Metric Driven Microcode Verification: Navigating Microcode Coverage Complexities

Seungyeon Yu, Damin Son, Tony Gladvin George, Kihyun Park, Dongkun An, Wooseong Cheong, ByungChul Yoo

Samsung Electronics

Agenda

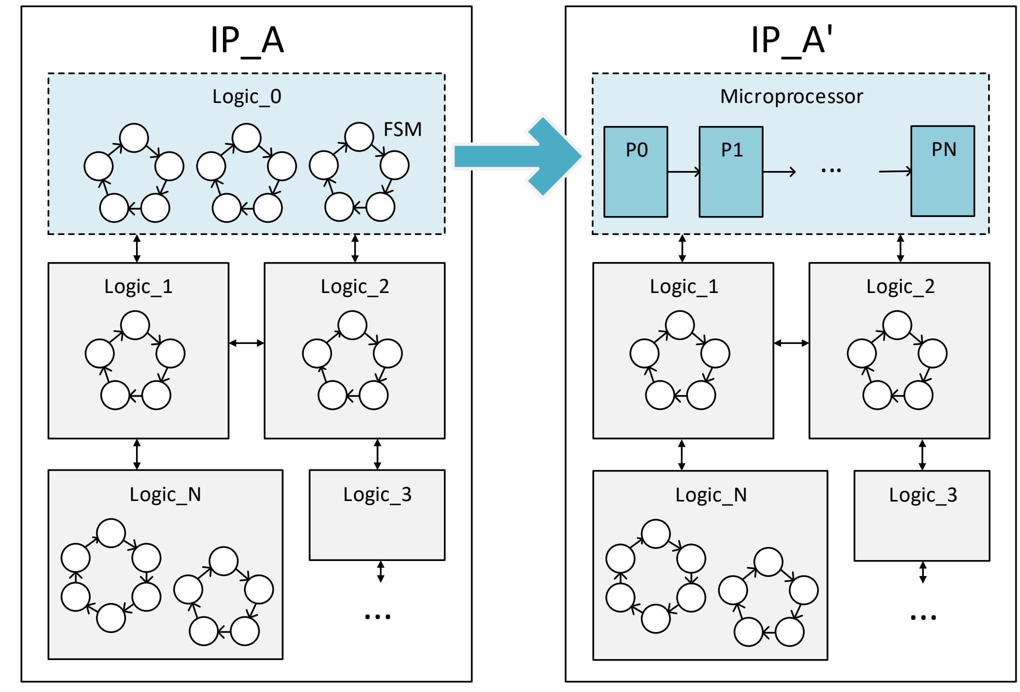
- Motivation
- Problem Statement
- Assertion-based Coverage Checker For Microcode
- Automated Checker File Generation
- Experimental Results
- Conclusion

Agenda

- Motivation
- Problem Statement
- Assertion-based Coverage Checker For Microcode
- Automated Checker File Generation
- Experimental Results
- Conclusion

Motivation

- Microprocessor applications to an in-house project
- Changes in architecture
- Expansion of verification scope
  - FSM Logic → FSM Logic + Microcode



Motivation

- The absence of indicators for the completion of microcode verification
  - Unpredictable whether all microcode is covered.
  - Possibility of bugs in microcode that has not been covered yet.

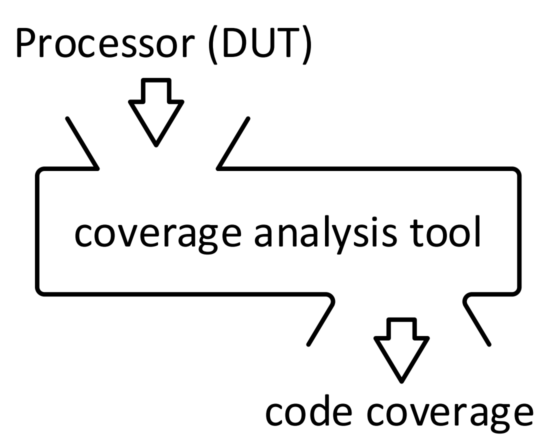
- The problem repeats whenever microcode is changed.
- So We propose a methodology to analyze the coverage of microcode.

