An Efficient Random Instruction Sequence Generation for Verification of Domain Specific Architecture Processor

Wonjae Lee[1], Youngsub Ko[2], Yelim Han[3], Doowon Lee[4],
Dahyun Yoo[5], Heewon Ahn[6], Joongbaik Kim[7]

Samsung Electronics Co., Ltd.
([1]wonja2.lee; [2]youngs.ko; [3]yelim12.han; [4]doowon.lee; [5]dah.yoo; [6]hwon.ahn; [7]joongb.kim@samsung.com)
1-1, Samsungjeonja-ro, Hwaseong-si, Gyeonggi-do, Korea

***Abstract*- The growing demand for parallel computation in embedded systems has driven the rise of domain specific architectures (DSA). Among them, very long instruction word (VLIW) architectures offer high performance and efficiency. However, verifying these processors is a significant challenge due to the complexity of generating valid and meaningful instruction sequences. This paper proposes an efficient, constraint-aware random instruction sequence (RIS) generation technique tailored to domain-specific VLIW processors. By randomly selecting and assembling pre-defined instruction graphs under hardware constraints, our method achieves a valid test rate of nearly 100% regardless of the number of instruction bundles, surpassing the rate of a conventional RIS. Our approach also facilitates directed verification by the addition of specific instruction graphs. Consequently, the proposed generator improves verification efficiency, aligns tests with realistic usage scenarios, and offers controllable verification focused on DSAs.**

I. Introduction

With the increasing of the AI and multimedia processing in embedded systems, the demand for massively parallel computation has surged, yet general-purposed processor has encountered limitations in terms of performance and power efficiency. To address this challenge, domain specific architectures (DSAs), optimized for specific domains, have emerged as an alternative. The DSAs, such as NPUs, not only enhance performance but also should be able to provide the flexibility to adapt to new algorithms.

Among the architectures capable of meeting these demands for DSAs, very long instruction word (VLIW) is a notable approach that bundles multiple independent operations into a single long instruction to be executed concurrently. By delegating the complexity of instruction dependency analysis to the compiler rather than the hardware, it minimizes power consumption thus delivers a combination of high performance, energy efficiency, and programmable flexibility. The potential of such architecture has been demonstrated through commercial examples that handle AI inference [1] or accelerate image processing [2]. These examples illustrate that VLIW-based DSAs can meet the requirements of embedded systems while achieving extreme efficiency.

In microprocessor verification, random instruction sequence (RIS) generation is a common verification technique used to automatically produce diverse instruction combinations, uncovering bugs that are hard to detect manually. However, this conventional RIS generation faces fundamental challenges when applied to VLIW architectures, which execute multiple instructions in parallel. The main challenge lies in generating valid instruction sequences that comply the hardware's pipeline and resource constraints to avoid illegal combinations that could cause structural hazards. Therefore, effective DSA verification requires a sophisticated, architecture-aware constrained-random methodology.

In this paper, we propose an efficient RIS generation method suited for the verification of domain-specific VLIW processors. Fig. 1 compares RIS generation between the conventional and ours. Since VLIW processors utilize multiple instructions packed into a single long word under strict dependency and resource constraints, conventional random generation suffers from a low efficiency in producing valid instruction sequences. Moreover, these purely random sequences often fail to represent real-world workloads, limiting their effectiveness in verifying practical scenarios. To address these challenges, by leveraging the domain-specific characteristic of the target processor, we apply the complex constraints of the architecture to the RIS generation process, thereby efficiently producing practical and valid tests that better reflect realistic usage scenarios.

