LLM-based Processor Verification: A Case Study for Neuromorphic Processor

Chao Xiao[1], Yifei Deng[2], Zhijie Yang[1],Renzhi Chen[3], Hong Wang[2], Jingyue Zhao[3],
Huadong Dai[3], Lei Wang[3*], Yuhua Tang[1] and Weixia Xu[1]
[1] National University of Defense Technology, Changsha, China
[2] University of Electronic Science and Technology of China, Chengdu, China
[3] Defense Innovation Institute, AMS, Beijing, China

***Abstract*—With the increasing complexity of the hardware design, conducting verification before the tapeout is of utmost importance. Simulation-based verification remains the primary method owing to its scalability and flexibility. A comprehensive verification of modern processors usually requires numerous effective tests to cover all possible conditions and use cases, leading to significant time, resource, and manual effort even with the EDA. Moreover, novel domain specific architecture (DSA), such as neuromorphic processors, will exacerbate the challenge of verification. Fortunately, emerging large language models (LLMs) have been demonstrating a powerful ability to complete specific tasks assigned by human instructions. In this paper, we explore the challenges and opportunities encountered when using the LLMs to accelerate the DSA verification using the proposed LLM-based workflow consisting of test generation, compilation&simulation, and result collection&processing. By verifying a RISC-V core and a neuromorphic processor, we examine the capabilities and limitations of the LLMs when using them for the function verification of traditional processors and emerging DSA. In the experiment, 36 *C* programs and 128 assembly snippets for the RISC-V core and the neuromorphic processor are generated using an advanced LLM to demonstrate our claim. The experimental results show that the code coverage based on the LLM test generation can reach 89% and 91% for the above two architectures respectively, showing a promising research direction for the future processor verification in the new golden age for computer architecture.**

***Index Terms*—processor function verification, large language model (LLM), test generation, neuromorphic processor**

I. INTRODUCTION

Due to the ongoing scaling of the semiconductor manufacturing process, the complexities of microprocessors have increased significantly, leading to continuously increasing verification complexity. In 2022, as indicated in the Wilson Research Group Functional Verification Study [1], a substantial 40%∼50% of the entire hardware design cycle is dedicated to verification efforts.

Simulation-based methods co-simulate the design under test (DUT) with a reference model and check the deviation of their outputs given the same input stimuli. In the last few decades, in order to perform thorough coverage tests for comprehensive verification, simulation-based approaches including random test generation, constrained random generation (CRG) [2], coverage-directed test generation (CDG) [3], and coverage-directed test selection (CDS) [4], are proposed. Their success in achieving higher coverage is largely thanks to the design-specific optimizations. However, the next decade will witness the new golden age for computer architecture [5], and emerging domain specific architectures (DSAs), such as neuromorphic processors [6], [7], pose challenges due to the lack of readily available off-the-shelf tests and random testing EDA tools like RAVEN [8] for Arm processor and RISCV-DV [9] for RISC-V processor. Besides, to achieve high coverage, existing simulation-based methods require the expertise and involvement of experienced verification engineers which means much manual effort.

To generate tests efficiently and reduce manual efforts, a promising new approach is the application of large language models (LLMs) to automatically generate tests. LLMs are trained on a large amount of text data and can complete a wide range of tasks such as text summarization and language translation. The well-known LLMs include GPT-2/3/4 from OpenAI [10] and have been already applied for software code generation [11], [12] and hardware design [13]–[16].

DAVE is the first work to derive Verilog snippets from natural language via fine-tuned GPT-2 [13]. Both Chip-Chat [14] and ChipGPT [15] utilized the state-of-the-art ChatGPT to help designers generate hardware logic designs. Thakur et al. [16] examined fine-tuning five different pre-trained LLMs models and created a collection of Verilog coding problems to check the functional correctness of generated code. To the best of our knowledge, the use of LLMs to generate tests for processor verification has not been explored yet.

Therefore, in this paper, we aim at the following question: What are the potential advantages and obstacles to integrating the LLMs into traditional processor and DSA verification processes? To explore this question, we perform the LLM-based verification for *Xuantie-C910* [17], a traditional RISC-V processor, and a neuromorphic processor based on the custom neuromorphic instruction set extension using GPT-3.5. Besides, we construct an LLM-based verification workflow for DSA which consists of test generation, compilation&simulation, and result collection&processing, where the only part of human effort is to ‘converse’ with GPT-3.5. The test cases are written in two programming languages, *C* and assembly, for the *Xuantie-C910* and neuromorphic processor, respectively using LLM.

This work was supported in part by the National Natural Science Foundation of China under Grants 62372461, 62032001 and 62203457, and in part by the Key Laboratory of Advanced Microprocessor Chips and Systems.