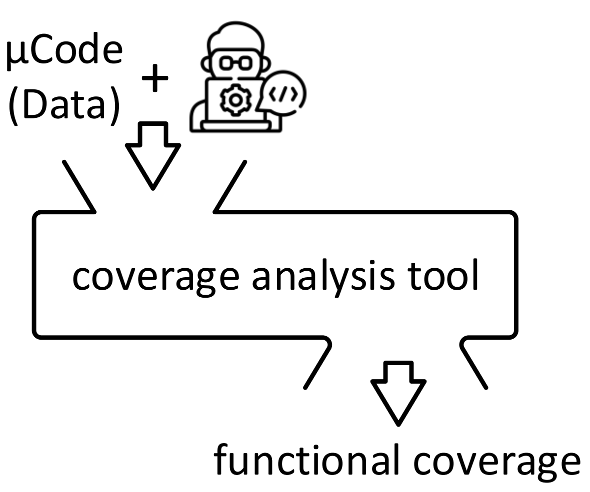
Agenda

- Motivation
- Problem Statement
- Assertion-based Coverage Checker For Microcode
- Automated Checker File Generation
- Experimental Results
- Conclusion

How to analysis the coverage of microcode?

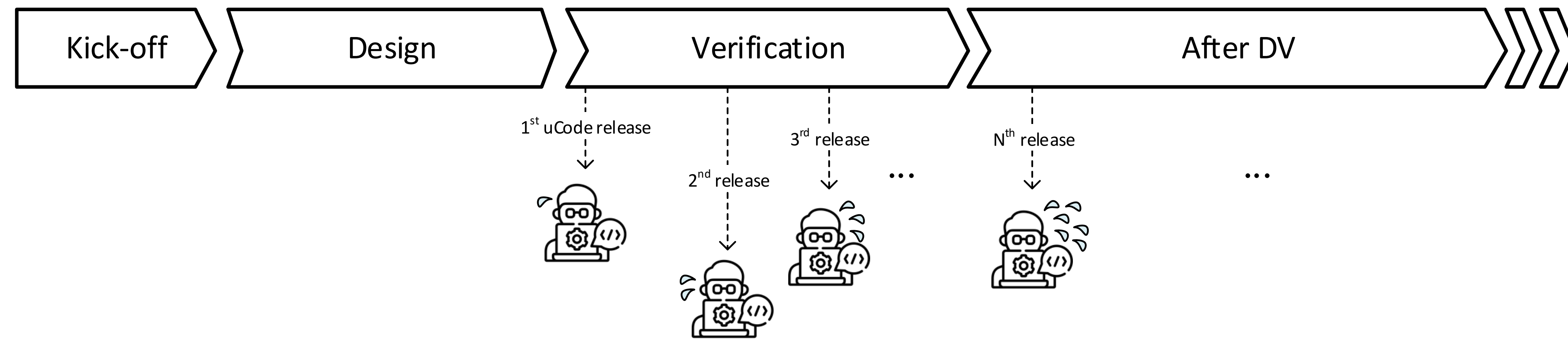
- The coverage analysis tool only supports coverage analysis results for microprocessors(DUT), not microcode(Data).
- For coverage analysis of microcode, verification engineers must write functional coverage manually.





An Increase in workload

- Whenever microcode is updated, verification engineers are required to manually modify the functional coverage.
- This approach increases the workload of verification, which affects the project schedule.

