UVM testbench design for ISA functional verification of a microprocessor

Gabriel Wang, MediaTek (Beijing) Inc.

Hongtao Ma, MediaTek (Beijing) Inc.

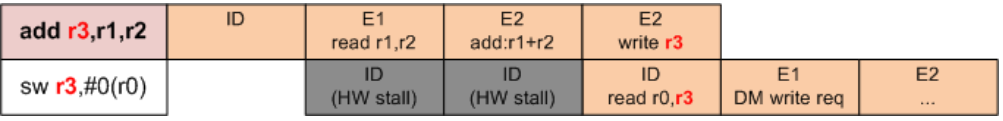
Maoduo Sun, MediaTek (Beijing) Inc.

Verify “add rt,ra,rb”

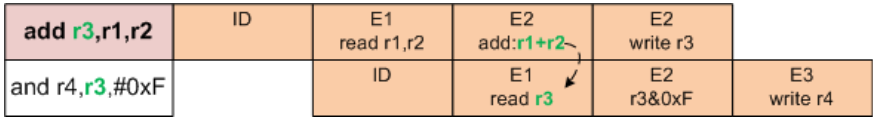
- **Functional correctness**
  - **Check: rt=ra+rb**
- **Operand completeness**
  - **Coverage**
    - **rt: R0~R15**
    - **ra/rb: R0~R15**
- **Instruction combination**
  - **Instruction sequence**

Random instruction generation

HW stall

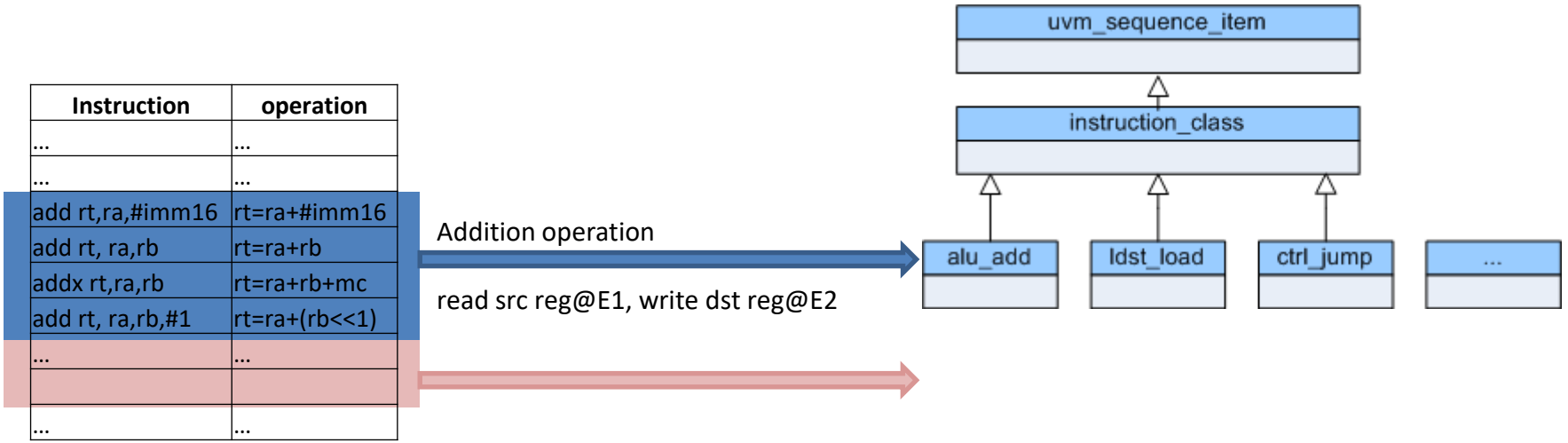


forwarding



Abstraction and categorization

- **Abstraction**
  - **Instruction → UVM sequence item**
  - **Multiple instructions → UVM sequence**
- **Categorization**
  - **Mapping 300+ instructions into 40 UVM classes**



Make random instructions predictable

- **Execute-then-generate**
  - **"blind randomization": register value is not referenced**

