Metric Driven Microcode Verification: Navigating Microcode Coverage Complexities

Seungyeon Yu*, Damin Son*, Tony Gladvin George*, Kihyun Park*, Dongkun An*, Wooseong Cheong*, ByungChul Yoo*
*Memory Division, Samsung Electronics
Email: (ssyeon.yu, dm.son, tony.gg, kihyun0.park, dongkun.an, ws.cheong, byung.yoo)@samsung.com

***Abstract*- As memory products need higher performance, there are large portions of functions that benefit from microcode-centric design than traditional logic-based designs. In response to these changes, coverage analysis is essential to verify the completeness of microcode verification. As the coverage analysis tool does not furnish coverage results for microcode, we propose a new method to ascertain the completeness of the microcode using an assertion-based coverage checker. This is a method that determines coverage by monitoring changes in the program counter (PC) before and after the operation of the branch instruction within the microcode. The derived coverage results enable the identification of insufficient scenarios, allowing verification engineers to reinforce the scenarios to achieve high coverage. By applying the proposed method to actual work, it was possible to detect an additional 361 hidden coverbins on average. In addition, it demonstrated an improvement rate of approximately 20.4% compared to the code coverage of the microprocessor alone, which can be potentially useful for our future research and development. Hence, the proposed method can serve as a reliable metric for assessing the completeness of microcode verification, while also serving as a tool to improve the verification completeness.**

I. INTRODUCTION

The growth in technological development has led to an unprecedented demand for high-speed and high-performance designs. The full-automation architecture was introduced by offloading the CPU workload to H/W to satisfy the advanced required performance, but it was difficult to avoid operational problems and scale-up performance due to high H/W dependency. In order to overcome the existing limitations, a microcode-based programmable processor was applied. Through this, it is possible to respond by modifying the microcode even if it does not work as intended due to hardware errors. In addition, in order to improve performance, the processor was placed in the form of a chain to enable multi-parallel processing of the command.

As the microcode-based general-purpose platform structure is applied, it is inevitable to verify the microcode operation as well as the existing Finite State Machine (FSM)-based RTL logic. In addition, in order to ensure the completeness of microcode verification, it is necessary to check the coverage and reinforce the insufficient verification scenario. Historically, microcode verification processes have been inadequate, lacking the capability to ensure completeness. With the constant evolution of design structures and methodologies, the need for design verification has grown. This paper explains a method that significantly improves the verification process by addressing the gaps in microcode verification.

This paper proposes a newly applied microcode-based IP simulation verification plan. The composition of this paper is as follows. First, Section II introduces the background knowledge of simulation verification and coverage to understand the proposed technology and the existing verification flow. And Section IV and V propose a coverage verification method to enhance the completeness of microcode verification and a Python automation script to increase versatility and flexibility. Section VI analyzes the simulation output obtained by applying the proposed plan for real-work. Finally, Section VI will describe the conclusions and future works.

II. BACKGROUND

A. Code Coverage

Code Coverage (CC) is an indicator that numerically expresses the degree to which the verification scenarios have tested the design. Through this, we can identify the functions or conditions that have not been performed in the logic. Code coverage is expressed as a percentage, and can be used an indicator of the completeness of the design and verification.

Figure 1 shows the flow of the verification methodology through code coverage measurement.

1) First, it is necessary to develop a test scenario to verify the function of the verification target (RTL). It is necessary to determine how to set the input value for function verification and the initial configuration required in RTL.

2) The functions that can be turned on/off and must be verified are set to selective option. If we need to set it randomly within a specific range or need to set it randomly on/off every time, apply it as a random option.

3) Regression is performed by changing the seed to a random value so that random operation can be performed every time for the scenario with options.

4) Regression should be performed to merge all simulation results to check the code coverage, that is, whether each simulation has performed all RTL functions.

5) Use these results to check if all the actual functions or conditions have been performed and to check if reinforcement is needed in the first test scenario.

The basic principle of verification is to check that all functions of the design have been performed. To that end, it is necessary to continuously check the completeness of verification and design by measuring code coverage. It analyzes whether all parts of the design have been performed, reinforces the test scenario, and repeats the above flow.