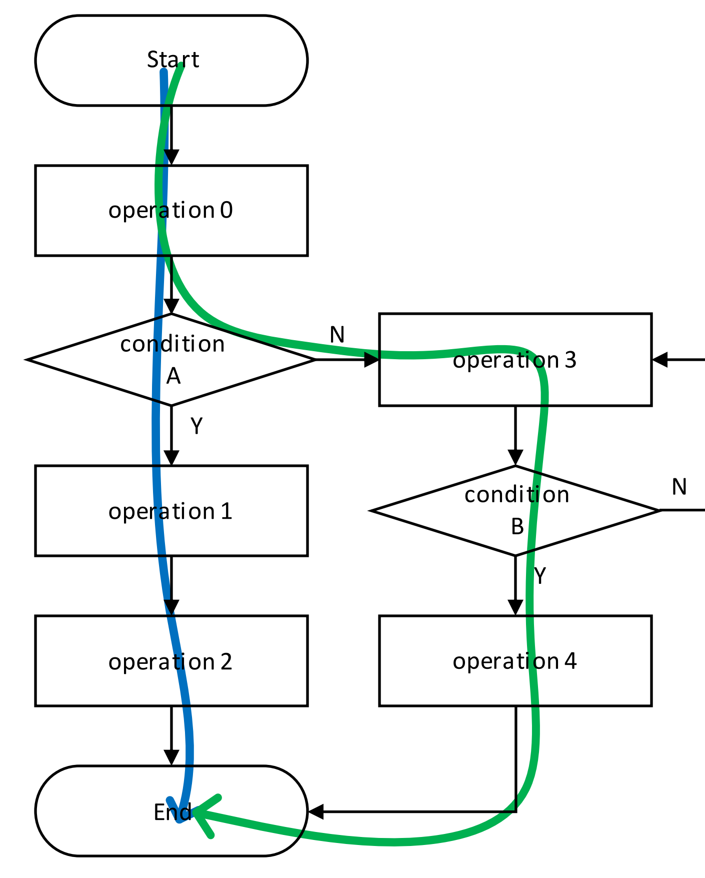


Agenda

- Motivation
- Problem Statement
- Assertion-based Coverage Checker For Microcode
- Automated Checker File Generation
- Experimental Results
- Conclusion

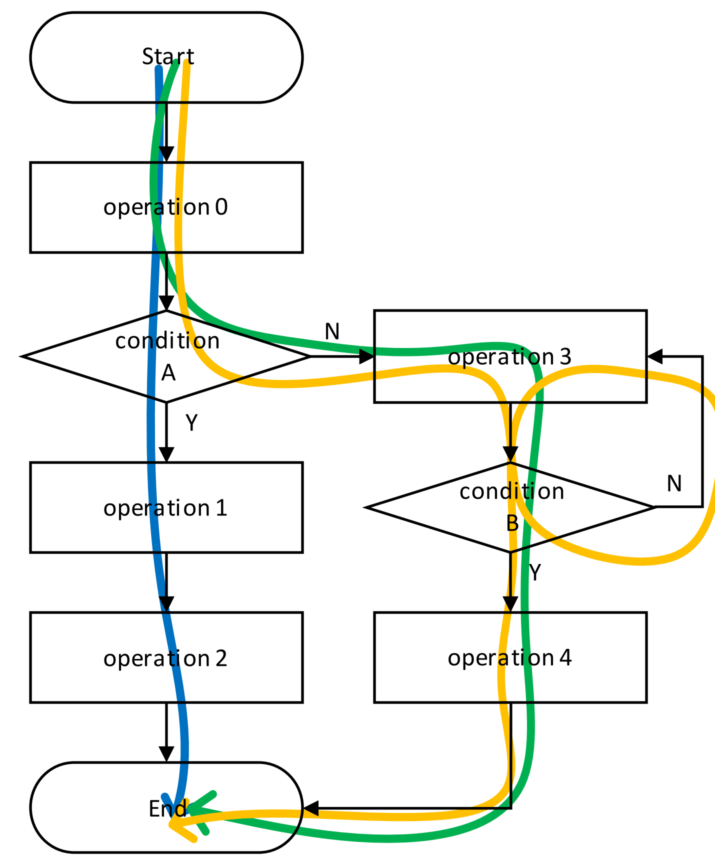
FC Checker: Operating concept

- Assumption
  - What if the criterion for achieving coverage 100% is to perform all operations?
- The operation flow implemented in microcode
  - blue path
    - op0 → (condition_A == T) → op1 → op2
  - green path
    - op0 → (condition_A == F) → op3 → (condition_B == T) → op4



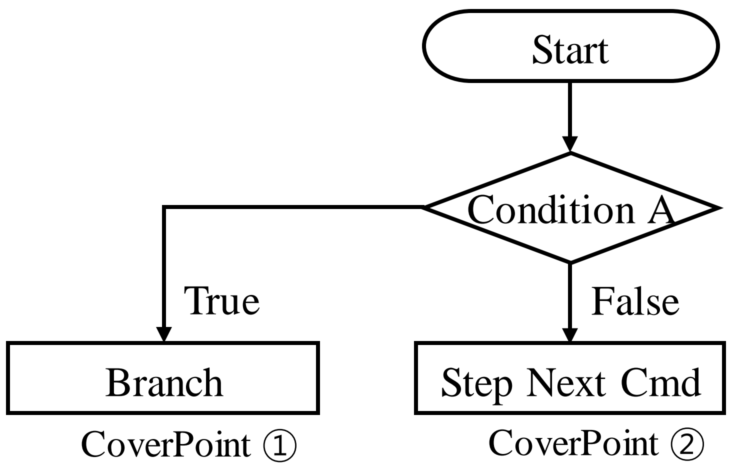
FC Checker: Operating concept

- If only the blue path and the green path are executed, it satisfies 100%.
- However, there is an additional path that needs to be covered.
    - yellow path
        - op0 → (condition_A == F) → op3 → (condition_B == T) → op4
- Therefore, the important part is where the flow changes according to the condition.