PC	instruction	operation
…	…	…
8	sub **r1**,r2,r8	**r1=r2-r8**
12	or r3,r4,r5	r3 = r4 | r8
16	lw r13,#0(r1)	r13 = DM[**r1**]
20	j **r1**	jump to pc=**r1**
…	…	…

if r1 value too large, e.g. 0xFFFF
May cause to access illegal memory region

if r1 value too small, e.g.12
May cause infinite backward jump

  - **"execute-then-generate"**
    - **All instructions are first "executed" during random instruction generation**

PC	instruction	operation
…	…	…
8	sub **r1**,r2,r8	**r1=r2-r8**
12	or r3,r4,r5	r3 = r4 | r8
16	lw r13,#0(**r2**)	r13 = DM[**r2**]
20	**sw r1**, #4(r3)	DM[r3+4]=r1
…	…	…

Each instruction "execute" and update register/DM value to mirror

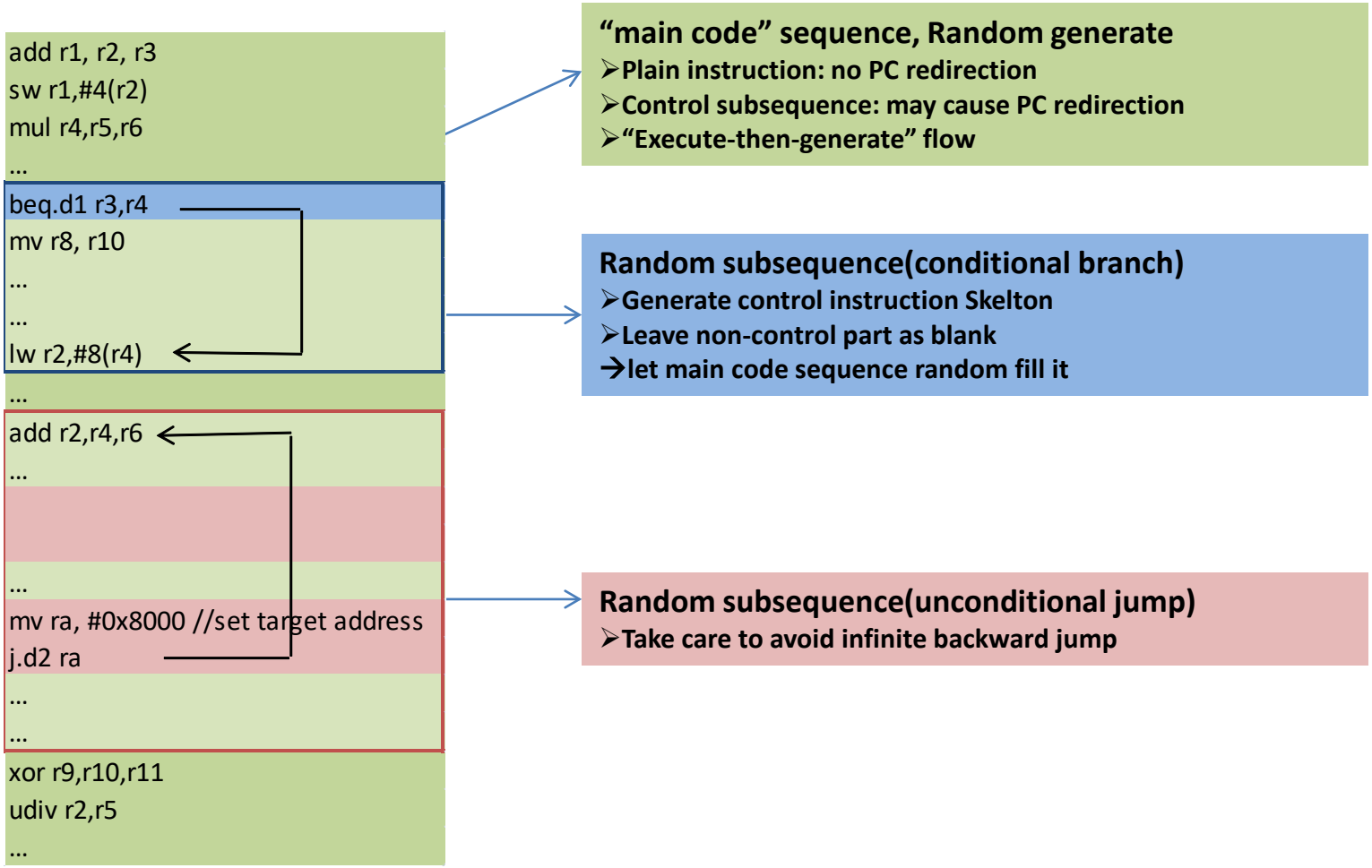
register mirror

DM mirror

Knowing all register/DM value, testbench can avoid randomizing unintended instructions