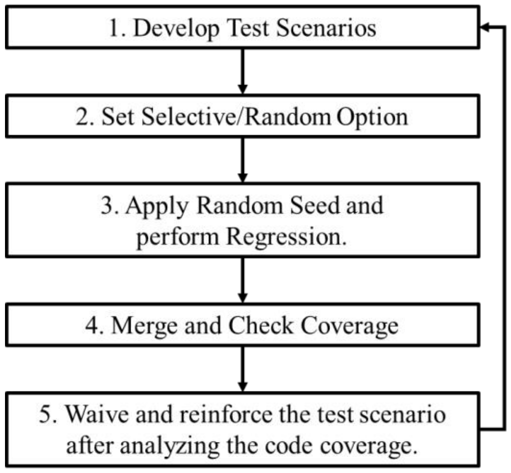

1. Develop Test Scenarios
2. Set Selective/Random Option
3. Apply Random Seed and perform Regression.
4. Merge and Check Coverage
5. Waive and reinforce the test scenario after analyzing the code coverage.

Figure 1. Verification Flow with Code Coverage

B. *Microcode (uCode)*

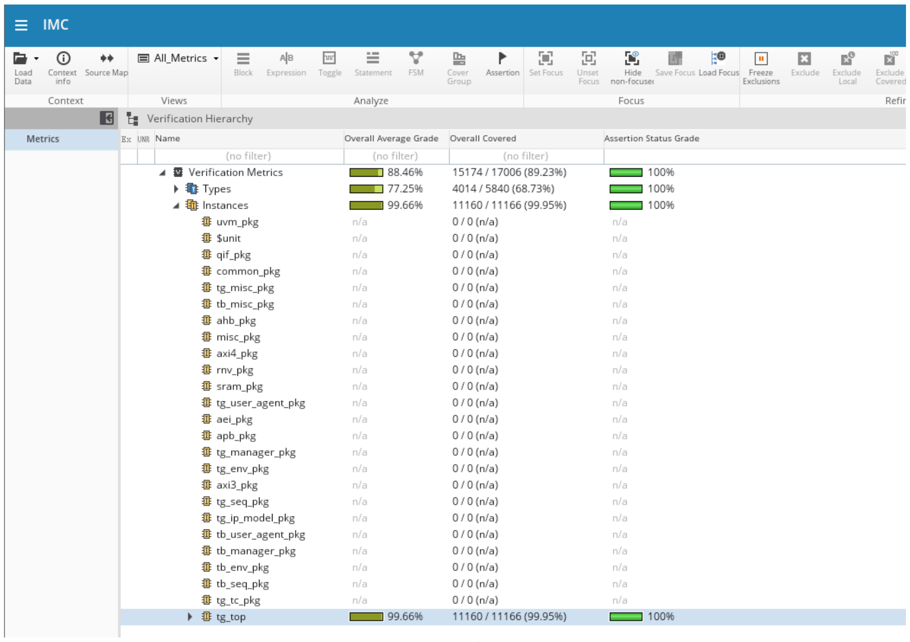
Microcode (uCode) is an instruction set that allows the intended operation to be performed inside a processor. In order to meet the demand of high speed and high performance design, a programmable processor was introduced. The processor operates in a 4-staged pipeline structure in the order of fetch, decode, execution, and write-back based on the commands stored in the internal instruction memory and performs the functions of data processing, data transfer, and branch. The processor is a custom one developed in-house, and has a branch delay slot. The branch delay slot is a feature that causes the instruction following the branch to also be executed when the branch instruction is operated.

C. *IMC (Integrated Metrics Center)*

IMC is a tool for analyzing coverage data collected while performing the code coverage flow described above. We can check how many functions were performed in the RTL during the actual simulation process. In Figure 2, code coverage can be checked using the overall covered value. The value of 11160/11166 (99.95%) can be interpreted as meaning that the operation of 11160 out of 11166 codes to be performed in RTL was confirmed. And this value is regarded as code coverage.

Through this value, we can check whether the actual function or condition is performed in RTL logic. It is possible to improve the completeness of verification and design by repeatedly analyzing this to reinforce and check the test scenario.

However, this verification is limited to RTL logic-based design, so it does not guarantee the completeness of microcode-based design verification installed in RTL in binary form. Therefore, the need to introduce functional coverage (FC) for microcode has emerged in order to understand the actual performance of the designed function, going beyond the code coverage-based verification that was checked through the existing code execution level.