FC Checker: Coverpoints of the Checker

- Implement a checker by monitoring the path of the microcode address (PC).
- Check these two cases to generate functional coverage for microcode.
  - Cover Point 1 : Condition A == TRUE → Branch
  - Cover Point 2 : Condition A == FALSE → Step Next Command



FC Checker: Coverpoints of the Checker

- Check these two cases to generate functional coverage for microcode.
  - Cover Point 1 : Condition A == TRUE → Branch
  - Cover Point 2 : Condition A == FALSE → Step Next Command

<Microcode Example>

```
00020178 <SRC_LABEL0> :
    20178 :  c0 00 20 17        cmp r2, r12                  <------ Condition A
    2017c :  7e 80 a0 32        bge *+1f8 <DST_LABEL0>       <------ Branch
    20180 :  00 00 00 00        nop                          <------ Branch delay slot
    20184 :  fc 00 f0 2c        lsr  r12, r15, $0xf
…
000201f8 <DST_LABEL0> :
    201f8 :  1b 00 c0 2e        add r11, r12, $0x1           <------ CoverPoint ①
…
```

FC Checker: Coverpoints of the Checker

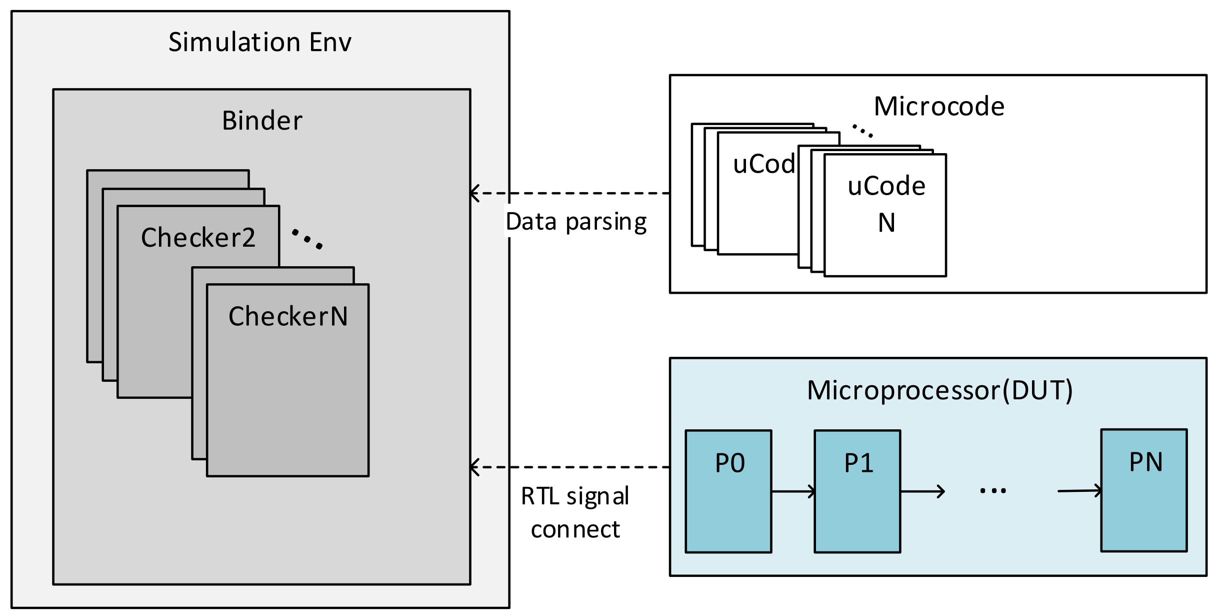
- Check these two cases to generate functional coverage for microcode.
  - Cover Point 1 : Condition A == TRUE → Branch
  - Cover Point 2 : Condition A == FALSE → Step Next Command

<Microcode Example>

```
00020178 <SRC_LABEL0> :
    20178 :  c0 00 20 17        cmp r2, r12                  <---- Condition A
    2017c :  7e 80 a0 32        bge *+1f8 <DST_LABEL0>       <---- Branch
    20180 :  00 00 00 00        nop                          <---- Branch delay slot
    20184 :  fc 00 f0 2c        lsr  r12, r15, $0xf          <---- CoverPoint ②
…
000201f8 <DST_LABEL0> :
    201f8 :  1b 00 c0 2e        add  r11, r12, $0x1
…
```

FC Binder : A file that binds checkers

- Binder file contains SV bind commands which bind multiple checker instances into the simulation environment.