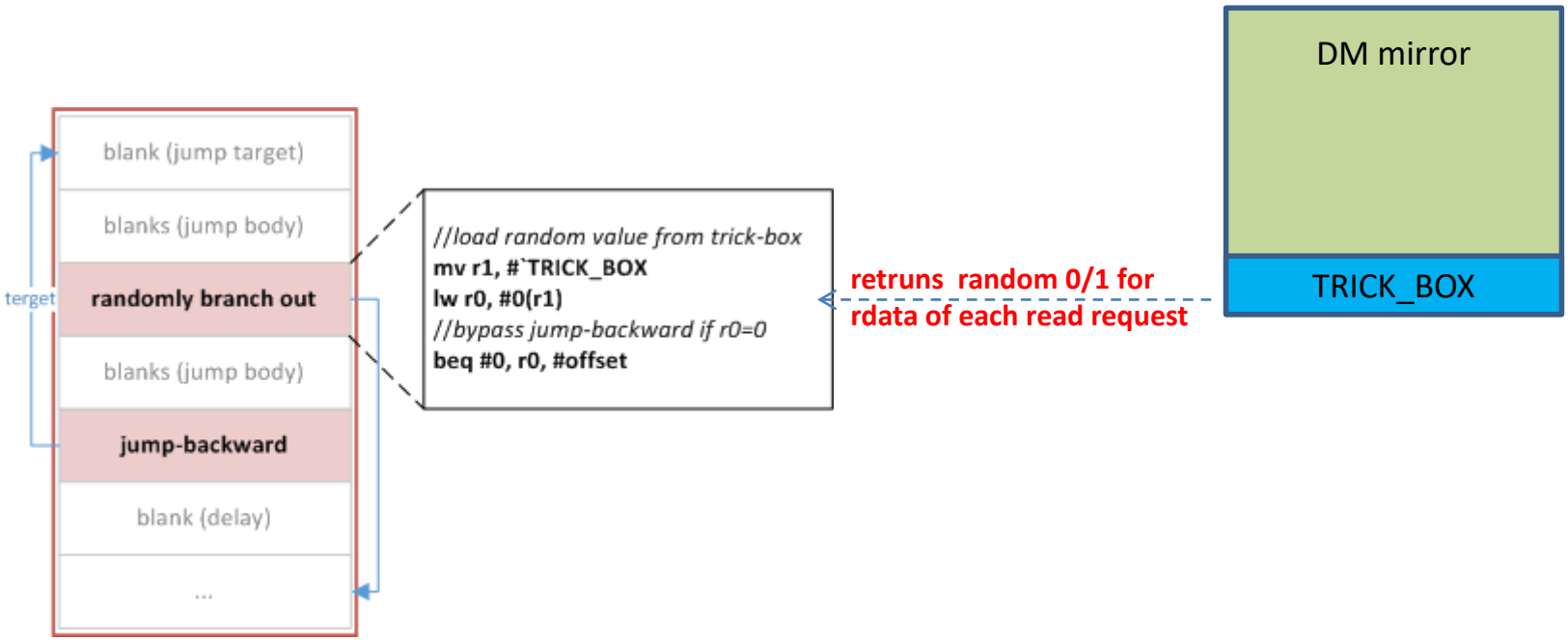
Layered sequence

- “main code” sequence and control sub-sequence



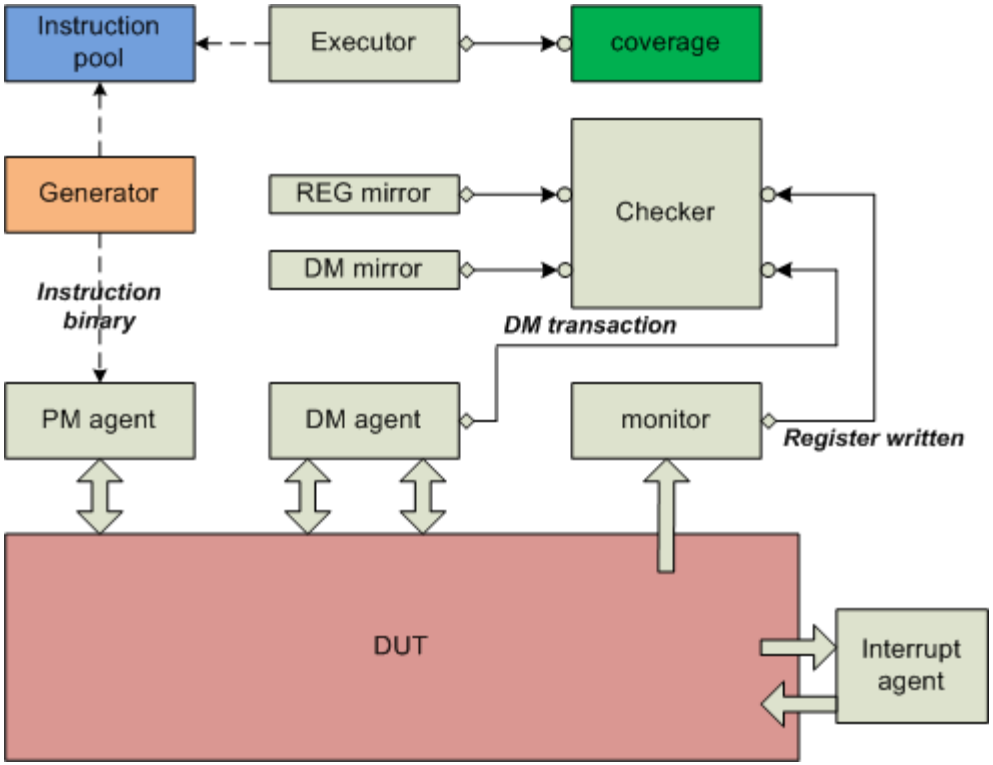
Avoid infinite backward jump

- **Control sub-sequence example**
  - **Backdoor manipulation of certain reserved DM address**
    - **→no infinite jump: jump instruction will be randomly skipped**



Bench structure

- **Functional correctness**



Instruction constraint

- **Intra-instruction constraint**
  - **Operand/value constraint**
  - **implemented in instruction class**

```
// Code Example A1: alu_add class implementation
class alu_add extends instruction_base
  rand reg_e rt, ra, rb;
  rand bit signed [15:0] imm16;
  constraint constr_instruction {
    instruction inside {ADD_RRI,   // add rt,ra,#imm16
                        ADD_RRR // add rt, ra, rb
                        }
    if (inst_contr != null) { !(instruction  inside {inst_contr.instruction_exclude});}
  }
  constraint constr_operand {
    rt inside {[R0:R15]};  ra inside {[R0:R7]};  rb inside {[R0:R7]};
    if (inst_contr != null) { !(rt  inside {inst_contr.w_reg_exclude});
      !(ra inside {inst_contr.r_reg_exclude});
      !(rb inside {inst_contr.r_reg_exclude});
    }
}
endclass
```

- **Inter-instruction constraint**
  - **Dynamically introduced by context**

PC	instruction
0	add r2,r4,r6
4	…
8	…
12	mv r8, #0x8000 //set target address
16	…
20	j.d2 r8
24	…
28	…