Ex	UNR	Name	Overall Average Grade	Overall Covered	Assertion Status Grade
		(no filter)	(no filter)	(no filter)	
		Verification Metrics	88.46%	15174 / 17006 (89.23%)	100%
		Types	77.25%	4014 / 5840 (68.73%)	100%
		Instances	99.66%	11160 / 11166 (99.95%)	100%
		uvm_pkg	n/a	0 / 0 (n/a)	n/a
		$unit	n/a	0 / 0 (n/a)	n/a
		qif_pkg	n/a	0 / 0 (n/a)	n/a
		common_pkg	n/a	0 / 0 (n/a)	n/a
		tg_misc_pkg	n/a	0 / 0 (n/a)	n/a
		tb_misc_pkg	n/a	0 / 0 (n/a)	n/a
		ahb_pkg	n/a	0 / 0 (n/a)	n/a
		misc_pkg	n/a	0 / 0 (n/a)	n/a
		axi4_pkg	n/a	0 / 0 (n/a)	n/a
		rnv_pkg	n/a	0 / 0 (n/a)	n/a
		sram_pkg	n/a	0 / 0 (n/a)	n/a
		tg_user_agent_pkg	n/a	0 / 0 (n/a)	n/a
		aei_pkg	n/a	0 / 0 (n/a)	n/a
		apb_pkg	n/a	0 / 0 (n/a)	n/a
		tg_manager_pkg	n/a	0 / 0 (n/a)	n/a
		tg_env_pkg	n/a	0 / 0 (n/a)	n/a
		axi3_pkg	n/a	0 / 0 (n/a)	n/a
		tg_seq_pkg	n/a	0 / 0 (n/a)	n/a
		tg_ip_model_pkg	n/a	0 / 0 (n/a)	n/a
		tb_user_agent_pkg	n/a	0 / 0 (n/a)	n/a
		tb_manager_pkg	n/a	0 / 0 (n/a)	n/a
		tb_env_pkg	n/a	0 / 0 (n/a)	n/a
		tb_seq_pkg	n/a	0 / 0 (n/a)	n/a
		tg_tc_pkg	n/a	0 / 0 (n/a)	n/a
		tg_top	99.66%	11160 / 11166 (99.95%)	100%

Figure 2. IMC (Integrated Metrics Center)

III. RELATED WORK

Research papers focusing on the verification of microprocessors have been published for approximately 20 years. The predominant themes revolve around exploring methods for verifying ISA (Instruction Set Architecture) operations, pipeline functionality, and pipeline hazards through formal verification techniques or property-based tests. These studies primarily emphasize validating the functionality of the microprocessor itself or verifying the execution of a single ISA defined within the processor.

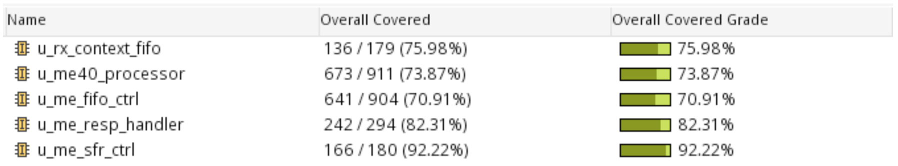
We would like to mention a paper that studied how to grasp the accuracy of operation when microcode is loaded on the processor[5]. Implementing an operation previously implemented in hardware as a microcode program cannot secure accuracy with the existing verification method. As a method for this, there is a study conducted by CENTTECH TECHNOLGIES. In order to perform a program written in microcode on actual hardware, it must be compiled into a binary file and programmed into memory inside the processor. The processor then proceeds with the operation according to the sequence of loaded programs, and the method introduced here is to determine the accuracy of the operation using a microcode translator. The microcode translator identifies the operation flow of commands stored in memory and shows the actual memory location of the commands. The approach involves modeling the end state by determining the program counter (PC) value, machine state, registers, and the starting state of a specific sequence, and then verifying whether it matches the actual behavior.

However, in practical industrial scenarios where IP verification is crucial, the focus should extend beyond verifying individual instructions to validating unit operations composed of a sequence of instructions. In other words, it becomes imperative to verify not just the microprocessor but also the microcode itself.