- uCode
  - uCode address (PC) with br instruction
  - uCode address (PC) after br instruction
  - Microcode operation label
- uProcessor
  - RTL signal drive

Implementation of FC checker

```
module uCodeFcChecker #(parameter SRC_PC='h0, parameter DST_PC='h0) (
    input                  i_clk          ,
    input                  i_rstn         ,
    input                  i_inst_mem_csn ,
    input                  i_inst_mem_wen ,
    input [15:0]           i_inst_mem_a
);

wire inst_mem_rd = !i_inst_mem_csn & i_inst_mem_wen;

property ucode_step_next(src_pc);
    disable iff (!i_rstn)      inst_mem_rd & (i_inst_mem_a == src_pc  ) |->
                           ##1 inst_mem_rd & (i_inst_mem_a == src_pc+1) |->
                           ##1 inst_mem_rd & (i_inst_mem_a == src_pc+2);
endproperty

property ucode_branch(src_pc, dst_pc);
    disable iff (!i_rstn)      inst_mem_rd & (i_inst_mem_a == src_pc  ) |->
                           ##1 inst_mem_rd & (i_inst_mem_a == src_pc+1) |->
                           ##1 inst_mem_rd & (i_inst_mem_a == dst_pc  );
endproperty

_COV_UCODE_STEP_NEXT : cover property ( ucode_step_next(SRC_PC) );
_COV_UCODE_BRANCH    : cover property ( ucode_branch(SRC_PC, DST_PC) );

endmodule
```

- Coverpoints
  - ucode_step_next
    - Case not branched due to unsatisfied condition
  - ucode_branch
    - Case branched due to satisfied condition
- Monitoring the memory signals where uCode is loaded.

Implementation of FC Binder

```
   bind `XX_TOP UcodeBinder z_uCode_Checker();

   module UcodeBinder();

       uCodeFcChecker #(.SRC_PC('h807f), .DST_PC('h808e))
u_CHKR0_SRC_LABEL0__DST_LABEL0 (
           .i_clk                ( i_clk        ),
           .i_rstn               ( i_rstn       ),
           .i_inst_mem_csn       ( o_inst_mem_csn ),
           .i_inst_mem_wen       ( o_inst_mem_wen ),
           .i_inst_mem_a         ( o_inst_mem_a   )
       );


       uCodeFcChecker #(.SRC_PC('h8082), .DST_PC('h8087))
u_CHKR1_SRC_LABEL1__DST_LABEL1 (
           .i_clk                ( i_clk        ),
           .i_rstn               ( i_rstn       ),
           .i_inst_mem_csn       ( o_inst_mem_csn ),
           .i_inst_mem_wen       ( o_inst_mem_wen ),
           .i_inst_mem_a         ( o_inst_mem_a   )
       );

       …

   endmodule
```

- The number of checker instances is generated as many as the number of branch instructions in the microcode.
- The binder provides the address information required for the FC checker instances as a parameter.
- The binder connects the checker instances and the RTL signal.