II. Related Work

Functional verification of a microprocessor is a challenging task that strive to find all the bugs in the microprocessor design. As an industry practice, random instruction tests are extensively used as they could uncover corner cases that

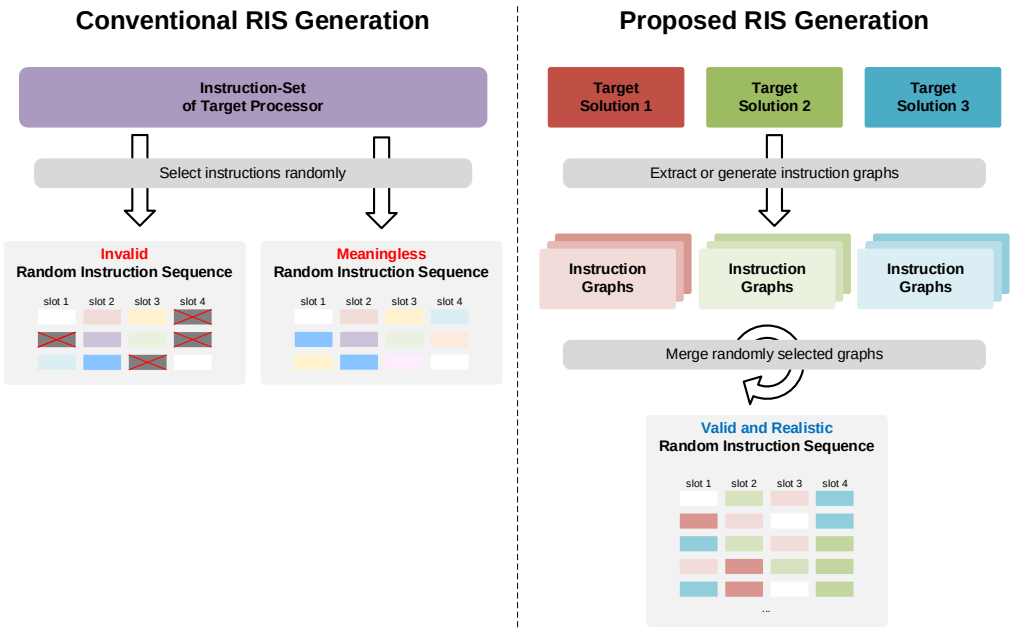

Figure 1. The concept of the proposed method

design and verification engineers have missed in the earlier verification stages. However, these may waste precious time and resource that are allowed for verification if they are generated in a purely random manner. That is because tests are often invalid (i.e., they do not abide by the processor's architecture specification) or less meaningful (i.e., they compose instruction sequences that are very unlikely to occur in the target applications).

To enhance efficiency, various test generation techniques have been proposed in the past decades. They tailor RIS tests so that they obey the specification and achieve a high coverage quickly, while still being able to trigger subtle bugs. In the literature, IBM's Genesys-Pro [3] is a well-known RIS test generator. Its two main components are the architecture description that models the specifics of the processor design under verification, and the architecture-independent knowledge for test generation engine. A test template delineates a sequence of instructions and various constraints that must be met in generated instructions. For the given test template, test cases are generated by using a constraint satisfaction problem (CSP) solver.

The RIS generation methods have been applied to VLIW processors (e.g., [4], [5], [6]). In the work presented in [4], Genesys-Pro is deployed to generate RIS for STMicroelectronics' processors. This work shows the test generator can generate valid VLIW instructions, but may incur significant effort in modeling the processor architecture and describing detailed constraints to avoid generating illegal instructions. Transmeta's DVGen [5] leverages a directed-random, minimally-constrained test generation. It can create a combined test by mixing multiple test specifications, each of which captures a unique test intent. Such mix-tests are shown effective in uncovering hard-to-find bugs due to concurrent occurrence of corner cases in different units of the design. In yet another approach [6], a framework was introduced that automatically generates tests by sequentially combining smaller pre-defined instruction sequence. This method is designed to create complex scenarios that validates the program counter unit's handling of parallel operations during control flow events.

EDA tools for designing application-specific instruction set processor (ASIP) (e.g., [7], [8]) offer RIS generators. While these generators can produce all possible instruction sequences, they often require non-negligible engineering effort to take into account the test intent.