Hence, our proposal revolves around introducing a methodology for analyzing microcode coverage, essential for addressing unforeseeable bugs caused by unpredictable scenarios. Random testing becomes indispensable for detecting bugs arising from unpredictable scenarios. Since these tests do not directly implement the scenarios deemed necessary by verification engineers, there is a need for coverage metrics to assess how thoroughly the functionality has been tested.

IV. PROBLEM STATEMENT

Microcode, which functions as a layer between machine code and hardware, plays a critical role in the operation of programmable HW engines. However, the traditional verification methods for microcode pose challenges. The main issue lies in the inability to confirm if all aspects of the microcode have undergone complete verification since the code coverage analysis tools cannot capture its coverage. In Figure 3, u_me40_processor shows the coverage of the processor. The assessment of a processor's code coverage reveals the degree to which it has processed all ISAs. However, this evaluation solely confirms the processor's accuracy for each opcode and not the complete microcode coverage. Hence, it is not possible to ensure microcode coverage through processor code coverage.



Name	Overall Covered	Overall Covered Grade
u_rx_context_fifo	136 / 179 (75.98%)	75.98%
u_me40_processor	673 / 911 (73.87%)	73.87%
u_me_fifo_ctrl	641 / 904 (70.91%)	70.91%
u_me_resp_handler	242 / 294 (82.31%)	82.31%
u_me_sfr_ctrl	166 / 180 (92.22%)	92.22%

Figure 3. Code Coverage (Processor)

As a way to check whether all of the designed microcode has been executed, it may be considered to check whether the processor accesses all of the internal memory addresses in which the code is stored. However, this is not a way to ensure that all microcode has been performed. In order for microcode to be executed in the processor, the process of fetch → decode → execution → write-back must be performed. When the microcode is actually performed, it is the execution step, and when the processor accesses the code of the internal memory, it is the fetch step. This means that even the fetched code may not actually be executed. Therefore, it is necessary not only to check whether the code is accessed or not, but also to check whether the code is actually executed.

Another problem is that its inherent flexibility leads to frequent updates in microcode, unlike the RTL code that freezes after a specific development stage. This fluidity in microcode requires verification engineers to often update their assertions, further complicating the verification process. This can lead to incomplete or inefficient verification, resulting in potential operational risks once mass-produced.

V. Proposed Method

To ensure the completeness of design and verification, we devise a strategy for creating functional coverage that checks the operations of the branch instruction within the microcode. The microcode Functional Coverage (FC) checker was developed with a focus on the operations of the branch instruction, which is a part of the microcode instruction set. Branch instructions are divided into conditional branch, which executes operations based on certain conditions, and unconditional branch, which executes immediate branching. Conditional branch instructions first execute a comparison (cmp) operation, and the execution of the branch operation is determined based on the results of this comparison. Using the conditional functionality of the branch instructions allows us to confirm that all operations are being executed.

As the internal operations of the programmable HW engine alter the Program Counter (PC), continuous monitoring of the actual PC value becomes essential during simulation. To determine the occurrence of a branch, one must extract the Source PC (SRC_PC), the Destination PC (DST_PC), and the Next PC(NEXT_PC). SRC_PC is the address where the branch instruction is located within the microcode, DST_PC is the address after the branch instruction is executed when the condition is satisfied, and NEXT_PC is the address after the branch instruction is executed when the condition is not satisfied.

As shown in Figure 4, if Condition_A is satisfied, the state will transition to 'Branch', performing the branching operation. If not satisfied, the state will move to 'Step_Next', executing the microcode of the next PC. The 'Branch' state signifies the case in which the actual PC value transitions from SRC_PC to DST_PC, while the 'Step_Next' state represents the case where the actual PC value shifts from SRC_PC to NEXT_PC. By monitoring the changes in the PC within the processor, it is possible to determine the occurrence of a branch as described earlier. This provides a criterion for determining whether all operation flows have been executed according to the selective option.