To protect target address: PC=16 shall not write r8

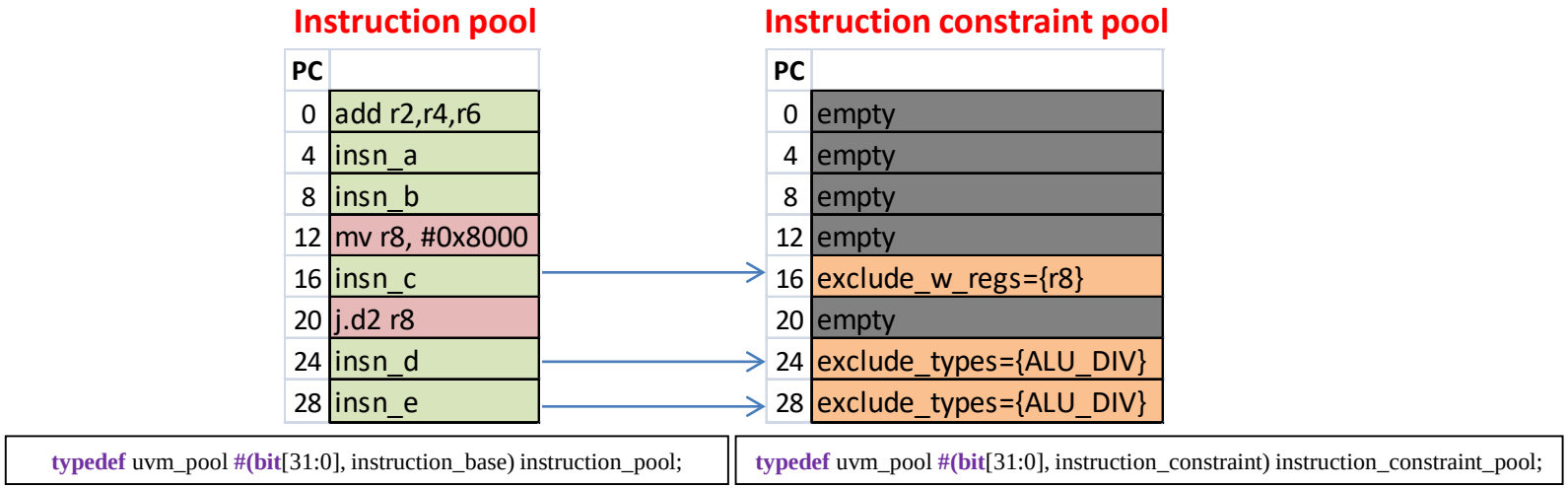
Delay slot limitation: division instruction should not be present in PC=24,28

Inter-instruction constraint

- **Modeled as an "instruction constraint" for each PC**
  - **Dynamically updated by instruction sequence**
  - **Constraint modeled as**
    - **exclude types, instructions, w_regs, …**
  - **Put into a instruction constraint pool**

```
// Code Example C1: instruction constraint class
class instruction_constraint extends uvm_object;
  bit [31:0]  pc;
  instruction_type_e exclude_types[$];
  instruction_e      exclude_inst[$];
  reg_e              exclude_r_regs[$];
  reg_e              exclude_w_regs[$];

endclass
```



Introducing data dependency

- **soft constraint in adjust_read_operand()**
  - **"TRY-TO" but not a MUST**

PC	Instruction	Read register	Write register
0	insn_a	R1	R2
4	insn_b	R3	R4
8	insn_c	R9	R10
16	"lw rt, #0(ra)"	ra	rt