III. Proposed Method

Considering the constraints described above, the efficient use of RIS for verifying a domain-specific VLIW processor must satisfy two requirements. First, the automatic random generation process must be able to produce a large number of valid tests, while satisfying the constraints of the instruction bundle. Second, the generated tests must be meaningful, i.e., they should reflect realistic scenarios. To meet these requirements, we focus on the domain-specific characteristic of the target processor. Here, "domain-specific" implies that the type of application solutions running on the processor is limited and known in advance. Moreover, a "domain" or "scenario" can be regarded as a massive instruction sequence composed of many sub-sequences of instructions. These sub-sequences are already executable on the target processor and are therefore assumed to be closer to real-world use cases than

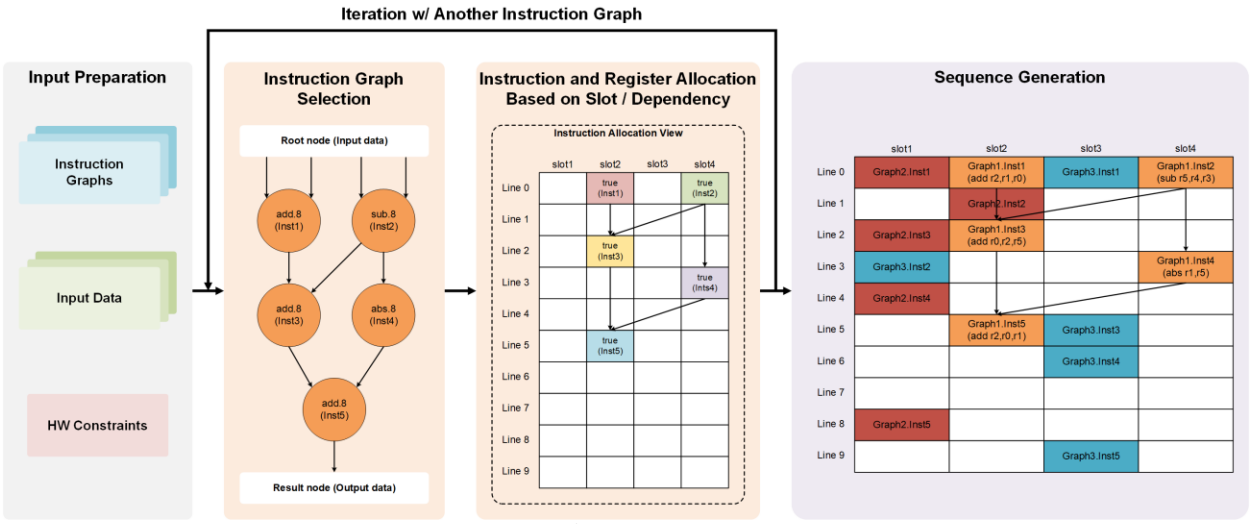

Figure 2. Steps of the proposed RIS generation

purely random sequences. By combining instruction sub-sequences derived from the target solution with controllable randomness, a large number of tests similar to the target solution can be efficiently generated. Such tests are particularly important for domain-specific processors that are designed with specialized architectures or instructions tailored to their target solutions.

Based on this assumption, we first define a large collection of instruction graphs (sub-sequences). Each graph represents the connectivity between compatible instructions, where the structure (e.g. the number and type of instructions) is flexibly configurable. These instruction graphs can be extracted from the binary of the target application or manually generated according to the verification objective. Fig. 2 shows the process of proposed RIS generation. As each instruction graph is itself a pre-verified test and the components of graphs are randomly placed within VLIW instruction-slot and sequence-slot, the probability of generating a valid test is significantly increased. Our compiler-level generator incorporates hardware constraints of the target processor, such as slot configuration, instruction latency, and register usage, by providing controllable random ranges. Consequently, our proposed method enables the efficient generation of random yet practical and valid test cases.

Moreover, by constructing pre-defined instruction graphs targeted to specific verification goals, we can easily supplement insufficient coverage and thus drive the verification process to the desired direction. Depending on the domain for which the processor targets, different sets of pre-defined instruction graphs can be used, or the number of graphs associated with particular instructions can be increased to strengthen verification of those instructions.

For the rest of this section, we illustrate each step of the proposed method by presenting its key concepts and examples.