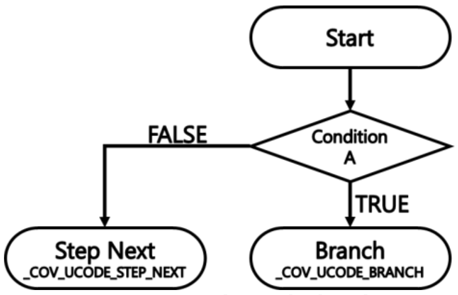


Figure 4. Microcode FC Checker Flow

There is a significant reason for verifying the execution of branch operations without relying on the code coverage method that maps the address information and operation status of the microcode in a one-to-one correspondence. While it is important to verify whether a specific operation has been executed, it is equally critical to verify the sequence of other operations conducted before and after. Considering the resources of microcode SRAM, it is necessary to use a common code in cases where there are redundant operations within each flow. If the microcode is structured in this manner, a specific operation executed in a single flow would be recognized as having been touched. However, the actual expected outcome is to confirm whether the operation has been executed across all flows. Consequently, drawing from the operation of the branch instruction within the microcode, the microcode FC checker was implemented.

VI. IMPLEMENTATION

It is suggested how the automated script was written to increase the usability of this method. Frequent modifications are made depending on the characteristics of microcode with high flexibility, and it is difficult for the verifier to respond every time. In addition, there is a need for a method that can be used generically in various projects other than specific projects. Therefore, we propose a method of automatically creating an assertion by introducing a Python-based automated script.

As shown in Figure 5, when a file with a microcode and an address where the code will be stored is used as an input of an automated script, a binder and a checker for functional coverage are generated as outputs. For microcode that is frequently changed, the checker is required for all branch points, and all checkers generated in this way need to be connected to the RTL signal one by one, so I would like to introduce an automated script for this.

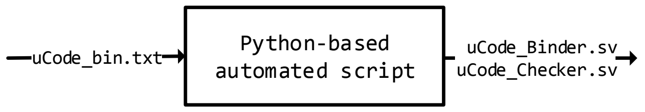


Figure 5. Microcode FC generation flow using Python-based automated script

A. Microcode (uCode) Analysis

An example of a microcode file used as an automated script input argument for generating functional coverage is shown in Figure 6. This code is an excerpt of a part of the actual microcode. Understanding of microcode is essential for FC creation, so I would like to explain it through examples.

‘20178’ on the left side of the first line represents 0x20178(hex) and refers to the address value where the microcode is located. We can check the format in which the microcode described at the bottom is grouped into labels, and the label name is <SRC_LABEL0>. The command ‘cmp r2, r12’ in address 0x20178 compares the values contained in r2 and r12. At this time, r2 and r12 mean registers and contain the immediate value or operation result value required for the microcode operation. The index value after ‘r’ is a number used to distinguish multiple registers.

The command of address 0x2017c is an operation to determine whether to branch by referring to the result of comparing r2 and r12. The ‘bge’ used in the line is a command that branches when the previous register is greater than or equal to the later register. Therefore, if r2 is greater than or equal to r12 (r2 >= r12), it jumps to the command of label <DST_LABEL0> corresponding to the address described at the back, '0x201f8'.

The microcode flow varies depending on the operation of the ‘cmp’ and ‘bge’ commands, and the main part of the functional coverage proposed in this paper is to check whether all possible cases have been touched. If r2 >= r12 is satisfied as a result of 'cmp', the microcode address will be changed in the same order as (0x2017c -> 0x20180 -> 0x201f8 -> ...) and if r2 < r12 is satisfied (0x2017c -> 0x20180 -> 0x20184 -> ...). The ‘nop’ instruction at 0x20180 is always executed, independent of the compare condition. Due to the branch delay slot characteristic inherent in the processor, the address does not immediately change. This characteristic influences the composition of SVA property statements, a topic that will be discussed in the following chapter.

```
00020178 <SRC_LABEL0> :
   20178: c0 00 20 17       cmp r2, r12
   2017c: 7e 80 a0 32       bge *+0x201f8 <DST_LABEL0>
   20180: 00 00 00 00       nop
   20184: fc 00 f0 2c       lsr r12,r15,$0xf
…

000201f8 <DST_LABEL0>:
   201f8: 1b 00 c0 2e       add r11,r12,$0x1
…
```

Figure 6. Input file(Microcode) Example

B. Microcode FC Checker : Tool implementation

Figure 7 depicts a checker based on System Verilog Assertion (SVA). SVA is a verification methodology that checks whether a signal has an expected value at a specific timing by directly referencing a particular RTL signal. The RTL signal required for the checker is received as a module input, and the expected behavior is described in the form of a property. The "cover" keyword is then utilized to verify if the condition has been satisfied at least once. The cover property syntax created checks repeatedly whether the condition is met for each clk posedge (rising edge: 0 ➔ 1) during run-time. When the specified case appears, it is marked as covered and turned off.