* Corresponding author: leiwang@nudt.edu.cn

979-8-3503-4859-0/DATE24/© 2024 EDAA

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on July 07,2025 at 09:14:43 UTC from IEEE Xplore. Restrictions apply.

```
Prompts: Create a C program to calculate the sum of two integers.
Responses:
  #include <stdio.h>
  int main(){
      int num1, num2, sum;
      printf(”Enter the first integer:”);
      scanf(”%d”, &num1);
      printf(”Enter the second integer:”);
      scanf(”%d”, &num2);
      sum = num1 + num2;
      printf(”The sum of %d and %d is %d.\n”, num1, num2, sum);
      return 0;
  }
```

Fig. 1. An example of prompts to LLM and its response for generating a C program which performs the function of integers adding.

The LLM-generated tests are then compiled into executable binary files (ELFs) and serve as stimuli in the RTL-level simulator. After collecting and analyzing the simulation results, we prompt LLM to generate the next round of test cases, targeting the uncovered parts of the DUT. The main contributions of this paper are:

- We present the first attempt, to our best knowledge, to demonstrate the ability of conversational LLM for traditional processor and DSA function verification.
- We perform the function verification on both a RISC-V processor and a neuromorphic processor to explore the potential advantages and obstacles of LLM-based processor verification.
- We share our experience and observation in using LLM for functional verification of traditional processors and DSA such as neuromorphic processors, as well as potential future research directions of processor verification based on LLM.

In the experiment, 36 *C* programs and 128 assembly snippets for the RISC-V core and the neuromorphic processor are generated using an advanced LLM to demonstrate our claim. The experimental results show that the code coverage based on the LLM test generation can reach 89% and 91% for the above two architectures respectively, showing a promising research direction for the future processor verification.

II. BACKGROUND

A. Processor Verification

Processor verification is indeed crucial for ensuring the functionality and reliability of the processor. Testing programs include bare metal programs, Uboot, operating systems, and more complex applications. However, it is challenging to obtain a suitable testing program and complete as much verification of the processor as possible within a limited time. For ARM architecture processors, the RAVEN [8] testing tool can be used. For RISC-V, the RISCV-DV [9] tool can be used. However, these approvals are not designed for Domain Specific Architecture (DSA) like neuromorphic processors, requiring a lot of manual effort for the verification. Code coverage is widely adopted by simulation-based verification, about 75% for ASIC projects and 55% for FPGA projects [1]. Therefore, code coverage is one of the most important indexes to evaluate the verification progress and quality.

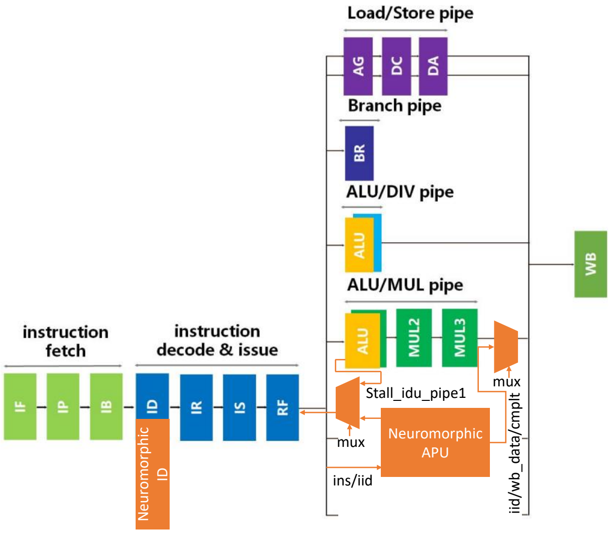


Fig. 2. Block diagram of tightly-coupled neuromorphic processor extended from *Xuantie-C910* using ISA-extension method.

B. Large Language Models (LLMs)

LLMs, consisting of hundreds of billions (or even more) of parameters, are generally based on transformer architectures. GPT-4 is the latest version of the GPT family. The inputs to LLMs are commonly referred to as ‘prompts’, where users can ‘program’ LLMs to successfully complete a specific task. A case where we ask GPT-3.5 to write a *C* program to calculate the sum of two integers is shown in Figure 1. It is apparent that GPT-3.5 not only understands the intent of the human prompts but also responds correctly and writes high-quality code.

C. The Processor under Verification

Neuromorphic computing aims to design and build computer systems inspired by the structure and function of the human brain. As the running platform of neuromorphic algorithms, neuromorphic processors such as TrueNorth [6] and Loihi [7] are proposed these years, showing unforgettable energy efficiency when compared with traditional processor architectures such as CPU, GPU, and DNN accelerators.

In order to accelerate neuromorphic computing, we extend the neuromorphic instruction set based on the RISC-V instruction set and implement tightly coupled neuromorphic computing units in the CPU core based on neuromorphic instructions. Our neuromorphic instruction set defines 10 types of extension registers, including weight vector, spike vector, output spike vector, neuron type/voltage threshold value parameters, and so on. In addition, the neuromorphic instruction set uses 3 instruction formats to define 8 types of 33 instructions, including wide vector loading&storing, data movement, neuron current and accumulation (vertical/horizontal), and so on.