A. Input Preparation

Fig. 3 shows the format (*json* file) of a sequence graph used in the proposed method, which consists of *nodes* and *edges*. Each *node* represents instruction information and it includes the node name, instruction type (e.g., add, sub, mul), and input and output operand counts. Each *edge* defines dependency between two instructions. Each node maintains a list of ingress edges, as shown in Fig. 3(b), where each edge is described as a pair of the source node name and its output index. As each instruction can have multiple outputs, the output index is used to distinguish which output is connected to the corresponding node. For the first nodes (i.e., those with no input dependency to other instructions' completion), random values are supplied to their input operands. To this end, we introduce a pseudo node called root node. We add this root node to the sequence graph and edges between the root node and the first nodes.

For validity check, a hardware constraint information file is provided as shown in Table I. The constraint information is defined for each instruction supported in the target processor. It is composed of two parts: instruction allocation information and code generation information. The instruction allocation information includes allocable slot, input and output register types, and instruction latency. This type of information will be used in the instruction allocate step which will be explained shortly. The code generation information includes assembly syntax information such as instruction name and register order in the instruction.

```
"Inst1" : {
   "instType"  : "vmv.s.x.8",
   "numInput"  : 1,
   "numOutput" : 1
},
"Inst2" : {
   "instType"  : "vnclip.wv.8",
   "numInput"  : 2,
   "numOutput" : 1
},
"Inst3" : {
   "instType"  : "vmacc.vv.8",
   "numInput"  : 3,
   "numOutput" : 1
}
```

(a) Instruction Node

```
"Inst1" : [
   ["Root", 0]
],
"Inst2" : [
   ["Root", 1], ["Root", 2]
],
"Inst3" : [
   ["Inst1", 0], ["Inst2", 0], ["Root", 3]
],
"Result" : [
   ["Inst3", 0]
]
```

(b) Connection Edge

Figure 3. Example of instruction sequence format

TABLE I
EXAMPLE OF HARDWARE CONSTRAINT INFORMATION

Instruction Name	Assembly	Available Slot	Latency	...	Input Register Type	Output Register Type
vmv.s.x.8	vmv.s.x	1110	1	...	R[a]	V[a]
vnclip.wv.8	vnclip.wv	1100	1	...	V:V	V
vmacc.vv.8	vmacc.vv	1010	4	...	V:V:R	V
vwadd.vv.8	vwadd.vv	1010	2	...	V:V	V:V
…						

[a]R: Scalar register / V: Vector register

B. Instruction Selection and Allocation

Fig. 4 shows our instruction allocation and code generation algorithms. First, it parses sequence graph information (*json* file) and hardware constraint information (text file) and generates graph and hardware constraint data structures. At this stage, the number of graphs to be used can be specified according to verification efficiency or objectives. In addition, as our target processor supports hardware interleaved multi-thread, it determines which thread will execute the corresponding sequence graph.

In the instruction allocation stage, three internal data structures are used (*InstReadyQueue*, *RegAllocTable*, *SlotAllocTable*). *InstReadyQueue* maintains the instructions that can be allocated to instruction slots as their predecessor instructions are already allocated. It also includes a target line depth that represents the minimum code line it can be allocated considering the latency of its predecessors' instruction latencies. The instructions in the queue are sorted based on their target line depths to allocate the instructions from the front. *RegAllocTable* represents the usage of each register element and is used for register allocation. It includes allocation status and lifetime information to know when it can be released for other instructions. *SlotAllocTable* shows the status of instruction allocation and it is a table of size S × L (S: number of VLIW slots, L: number of code lines). Each element includes the information of the allocated instruction such as the instruction name and register information. This table will be used in the next step, code generation.

Here, we describe each step of the pseudocode in Fig. 4. Initially, *InstReadQueue* is filled with the first nodes with no predecessor instruction. We also mark the target line depth as 0 for these first nodes (lines 3 – 6). Then, it iteratively allocates the instructions in *InstReadyQueue* until all instructions are allocated (i.e., *InstReadQueue* becomes empty) (line 7). Before each instruction allocation, it checks if the already allocated registers in *RegAllocTable* are expired by checking the lifetime information so that the registers are freed to be allocated to other instructions. After the register status update, it extracts an instruction from *InstReadyQueue* (line 8). As explained earlier, instructions are sorted by their target line depths so that the instruction with the minimum line depth is prioritized for allocation. But, as there could be multiple instructions with the same line depth, a target instruction is randomly selected among those with the minimum line depth – this allows us to generate different instruction sequences from the same sequence graph when performing a regression test.

