code generation, we can activate the ISA model by

1. Input an instruction opcode.
2. Decode instruction to AST.
3. Execute the AST semantic body.



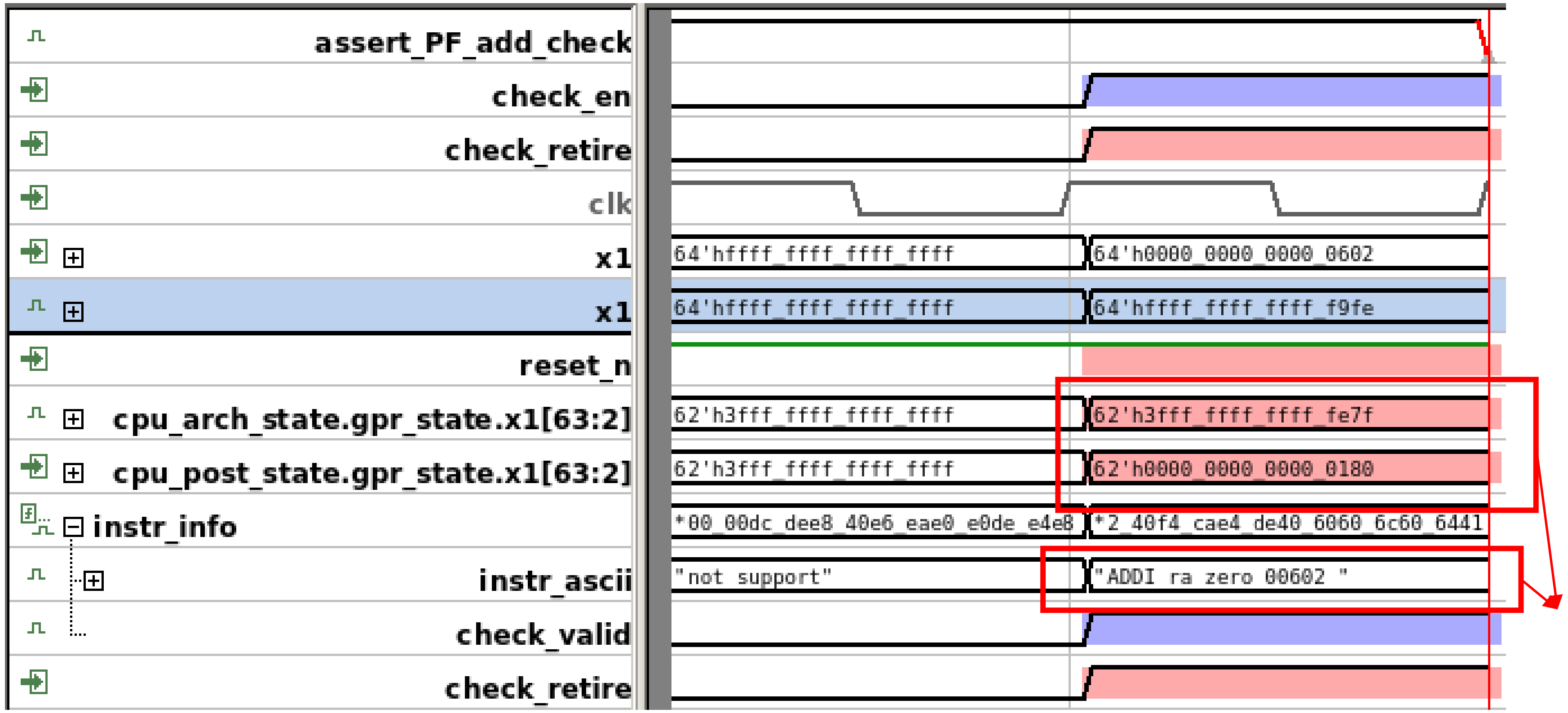
```
always_comb begin
   // Input instruction
1. if (|instruction) begin
    // Initialize the Instruction with pre_state.
    set_cpu_state(cpu_state);
    // Decode and execute instruction by sail-riscv function.
2.  inst_ast = ext_decode(instruction);
3.  result = execute(inst_ast);
    // Get n+1 state(Arch state)
    arch_cpu_state = get_cpu_state(cpu_state);
  end
end
```

ISA-Formal Overview



Formal properties

- ALU instruction
  - Instruction successfully retired
    - post_state.GPRs == architectural state.GPRs



```
assert_PF_add_check
check_en
check_retire
clk
x1                                 64'hffff_ffff_ffff_ffff            64'h0000_0000_0000_0602
x1                                 64'hffff_ffff_ffff_ffff            64'hffff_ffff_ffff_f9fe
reset_n
cpu_arch_state.gpr_state.x1[63:2]  62'h3fff_ffff_ffff_ffff            62'h3fff_ffff_ffff_fe7f
cpu_post_state.gpr_state.x1[63:2]  62'h3fff_ffff_ffff_ffff            62'h0000_0000_0000_0180
instr_info                         *00_00dc_dee8_40e6_eae0_e0de_e4e8  *2_40f4_cae4_de40_6060_6c60_6441
  instr_ascii                      "not support"                      "ADDI ra zero 00602 "
  check_valid
check_retire
```

ADDI x1, x0, ‘h00602
X1[63:2] should be ‘h0180

Formal properties

- Load/Store instruction
  - Memory system abstraction
    - Oracle address
    - Oracle data
  - Instruction successfully retired and access oracle memory
    - post state == architectural state
    - memory data == oracle data
- Constrain Async Interrupt event not happen.
  - E.g. assume mip[meip] == 0
- Complexity reduction
  - Black-box MMU, DCache, ICache, BTB

Conclusion

- Check instruction functionality:
  - mathematical prove the core conform to the RISC-V ISA Spec
  - Identifies design errors early in the development process, reducing the cost and effort required to fix them later.
- Check uArch functionality:
  - In CPU pipeline, instruction done based on forwarding(speculative state)
  - In Formal model, instruction done based on register file.(committed arch state)
  - Effective at finding bugs in the datapath, pipeline control, forwarding/stall logic involving complex sequences of instructions.

Q&A

- Contact
  - Stanley, likey714@andestech.com
  - SJ, shewu677@andestech.com
  - Yung-Ching, ychsiao@andestech.com