We have implemented our neuromorphic instruction set-based micro-architecture extended on the high-performance open-source RISC-V processor *Xuantie-C910* [17]. The architecture of the extended neuromorphic processor is shown in Figure 2. The original *Xuantie-C910* processor core adopts a superscalar architecture with 3 decoding and 8 execution, with a pipeline depth of 9-12 levels and a working frequency of 1GHz. It can support various functions, such as cache prefetching,

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on July 07,2025 at 09:14:43 UTC from IEEE Xplore. Restrictions apply.

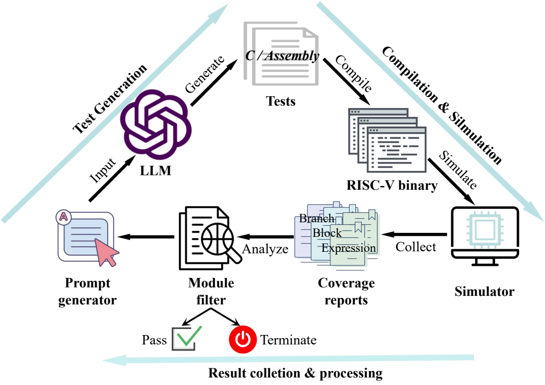


Fig. 3. The overview of LLM-based processor verification components and workflow which consists of test generation, compilation&simulation, and result collection&processing.

cache path prediction, branch prediction, unordered memory access, etc., to improve performance and reduce the power consumption of the processor. The new neuromorphic processor core supports the execution of neuromorphic instructions mentioned above.

We have extended the vector register to store spikes, weights, and neuron states related to the neuron model; We extend the memory access system to implement ScratchPad Memory (SPM) and load store unit to improve the memory bandwidth; We implement the computing components to achieve two computing methods (”synapse-wise and neuron-wise”) and a parallel computing neuron array consisting of 512 neurons units.

III. METHODOLOGY

A. Overview

Figure 3 shows the overview of how we apply LLMs to perform processor-level function verification. It generally contains three parts: test generation, compilation&simulation, and result collection&processing.

The workflow of our LLM-based test generation is as follows. Test generation focuses on creating new tests based on the processed results to cover more possible conditions. The test cases are written in *C* and assembly for RISC-V ISA and neuromorphic ISA, respectively. Then, the generated tests are compiled into executable binary files, which are served as stimuli in the simulator. After the simulation, different types of coverage information are collected and processed. Based on the processed results, verification engineers examine the uncovered conditions along with their underlying reasons. Subsequently, they provide feedback to the LLMs regarding the strategies generated by the subsequent round of tests. Then, the next iteration of verification begins.

B. Test Generation for RISC-V Processor

In this work, we are committed to exploring the use of LLM to generate tests. We need to prompt the LLM on what test programs we need, and how LLMs generate the corresponding tests is out of scope for the purposes of this paper. We use GPT-3.5 as the basic LLM.

```
Prompts: please write a C program containing the following
      operations: (1)signed addition, (2)signed immediate
      addition, (3)logical shift left .
Responses:
#include <stdio.h>
  int main() {
    int num1 = 10;
    int num2 = 20;
    int result ;
    // Signed addition
    result = num1 + num2;
    printf(”Signed Addition: %d\n”, result);
    // Signed immediate addition
    result = num1 + 5;
    printf(”Signed Immediate Addition: %d\n”, result);
    // Logical shift left
    result = num1 << 2;
    printf(”Logical Shift Left: %d\n”, result);
    return 0;
  }
```

(a)

```
Prompts: Please help us write a complex C
      program to verify the branch prediction
      technology of the processor.
Responses:
  #include <stdio.h>
  #define SIZE 1000000
  int main() {
  // Array with a large number of elements
  int array[SIZE];
  int i;
  int sum = 0;
  // Initialize array with random values
  for (i = 0; i < SIZE; i++) {
    array[i] = rand() % 100;
  }
  // Compute the sum of all elements
  for (i = 0; i < SIZE; i++) {
    if (array[i] < 50) {
      sum −= array[i];
      } else {
        sum += array[i];
      }
  }
  printf(”Sum: %d\n”, sum);
  return 0;
}
```

(b)

Fig. 4. Prompts to GPT-3.5 in RISC-V processor test generation. (a) an example of generating C programs for testing RISC-V ISA. (b) an example of our prompts for specific function unit test generation and the LLM responses for testing the branch prediction.

1) ISA Prompts: The fact that instruction set architecture defines the interface between the software and hardware components motivates us to use the ISA as the prompt to GPT-3.5. To do this, we take the function description of each RISC-V instruction implemented in the target processor as the prompt. Figure 4(a) shows the content of the conversation with GPT-3.5.