For the selected instruction, it checks if the required resources (slot and register) are available by checking *RegAllocTable* and *SlotAllocTable* (lines 11 – 12). If any resource is unavailable, the instruction is enqued again into *InstReadyQueue* again with the target line depth increased by 1 so that allocation for that instruction can be tried in the next line.

```
1: parseData(SequenceGraphInfo, HwConstraintInfo)
2: initialize(InstReadyQueue, RegAllocTable, SlotAllocTable)
3: for firstNode in firstNodeList {
4:     firstNode.lineDepth = 0
5:     InstReadyQueue.push(firstNode)
6: }
7: while(!InstReadyQueue.empty()) {
8:    targetNode = getMinLineDepthNode(InstReadyQueue) // Randomly choose the target node from the minimal line depth nodes
9:    curLineDepth = targetNode.lineDepth
10:   updateRegisterState(curLineDepth, RegAllocTable)
11:   slotList = checkAvailableSlot(curLineDepth, SlotAllocTable)
12:   regList = checkAvailableReg(curLineDepth, RegAllocTable)
13:   if (slotList && regList) {
14:         allocRandomSlot(curLineDepth, SlotAllocTable, slotList) // Slot randomization
15:         allocRandomReg(curLineDepth, RegAllocTable, regList) // Reg randomization
16:   }
17:   else targetNode.lineDepth = curLineDepth + 1
18:   for nextNode in targetNode {
19:      allInputAvailable = true
20:      for input in nextNode {
21:         if (input is not allocated) allInputAvailable = false
22:      }
23:      if (allInputAvailable) {
24:         nextNode.lineDepth = curLineDepth + targetNode.latency + rand()%MAX_RANGE // Latency randomization
25:         InstReadyQueue.push(node)
26:      }
27:   }
28: }
```

Figure 4. Pseudocode of instruction selection and allocation

If all resources are available, it randomly selects a slot and register from the available resources in *RegAllocTable* and *SlotAllocTable*, respectively (lines 14 – 15). Then, the status of the resources used in the current instruction is updated in *RegAllocTable* and *SlotAllocTable* so that they cannot be used for next allocations in the same cycle.

Lastly, it updates the status of the successor instructions of the current instruction, whether they become ready or not. If all the predecessor instructions are allocated, the successor instructions are added into *InstReadyQueue* with setting the target line depth as the value between [*cur line depth + instruction latency*] and [*cur line depth + max latency*] to consider real situation where the instruction allocation can be delayed due to the result of compiler scheduling (lines 24 – 25).

C. Code Generation

In the code generation, it generates an assembly code for the target sequence and a wrapper code written in C language. The assembly code generation utilizes the instruction information in *SlotAllocTable*. First, it generates an assembly instruction for each instruction slot considering the assembly syntax information specified in hardware constraint. Then, it merges the instruction in the same line to generate the final VLIW instruction for the corresponding line. The sequence of assembly VLIW instructions are incorporated into a single function.

In the C-wrapper code (*main*), an initialization code is added in order to load input data from an input file into the internal memory. Then, it calls the assembly function to run the target sequence. After the execution, the resulting register values are stored in a section of the memory. Then the final outputs in the memory are dumped into an output file for golden matching with the reference. In addition, coverage information like the instruction type and instruction dependency is generated for coverage measurement.