Agenda

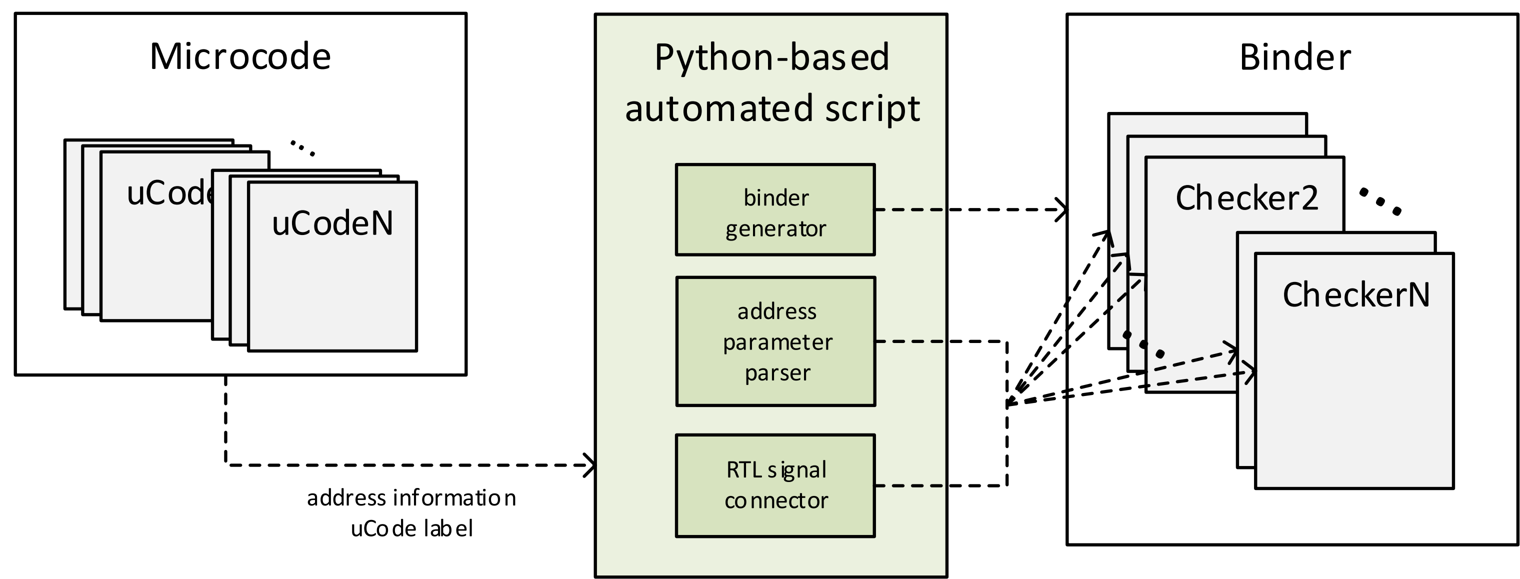
- Motivation
- Problem Statement
- Assertion-based Coverage Checker For Microcode
- Automated Checker File Generation
- Experimental Results
- Conclusion

Automated Checker File Generation

- The high flexibility of microcode leads to frequent modifications, making it difficult for verification engineers to respond every time.
- The driving of inputs and parsing of uCode information particularly increases the workload of verification engineers.
- Therefore, we propose a method of automatically creating an assertion coverage by introducing a Python-based automated script.

Automated Checker File Generation

- When an uCode file is inputted, the python script generates a checker file and a binder file containing uCode information and RTL connections.

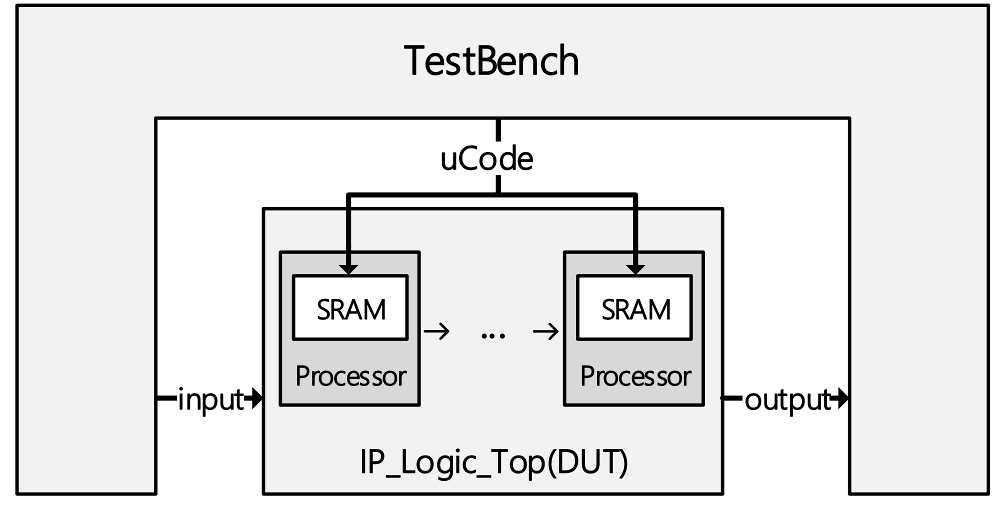


Agenda

- Motivation
- Problem Statement
- Assertion-based Coverage Checker For Microcode
- Automated Checker File Generation
- Experimental Results
- Conclusion