2) Prompts for Hardware-aware Test Generation: Modern processors consist of many specific function units such as branch direction prediction in *Xuantie-C910*, to achieve high-performance computing. For these specific function units, we provide GPT-3.5 with the corresponding prompts. Figure 4(b) shows how we prompt GPT-3.5 to generate tests for the verification of the branch prediction module. In the conversation, GPT-3.5 first calls the *rand()* function to initialize the array, complicating the following branches. Then, it creates a loop body with 1x10^6 iterations, which contains two branches. Clearly, GPT-3.5 has understood our prompts and effectively generated the testing programs.

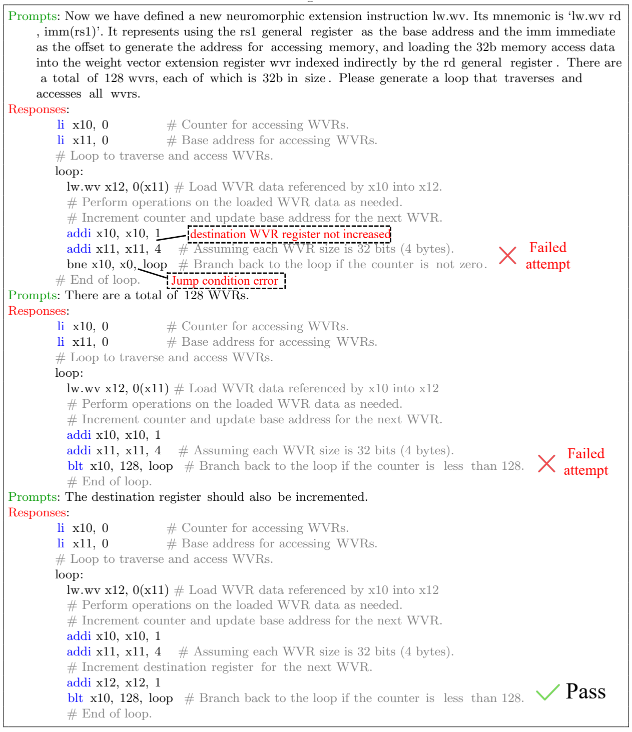
C. Test Generation for the Neuromorphic Architecture

Since the neuromorphic instruction set is unknown to GPT-3.5, the first step before test generation is to teach GPT-3.5 how to write a neuromorphic application based on the custom neuromorphic instruction set.

The testing process is performed as follows. Firstly, we teach GPT-3.5 the function and usage of each neuromorphic instruction. For example, Figure 5 shows the prompts and responses when teaching GPT-3.5 the *lw.wv* instruction, an instruction to load synaptic weights from memory to the extended Weight Vector Register (WVR). Each synaptic weight is stored in 4 bits and *WVRs* are the 128x32bits weight vector registers. As shown in Figure 6, since the LLM has already understood the usage of the instruction we teach, a simple assembly program code segment with cyclic weights and spike loading can be generated by the LLM then.

The details of the testing results and our analysis are as follows. Regarding this novel neuromorphic instruction, generating the correct code is very challenging for GPT-3.5. It often requires iterative refinement and collaboration with humans to ensure the generated code aligns with the intended functionality. As shown in Figure 5, remarkably, despite the vagueness of our

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on July 07,2025 at 09:14:43 UTC from IEEE Xplore. Restrictions apply.