The microcode FC checker shown in Figure 7 receives SRAM interface signals such as i_inst_mem_csn, i_inst_mem_wen, and i_inst_mem_a as inputs from the module declaration unit. The i_inst_mem_csn and i_inst_mem_wen interfaces are signals that can check write or read enable, and i_inst_mem_a is an SRAM address that is accessed when the enable is satisfied. The reason for checking the SRAM signal to measure the microcode coverage is as follows.

The microcode is described using assembly commands, and the converted binary data must be written to the instruction SRAM in the design when initializing the verification environment. During simulation run-time, if control signals such as address, csn, and wen of the instruction SRAM are included as input values, the desired output data can be obtained and microcode operation can be performed. Therefore, to determine whether a specific microcode operation has been performed, it is necessary to check whether the corresponding SRAM address has been read. In other words, it can be seen that the microcode FC checker in Figure 7 used the SRAM signal for the property describing the expected operation.

Before taking a closer look at the explained property syntax, it would be helpful to understand the operators used in the syntax.

- disable iff (*) : Syntactic disable if * is TRUE
- ##N : Condition check after N-delay
- A |-> B : If A is true, check the condition of B in the same clk.

To further elaborate, the next step is to examine the two properties of the checker, 'ucode_step_next' and 'ucode_branch' shown in Figure 7.

The 'ucode_step_next' property refers to the case where the microcode executes the instruction at the next address without jumping to it when a conditional branch instruction fails to satisfy the specified condition. In the checker, this property passes when the SRAM address changes in the order of (SRC_PC -> SRC_PC+1 -> SRC_PC+2). The inclusion of 'SRC_PC+1' in the address path is attributed to the branch delay slot. During the instruction SRAM read, if the SRAM address continuously increases from SRC_PC for 3 cycles, the microcode at the next address will be executed even if the branch condition is not satisfied. This is because the microcode has failed to satisfy the branch condition and has instead executed the next address's microcode. SRC_PC is a parameter that can be received when the checker instance is created, and it represents the SRAM address where the conditional branch instruction is located.

The 'ucode_branch' property covers the scenario where a conditional branch instruction satisfies the condition. The property passes when the SRAM address changes in the order of (SRC_PC -> SRC_PC+1 -> DST_PC). Similarly, the characteristic of the branch delay slot was also incorporated. During the instruction SRAM read, if the SRAM address increases by 1 and then changes to DST_PC over 3 cycles, the microcode will jump to the pre-defined address after satisfying the branch condition. It should be noted that DST_PC is an input parameter that represents the SRAM address after the branch operation takes place. Due to the 1-cycle delay caused by the SRAM output data fetch, the microcode cannot jump directly from SRC_PC to DST_PC.

It is possible to check whether the scenario has been performed when the previously described property syntax is written after the cover and simulation is turned through the simulation or IMC tool.

```
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
);

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
```

Figure 7. Microcode FC(Function Coverage) Checker

C. Microcode FC Binder : Integration into the workflow

One if the outputs of the FC generation script is the ‘binder file’, UcodeBinder. It contains SV bind commands which bind multiple checker instances into the simulation environment. The binder file creates each checker instance and connects RTL signals required as checker input. A checker instance is mapped to a branch operation in the microcode file, thus a checker instance is created for each branch instruction. The source and destination addresses of the branch command are parsed and the value is driven to the parameter.

Figure 8 shows an example of a binder file. It is divided into the part that assigns the RTL signal required for the checker and the part that creates the checker instance, and the number and configuration of the instance may increase depending on the microcode configuration.

When creating a checker instance, it was named using the labels for the address before the branch and the label after the branch in the microcode. In order to check whether all the scenarios expected by the created test bench are satisfied, the option is applied to all possible ranges or randomized to perform simulation several times. When checking the result by merging the coverage extracted for multiple simulations performed, it is possible to check whether all expected design operations have been performed in the described scenario.

```
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
```

Figure 8. Microcode Checker Binder

VII. Case Study

In an actual project setting, the method proposed was implemented under the subsequent conditions.