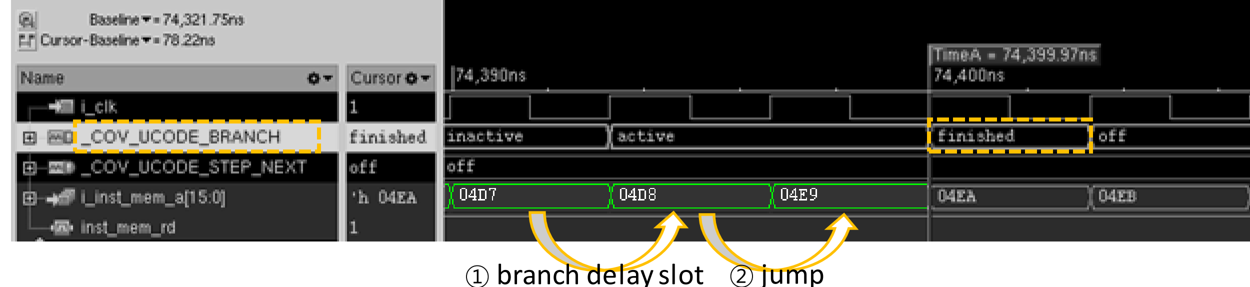
Experimental Condition

- Custom microprocessors in the IP
  - Processor has a branch delay slot.
  - It is based on RISC-V.
  - DUT takes h/w and uCode into consideration together.
- The verification test suite contains many different combinations of external stimuli.

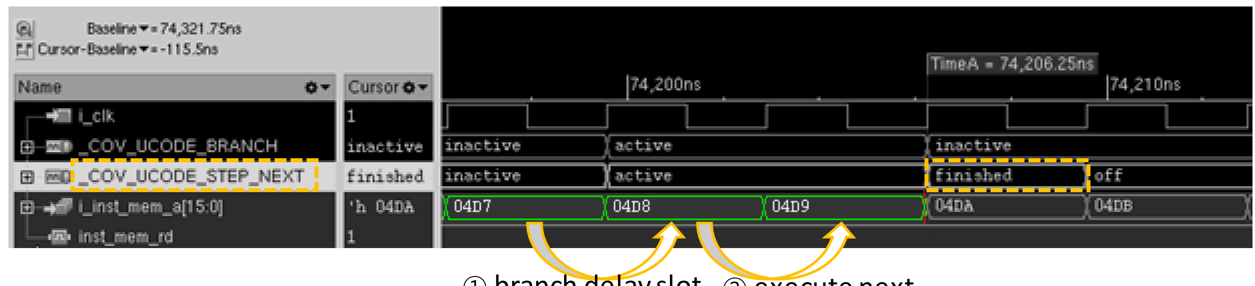


Experimental Results

- Branch



- Execute next instruction



Experimental Results

- The code coverage of the processor does not indicate whether the IP operation flow has been performed.
- Since it is coverage for the DUT that operates the ucode, it cannot be an indicator for uCode verification.
- Through the ucode checker, we can confirm that **98%** of FC has been achieved.
- The solution is reusable for all IPs that use the processor and has been applied to several projects in practice.

Type Name	Overall Average Grade	Overall Covered
Ucode0Binder	94.44%	34 / 36 (94.44%)
Ucode1Binder	100%	58 / 58 (100%)
Ucode2Binder	100%	26 / 26 (100%)
Ucode3Binder	97.56%	80 / 82 (97.56%)

Project	Coverbin (Covered bin / Total Cover bin)		Code Coverage (%)	Functional Coverage (%)
	Processor	Microcode	Processor	Microcode
IP A	673/911	198/202	73.8	98.0
IP B	852/1143	389/422	74.5	92.2
IP C	849/1143	431/460	74.3	93.7

Agenda

- Motivation
- Problem Statement
- Assertion-based Coverage Checker For Microcode
- Automated Checker File Generation
- Experimental Results
- **Conclusion**

Conclusion

- Universally applicable and easily reused
- Serve as a criterion for signing off on microcode verification
- Significantly reduce the time and effort required for microcode verification
- Improve the quality and reliability of microcode

Questions

- ssyeon.yu@samsung.com