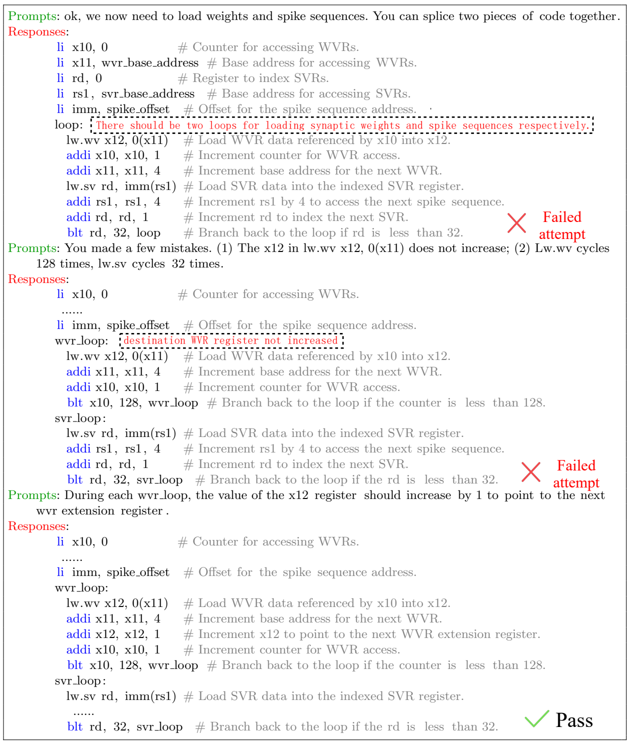


```
Prompts: Now we have defined a new neuromorphic extension instruction lw.wv. Its mnemonic is ‘lw.wv rd
    , imm(rs1)’. It represents using the rs1 general register as the base address and the imm immediate
    as the offset to generate the address for accessing memory, and loading the 32b memory access data
    into the weight vector extension register wvr indexed indirectly by the rd general register . There are
     a total of 128 wvrs, each of which is 32b in size . Please generate a loop that traverses and
    accesses all wvrs.
Responses:
        li  x10, 0              # Counter for accessing WVRs.
        li  x11, 0              # Base address for accessing WVRs.
        # Loop to traverse and access WVRs.
        loop:
          lw.wv x12, 0(x11) # Load WVR data referenced by x10 into x12.
          # Perform operations on the loaded WVR data as needed.
          # Increment counter and update base address for the next WVR.
          addi x10, x10, 1
          addi x11, x11, 4    # Assuming each WVR size is 32 bits (4 bytes).
          bne x10, x0, loop   # Branch back to the loop if the counter is not zero.
        # End of loop.
Prompts: There are a total of 128 WVRs.
Responses:
        li  x10, 0              # Counter for accessing WVRs.
        li  x11, 0              # Base address for accessing WVRs.
        # Loop to traverse and access WVRs.
        loop:
          lw.wv x12, 0(x11) # Load WVR data referenced by x10 into x12
          # Perform operations on the loaded WVR data as needed.
          # Increment counter and update base address for the next WVR.
          addi x10, x10, 1
          addi x11, x11, 4    # Assuming each WVR size is 32 bits (4 bytes).
          blt  x10, 128, loop   # Branch back to the loop if the counter is less than 128.
          # End of loop.
Prompts: The destination register should also be incremented.
Responses:
        li  x10, 0              # Counter for accessing WVRs.
        li  x11, 0              # Base address for accessing WVRs.
        # Loop to traverse and access WVRs.
        loop:
          lw.wv x12, 0(x11) # Load WVR data referenced by x10 into x12
          # Perform operations on the loaded WVR data as needed.
          # Increment counter and update base address for the next WVR.
          addi x10, x10, 1
          addi x11, x11, 4    # Assuming each WVR size is 32 bits (4 bytes).
          # Increment destination register for the next WVR.
          addi x12, x12, 1
          blt  x10, 128, loop   # Branch back to the loop if the counter is less than 128.
          # End of loop.
```

Fig. 5. Example of teaching GPT-3.5 the *lw.wv* extended neuromorphic instruction. After two attempts, LLM understands the instruction and generates the correct program for testing different conditions of the instruction execution.

further prompt, GPT-3.5 demonstrates an impressive ability to pinpoint the source of the error directly. After the initial prompt, we further provide two vague prompts to correct errors in the initial program. To our surprise, GPT-3.5 swiftly identifies the precise location of the mistake. For instance, in the second prompt, where we mention a total of 128 *WVR* registers, GPT-3.5 not only comprehends the need to iterate 128 times but also accurately modifies the error by changing '*bne x10 x0 loop*' to '*blt x10 128 loop*'. This remarkable capability showcases GPT-3.5's programming language understanding and its ability to address errors even in ambiguous contexts.

As shown in Figure 6, once GPT-3.5 has acquired a solid understanding of the fundamental usage and functionality of each extended neuromorphic instruction, we engage it in combining multiple instructions to create complex test programs. As a result, GPT-3.5 generates the assembly program to load synaptic weights and spike sequences. The *lw.sv* instruction, similar to the *lw.wv*, is to load spike sequences from memory to the 32 extended Spike Vector Registers (SVRs). Regardless of the errors in the initially generated programs, it reflects the capability of LLMs to retain and effectively apply previously acquired knowledge. Based on this fact, we prompt GPT-3.5 to generate a set of combinational test programs with different neuromorphic instructions, which are then translated into machine code using an assembler and subsequently sent to the simulator as stimuli. This process leverages GPT-3.5's knowledge to synthesize intricate sequences of neuromorphic instructions and generate comprehensive test conditions.



```
Prompts: ok, we now need to load weights and spike sequences. You can splice two pieces of code together.
Responses:
        li  x10, 0              # Counter for accessing WVRs.
        li  x11, wvr_base_address  # Base address for accessing WVRs.
        li  rd, 0               # Register to index SVRs.
        li  rs1, svr_base_address  # Base address for accessing SVRs.
        li  imm, spike_offset   # Offset for the spike sequence address.
        loop:
          lw.wv x12, 0(x11)    # Load WVR data referenced by x10 into x12.
          addi x10, x10, 1     # Increment counter for WVR access.
          addi x11, x11, 4     # Increment base address for the next WVR.
          lw.sv rd, imm(rs1) # Load SVR data into the indexed SVR register.
          addi rs1, rs1, 4     # Increment rs1 by 4 to access the next spike sequence.
          addi rd, rd, 1       # Increment rd to index the next SVR.
          blt  rd, 32, loop    # Branch back to the loop if rd is less than 32.
Prompts: You made a few mistakes. (1) The x12 in lw.wv x12, 0(x11) does not increase; (2) Lw.wv cycles
    128 times, lw.sv cycles 32 times.
Responses:
        li  x10, 0              # Counter for accessing WVRs.
         ......
        li  imm, spike_offset   # Offset for the spike sequence address.
        wvr_loop:
          lw.wv x12, 0(x11)    # Load WVR data referenced by x10 into x12.
          addi x11, x11, 4     # Increment base address for the next WVR.
          addi x10, x10, 1     # Increment counter for WVR access.
          blt  x10, 128, wvr_loop  # Branch back to the loop if the counter is less than 128.
        svr_loop:
          lw.sv rd, imm(rs1) # Load SVR data into the indexed SVR register.
          addi rs1, rs1, 4     # Increment rs1 by 4 to access the next spike sequence.
          addi rd, rd, 1       # Increment rd to index the next SVR.
          blt  rd, 32, svr_loop   # Branch back to the loop if the rd is less than 32.
Prompts: During each wvr_loop, the value of the x12 register should increase by 1 to point to the next
    wvr extension register .
Responses:
        li  x10, 0              # Counter for accessing WVRs.
         ......
        li  imm, spike_offset   # Offset for the spike sequence address.
        wvr_loop:
          lw.wv x12, 0(x11)    # Load WVR data referenced by x10 into x12.
          addi x11, x11, 4     # Increment base address for the next WVR.
          addi x12, x12, 1     # Increment x12 to point to the next WVR extension register.
          addi x10, x10, 1     # Increment counter for WVR access.
          blt  x10, 128, wvr_loop  # Branch back to the loop if the counter is less than 128.
        svr_loop:
          lw.sv rd, imm(rs1) # Load SVR data into the indexed SVR register.
           ......
          blt  rd, 32, svr_loop   # Branch back to the loop if the rd is less than 32.
```