IV. VERIFICATION FLOW

Fig. 5 illustrates the overall verification flow using the proposed method. This flow verifies the instruction sequence by executing it through two paths and comparing their outputs. The first path is the reference C-model path. The randomly generated input data and the set of instruction sequences are fed into the reference C-model, which models the functional behavior of each instruction. The C-model produces output data, which serve as the golden reference (the correct result) for the processor operation. Note that, since each sequence is independent, the C-model processes the sequences one at a time. The second path is the processor hardware-simulation path, which utilizes the proposed RIS generator. As this path must reflect the target processor's complex architecture to ensure the validity of the generated test, additional hardware-specific information is supplied as input, unlike the reference C-model path. As described earlier, the proposed RIS generator receives three inputs and generates RIS test code that executes each sequence in parallel. The generated test code is then compiled into a binary, which is executed on a cycle-accurate instruction set simulator (CA ISS) or an RTL simulator. The outputs from the two paths are compared to determine whether the test passes. This verification flow is implemented in an UVM-based testbench environment. The RTL simulation and code-coverage measurements are performed using Cadence Xcelium and Integrated Metrics Collector (IMC).

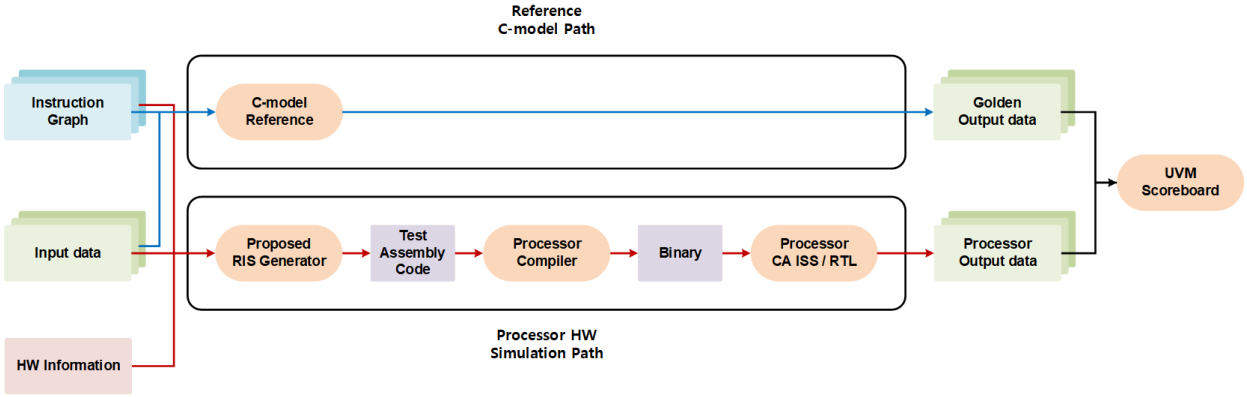

Figure 5. Verification flow with the proposed RIS generation

V. RESULTS

To measure the efficiency of generated tests, the proposed method is compared to a simple RIS method that we used as a baseline. Our simple RIS method is constructed as follows: each slot of a single bundle is randomly generated as long as it ensures no conflicts or compile errors occur. However, instruction latencies and dependencies between instructions in different lines (or bundles) are not considered. We rely on this simplification because modeling these factors is considerably difficult for complex processor architectures, such as those with application-specific optimizations like asymmetric VLIW slot configurations.

Using our proposed method, the verification of the target domain-specific VLIW processor produces a higher proportion of valid tests compared to the simple RIS method. In the case of a simple RIS, as the number of instruction bundles increases, the proportion of valid tests decreases exponentially. As shown in Fig. 6, it drops to approximately 55.5% for 20 instruction bundles and further decreases to 7.4% for 40 instruction bundles. In contrast, our proposed method achieves a significantly higher valid test generation rate, approaching close to 100%, provided that the instruction graph is well-defined, regardless of the number of instruction bundles.

Table II presents a comparison of the number of test vectors and coverage metrics between the simple RIS and the proposed RIS. The simple RIS does not consider HW constraint, which limits its capacity for generating valid tests and makes it challenging to create complex test cases. Furthermore, if the number of test cases is insufficient, tests for certain types of instructions may not be generated. In contrast, the proposed method allows for the direct insertion of desired sequences in the sequence set, enabling the generation of tests that include the required instructions while maintaining sufficient randomness. Consequently, despite using approximately half as many test vectors, the proposed method achieves comparable or higher coverage compared to the simple RIS. In our practical case study, the proposed method detected a bug in a specific functional unit. This bug was triggered by a rare input pattern, making it difficult to identify using prior regression test. The proposed method achieved higher verification efficiency by more frequently generating instruction sequences that are prone to inducing this critical input pattern.