- Our actual IP incorporates 4 microprocessors, with the target being the microcode loaded onto the processor SRAM.
- 10 simulation options have been applied, each being a toggle-enabled option with possible values of [0, 1].
- Random seeds have been applied, with the total number of tests amounting to approximately 6,000. The random seed generates various stimulus combinations, leading the microcode to execute through different paths. The coverage outputs generated by multiple seeds can be merged, thereby providing a combined coverage of approximately 6000 for random scenarios.

The results of the coverage for a single branch instruction in the microcode, as confirmed through a simulation tool, are presented in Figures 9 and 10. The coverage analysis results can be individually scrutinized for each branch present within the microcode. Figure 9 illustrates a scenario in which the branch condition is unfulfilled. The observation indicates that when the 'i_inst_mem_a', identical to the microcode's PC, remains steady for three sequential cycles, the coverage clause concludes and deactivates. Conversely, Figure 10 delineates a case where the branch condition is satisfied and subsequently covered. This is demonstrated by the waveform when the 'i_inst_mem_a' transitions to a designated branch destination PC.

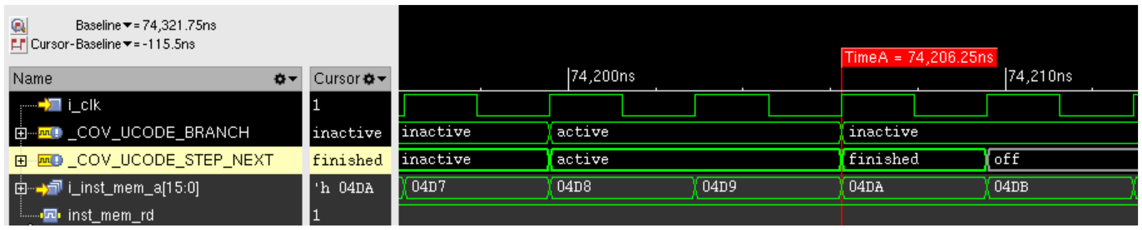


Figure 9. Simulation results for coverage (ucode_step_next)

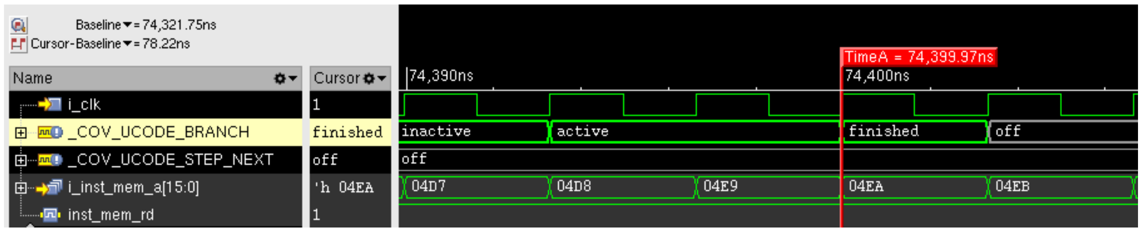
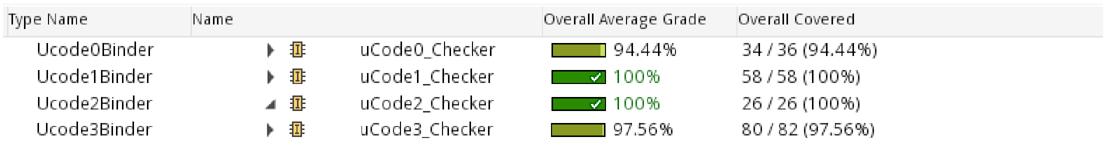


Figure 10. Simulation results for coverage (ucode_branch)

Figure 11 depicts the outcome of coverage verification conducted across the entire microcode utilized in a specific IP within a real-world project. The IP has 4 processors internally. Since each processor runs a unique microcode, a corresponding number of binder files must be generated, equal to the number of processors. As shown in Figure 11, the coverage results for the 4 uCodeCheckers can be verified.