There should be two loops for loading synaptic weights and spike sequences respectively.
Failed attempt
destination WVR register not increased
Failed attempt
Pass

Fig. 6. Example of prompts to GPT-3.5 for generating code segment with the loop of loading synaptic weights and spike train operands, which can be seen as part of a simple neuromorphic application. This example shows the ability of LLM for understanding unknown knowledge and generating tests for DSA.

D. Compilation & Simulation

The generated tests are compiled into executable binary files, which are subsequently utilized as stimuli within the simulator. Once binary files are obtained, we identify the instructions contained within and proceed to search for instructions that have not been covered. Throughout the simulation process, we diligently record three coverage metrics for each module. These metrics facilitate a comprehensive evaluation of the verification progress, allowing us to identify coverage gaps and drive improvements in the test suite for each module.

E. Result Collection & Processing

1) Coverage Collection: After the simulation, we parse the log files to obtain the different types of coverage, including block coverage, expression coverage, and toggle coverage for further understanding and re-prompt to LLM.

2) Module Filter: Module Filter is proposed in this work to timely find modules whose coverage achieves satisfactory thresholds in the test generation loop, thereby terminating the loop.

In detail, during the verification process, certain modules of the DUT may quickly achieve high coverage, even reaching 100%. This occurrence is typically observed when the module's functionality is simple, and the designed tests effectively cover the designated functionality. For example, the *ct_iu_alu* module in *Xuantie-C910* is responsible for handling simple operations like left shift and addition. The coverage for this module quickly achieves a high level by incorporating arithmetic opera-

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on July 07,2025 at 09:14:43 UTC from IEEE Xplore. Restrictions apply.

TABLE I
ISA COVERAGE USING LLM TO GENERATE THE TEST PROGRAM

Instruction Type	Num. of Ins.	Covered	Coverage
RV64I	63	57	90.48%
RV64M	13	12	92.31%
RV64A	22	21	95.45%
RV64F	30	29	96.67%
RV64D	32	31	96.88%
RV64C	37	29	78.38%
Neuromorphic ISA	33	33	100.00%

tions. For such 'simple' modules, once the coverage reaches the defined threshold, they are deemed as passed in our workflow.

Consequently, generating additional tests specifically for these passed modules is no longer pursued, taking into account considerations such as diminishing returns, resource optimization, and time constraints. This approach allows for the efficient allocation of verification resources, prioritizing the assessment of more complex and critical modules while still ensuring that the entire system undergoes thorough verification.

If, despite multiple rounds of testing, the coverage of certain modules remains below the set threshold, it may indicate that the LLM is unable to generate effective tests targeting the uncovered parts of the DUT based on the currently provided prompts. The reasons for this failure can be multi-faceted and include factors such as neglecting edge cases, intricate dependencies or interactions between modules, and more. Consequently, to mitigate testing costs, it becomes necessary to terminate test generation specifically for these 'difficult' modules.

In addition to the previously mentioned two types of modules, the remaining modules will continue the iterative testing until the 'termination' or 'pass' conditions are met.