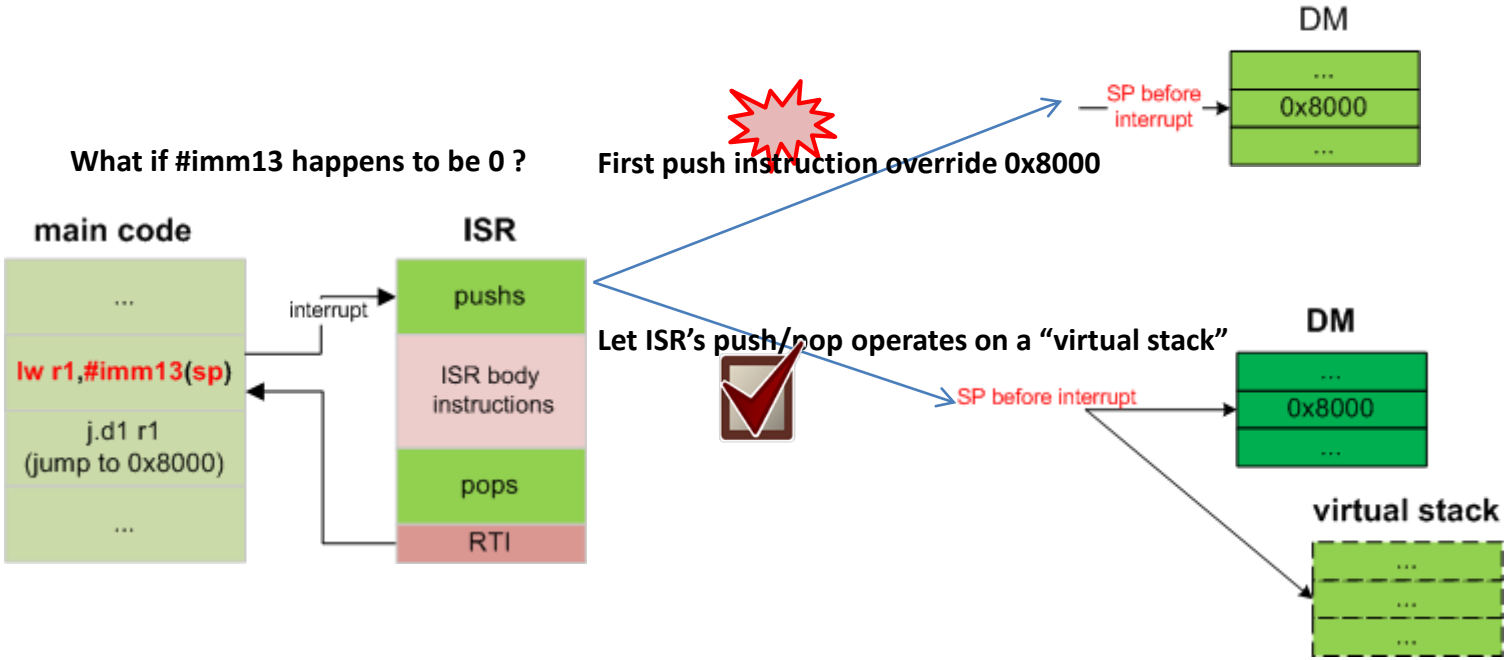
To randomly induce RAW data dependency:
Let ra inside {R2,R4,R10}

```
class ldst_ld extends instruction_base;
reg_e r_reg_candidate[$];
…
virtual function void adjust_read_operand();
  if (instruction == LW_RR) begin
    if  (!(ra inside {r_reg_candidate})) begin
      std::randomize(ra) with {
                                  ra inside {[R0:R7]};          //intrinsic hard constraint
                                  soft ra inside {r_reg_candidate}; //additional soft constraint
                                  };
    end
  end
endfunction
…
endclass
```

Intrinsic constraint may conflict with data dependency intension!
→soft constraint

Handling with random interrupt

- ISR instructions modeled as isr sub sequence
  - The only sequence not be executed during instruction generation
- push+pop provides basic protection of GPR
- How to protect DM?
  - To makes “execute-then-generate” flow effective



Result check

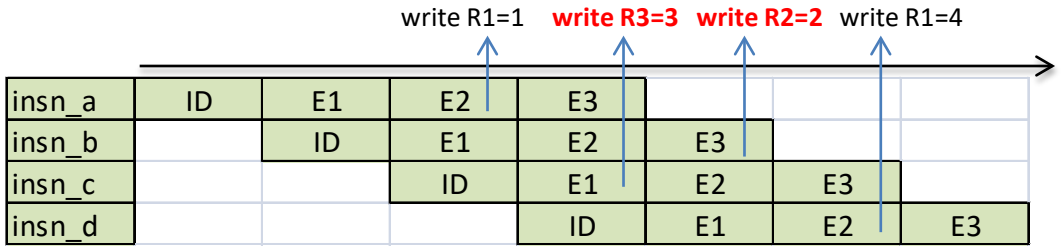
- **Timing gap between reference and DUT**

Reference model

PC	
0	insn_a
4	insn_b
8	insn_c
12	insn_d

PC=0: write R1=1
PC=4: write R2=2
PC=8: write R3=3
PC=12: write R1=4

DUT



- **Compare in a multi-stream way**