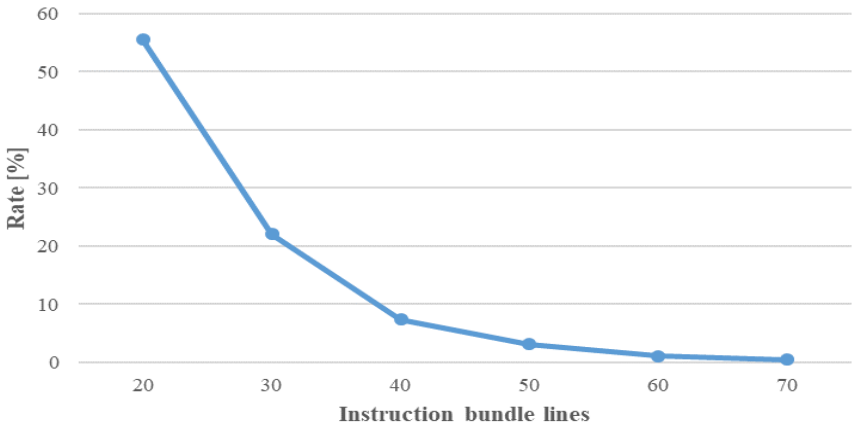

Figure 6. Valid test rate of the simple RIS method

TABLE II
COMPARISON OF CODE COVERAGE

			Simple RIS	Proposed RIS
Number of test vectors			2,334	1,177
Coverage	Block	Total bins	361,384	
		Covered bins	334,421	336,421
		Score	92.54%	93.09%
	Expression	Total bins	2,105,767	
		Covered bins	1,880,120	1,869,113
		Score	89.28%	88.76%
	Toggle	Total bins	2,542,774	
		Covered bins	2,192,105	2,326,438
		Score	86.21%	91.49%

VI. CONCLUSION

The contributions of the proposed RIS generation for DSA processors can be summarized as follows. First, by generating random instruction sequences that consider hardware constraints, a higher proportion of valid tests can be produced, thereby expediting verification and coverage closure. Second, its ability to create tests closer to real-case scenarios enhances verification quality compared to the conventional RIS. Finally, the verification direction can be controlled through the adjustment of constraints and pre-defined instruction graphs.

In conclusion, our graph-based RIS generation methodology offers improved efficiency and controllability compared to conventional methods. By utilizing instruction graphs extracted from target scenarios, the proposed method enables the generation of complex test cases that reflect the characteristics of the target workload while effectively handling complex architectural constraints. Therefore, it serves as an effective verification solution for DSAs, by balancing the precision of directed testing with the benefits of random generation.

REFERENCES

[1] Versal Adaptive SoC AIE-ML, https://docs.amd.com/r/en-US/am020-versal-aie-ml

[2] J. Redgrave, A. Meixner, N. Goulding-Hotta, A. Vasilyev, and O. Shacham, “Pixel visual core: Google’s fully programmable image vision and AI processor for mobile devices,” in Proc. Hot Chips Sympsium (HCS), 2018

[3] A. Adir et al., “Genesys-Pro: innovations in test program generation for functional processor verification,” in IEEE Design & Test of Computers, vol. 21, no. 2, pp. 84-93, March-April 2004

[4] A. Adir et al., “VLIW - a case study of parallelism verification,” in Proc. Design Automation Conference (DAC), 2005

[5] K. D. Rich, R. Shaw, S. G. Govindaraju, and D. Dobrikin, “DVGen: increasing coverage by automatically combining test specifications,” in Proc. High Level Design Validation and Test Workshop (HLDVT), 2006

[6] S. Bilava and S. Honwadkar, "A framework for verification of program control unit of VLIW processor," in Proc. DVCon India, 2014

[7] Synopsys ASIP Designer, https://www.snyopsys.com/dw/ipdir.php?ds=asip-deisgner

[8] Codasip Studio, https://codasip.com/products/codasip-studio/