IV. EXPERIMENT

A. *Experiment Setup*

In this paper, we utilize a native LLM, i.e., GPT-3.5, specifically without fine-tuning it for processor function verification. Our approach involves generating tests using GPT-3.5 based on the given prompts. For the RISC-V ISA, we generate 36 *C* programs, while for the neuromorphic ISA, we generate 128 assembly programs. To assess the effectiveness and quality of the generated tests, we have evaluated the following three metrics: *ISA coverage:* This metric measures the coverage of the ISA by the generated tests. It determines the extent to which different instructions are tested. *Block coverage:* It measures the percentage of basic blocks executed by the generated tests, helping identify areas that may have been skipped or insufficiently tested. *Expression coverage:* It is a mechanism that factorizes logical expressions and monitors them during the simulation run. It provides metrics to quantify the degree of verification completeness. *Toggle coverage:* It measures the activity of various signals in a design and provides information on untoggled signals or signals that remain constant during the simulation run.

B. *Experiment Results*

1) ISA Coverage: The ISA coverage is shown in Table I. For the neuromorphic ISA, the generated tests cover all neuromorphic instructions because the tests are written in assembly directly. For the RISC-V ISA, it's important to note that the tests are written in the *C* programming language. Due to the nature of the *C* language and its compilation process, it can be challenging to achieve full coverage. The six uncovered instructions in RV64I are *csrrc*, *csrrci*, *csrrs*, *csrrsi*, *csrrw*, and *ebreak*. The first five instructions are used for interacting with CSRs (Control and Status Registers). The *ebreak* is commonly used as a debugging or breakpoint instruction. It is apparent that achieving coverage of these six special instructions in a compiled *C* program can be challenging.

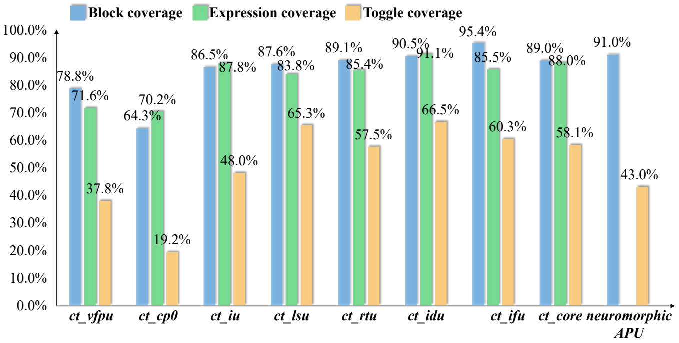


Fig. 7. Block, expression, and toggle coverage the LLM-generated test can achieve in RISC-V processor and neuromorphic processor verification.

2) Block, Expression, and Toggle Coverage: The block, expression, and toggle coverage are depicted in Figure 7. The *ct_vfpu*, *ct_cpu0*, *ct_iu*, *ct_lsu*, *ct_rtu*, *ct_idu*, and *ct_ifu* are sub-modules of *ct_core*. The total block and expression coverage of *ct_core* reach 88.98% and 88.01%. Notably, the block coverage of the neuromorphic extension reaches 91.02%. This can be attributed to the SIMD nature of the neuromorphic architecture, which facilitates a more straightforward verification process and consequently yields higher block coverage. It is crucial to highlight that the toggle coverage remains generally low for all modules. This can be mainly attributed to the intricacies involved in evaluating signal flips within the design code. While toggle coverage proves useful in gate-level testing, controlling and managing these signal flips becomes particularly challenging when dealing with high-level language programs such as *C* programs.

V. DISCUSSION

Speed and scale of LLM-based verification: Although we have automated crucial stages such as result collection and processing, one significant aspect remains manual—the need for manual engagement in each conversation with GPT-3.5. The manual participation process involves human experts examining the coverage outcomes and formulating prompts or queries that can effectively guide GPT-3.5 toward addressing the uncovered parts of the DUT. To address this challenge and increase the speed and scale of LLM-based verification, future efforts should prioritize exploring techniques and tools for automatic conversion of coverage results into natural language prompts. The objective of this automated conversion process should be creating prompts specifically targeted at undiscovered parts of the DUT.

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on July 07,2025 at 09:14:43 UTC from IEEE Xplore. Restrictions apply.

Scalability for novel DSA: Our research showcases the tremendous potential of LLMs in the realm of functional verification for novel DSA. With the appropriate teaching and deep understanding of new architectural features, GPT-3.5 demonstrates its capability to generate tests that effectively perform the functionality verification of these novel processors. It should be noted that in this study, we use a dialogue-based approach to facilitate GPT-3.5 in acquiring an understanding of the intricate nature of neuromorphic ISA and this process still needs human participation, which can be considered as a limitation. To address this limitation, future endeavors should consider exploring the integration of deep learning methods to fine-tune LLMs. By doing so, LLMs can be trained to autonomously adapt and comprehend the novel processor architecture. This autonomous adaptation empowers the LLMs to generate tests and perform functional verification without heavy reliance on human involvement.