Around 6,000 random seeds were employed to perform a regression test, establishing that all cases, except for four coverbins, were encompassed in the formulated scenario. Insights from the regression test enable the identification of untouched coverbins. This, in turn, provides validation engineers with the opportunity to enhance additional scenarios or initiate discussions with designers to undertake the waiver procedure.



Type Name	Name	Overall Average Grade	Overall Covered
Ucode0Binder	uCode0_Checker	94.44%	34 / 36 (94.44%)
Ucode1Binder	uCode1_Checker	✓ 100%	58 / 58 (100%)
Ucode2Binder	uCode2_Checker	✓ 100%	26 / 26 (100%)
Ucode3Binder	uCode3_Checker	97.56%	80 / 82 (97.56%)

Figure 11. Coverage Analysis results for Random seed tests

VIII. Experimental Result

Table I showcases the verification results for three projects, IP A, IP B, and IP C. The combined coverage results for the random seed test previously conducted correspond to IP A. With the introduction of Microcode Functional Coverage (Microcode FC):

IP A: Improved its microcode coverage from the initial 73.8% (673 out of 911) to an impressive 98% (198 out of 202), with an addition of 202 coverbins.

IP B: The coverage rate increased from 74.5% (852 out of 1143) to 92.2% (389 out of 422) by using 422 new coverbins.

IP C: Implemented the same methodology, boosting its coverage from 74.3% (849 out of 1143) to 93.7% (431 out of 460).

TABLE I.
COVERAGE RESULTS FROM THREE PROJECTS

Project	Coverbin *(Coverd bin / Total Cover bin)*		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

A. *Increased Coverage Insights*

Previously unverifiable microcode coverages can now be checked, ensuring a more comprehensive verification process. By merging the coverage results of tests approved with multiple selective options, it is possible to supplement scenarios that have not been covered, thereby enhancing the completeness of microcode verification.

B. *Improved Verification Completeness*

With the aid of automation scripts, the solution can be applied across various projects. It guarantees more comprehensive verification and adapt to changes as microcode designs evolve.

The increasing reliance on microcode usage indicates the need for microcode-based IP verification techniques. The method proposed in this paper, emphasizing microcode FC measurement automation, offers a way forward, ensuring the consistent quality of programmable hardware engines.

C. *Comparison with Existing Method*

The existing verification method uses code coverage to supplement the verification scenario to check whether all necessary functions have been performed. Verification is performed using this flow by replacing the coverage for the microcode with functional coverage. The completeness of the verification can be improved by supplementing the scenario for the untouched cover point by analyzing the functional coverage.

IX. CONCLUSION

This paper introduces a method for verifying the completeness of microcode coverage. To ensure that all scenarios in a test driving various selective options are covered, it is necessary to examine the actual flow of the microcode operation. The methodology we propose is a technique for composing functional coverage that verifies the flow of microcode operation. Through this, we can address the issue of not being able to guarantee completeness due to the lack of a standard for verifying the completion of microcode.

Furthermore, we have implemented a Python automation script that enables the proposed method to be applied to intellectual properties (IPs) operating through microcode across all projects. One prominent characteristic of microcode is its flexibility to changes at any stage during the project development process. However, this feature necessitates frequent adaptations from verification engineers to these changes, causing inconvenience. Through our automation script, the verification of results can be achieved without the need for additional code writing, which is anticipated to bolster operational efficiency.

In response to the escalating performance requirements of memory solution products, the complexity of their design is expected to inevitably increase. Consequently, the utilization of structures involving microcode is likely to expand, making alterations in the verification methods for microcode-based IPs unavoidable. We have initiated the automation of microcode FC measurement to secure the quality of SSD Controllers at an early stage. We propose the implementation of the method presented in this paper.

REFERENCES

[1] Cadence, Integrated Metrics Center User Guide Integrated Metrics Center User Guide 19.09.
[2] Accellera, SystemVerilog Language Reference Manual.
[3] IEEE, Standard for SystemVerilog Unified Hardware Design, Specification, and Verification Language (IEEE Std 1800™-2017).
[4] RISC-V, The RISC-V Instruction Set Manual Volume I: User-Level ISA.

[5] Davis, Jared, Anna Slobodova, and Sol Swords. "Microcode verification–another piece of the microprocessor verification puzzle." International Conference on Interactive Theorem Proving. Cham: Springer International Publishing, 2014.