Effectiveness of generated tests: Based on the conversation with GPT-3.5 and the experimental results, it is evident that the generated tests exhibit a high level of effectiveness and are well-suited for addressing uncovered conditions. This effectiveness can be attributed to the careful consideration and planning used by the verification engineers when preparing the prompts for each conversation.

VI. RELATED WORKS

To promote the efficiency and quality of verification, many test generation methods have been proposed. Constrained random generation (CRG) [2] is a test generation technique, which specifies and solves constraints to generate new tests thus creating better tests than pure random generation. Coverage-directed test generation (CDG) [3] leverages artificial intelligence (AI) techniques to automate test biasing during the verification process. However, one of the challenges with CDG is that it requires significant domain knowledge to be encoded into the AI model. Coverage-directed test selection (CDS) [4], based on supervised learning from coverage feedback, selects effective tests from the large set of generated tests and prioritizes them for simulation. Novelty-driven verification [18] constructs an unsupervised model of similarity from previous tests and ranks newly generated tests, with preference given to the most dissimilar.

Recently, researchers have been exploring and applying LLMs for various professional uses. Codex, fine-tuned on millions of open-source software repositories from GitHub, solves 28.8% of the programming problems on the HumanEval dataset [11]. Nijkamp et al. [12] have trained and released a family of LLMs up to 16.1B parameters for program synthesis. Despite the application of LLMs in the software domain, there are also many works focused on the hardware domain. DAVE [13], the first work exploring the use of LLMs in the hardware design, is fine-tuned from the GPT-2 model over synthetically generated 'English-Verilog' pairs. Given that LLMs perform effectively in an interactive manner, Blocklove et al. [14] conducted a conversational case study where a hardware engineer collaborated to co-design an 8-bit accumulator-based microprocessor architecture, using ChatGPT-4. It is the first time to write the complete HDL for a tape out using AI. However, to the best of our knowledge, the use of LLMs to generate tests for processor verification has not been explored yet.

VII. CONCLUSION

In this work, we have applied GPT-3.5 to conduct the functional verification on both traditional and novel processor architecture. To test and validate new processor architectures, the LLMs can efficiently generate effective test cases through the process of learning and adaptation. Collaborating with LLMs allows verification engineers to focus more on identifying potential deficiencies in the current iteration, rather than investing excessive effort in test generation. Our future direction aims at fully automating the entire verification process. It includes two key aspects. Firstly, eliminating the human involvement to analyze and identify defects in each iteration of the verification process. Secondly, when confronted with new architectures, automatically teaching the LLMs on a new instruction set architecture.

REFERENCES

[1] Foster et al. The 2022 wilson research group functional verification study. https://blogs.sw.siemens.com/verificationhorizons/2022/10/10/prologue-the-2022-wilson-research-group-functional-verification-study/, 2022.

[2] Kitchen et al. Stimulus generation for constrained random simulation. In *2007 IEEE/ACM International Conference on Computer-Aided Design*, pages 258–265, 2007.

[3] Fine et al. Coverage directed test generation for functional verification using bayesian networks. In *Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)*, pages 286–291, 2003.

[4] Masamba et al. Supervised learning for coverage-directed test selection in simulation-based verification. In *2022 IEEE International Conference On Artificial Intelligence Testing (AITest)*, 2022.

[5] Jeff et al. A new golden age in computer architecture: Empowering the machine-learning revolution. *IEEE Micro*, 38(2):21–29, 2018.

[6] Merolla et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668 – 673, 2014.

[7] Davies et al. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, pages 82–99, 2018.

[8] Andrea et al. Raven- a real-time analysis and verification environment. *Proc. of DSP Deutschland 97*, 1997.

[9] Michalak et al. Riscv-dv. https://github.com/chipsalliance/riscv-dv, 2023.

[10] OpenAI. Gpt-4 technical report, 2023.

[11] Chen et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[12] Nijkamp et al. Codegen: An open large language model for code with multi-turn program synthesis, 2023.

[13] Pearce et al. DAVE: Deriving automatically verilog from english. In *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*. ACM, nov 2020.

[14] Blocklove et al. Chip-chat: Challenges and opportunities in conversational hardware design, 2023.

[15] Chang et al. Chipgpt: How far are we from natural language hardware design, 2023.

[16] Thakur et al. Benchmarking large language models for automated verilog rtl code generation. In *2023 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, 2023.

[17] Chen et al. Xuantie-910: A commercial multi-core 12-stage pipeline out-of-order 64-bit high performance risc-v processor with vector extension : Industrial product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 52–64, 2020.

[18] Blackmore et al. Novelty-driven verification: Using machine learning to identify novel stimuli and close coveragen. In *Proceedings of the 2021 Design and Verification Conference (Virtual)*, 2021.

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on July 07,2025 at 09:14:43 UTC from IEEE Xplore. Restrictions apply.