TiDe : Timing diagram to Design verification model

Ramya B T, Samsung Semiconductor India R&D Center, Bangalore, India
ramya.bt@samsung.com
Sairam Jujjarapu, Samsung Semiconductor India R&D Center, Bangalore, India
s.jujjarapu@samsung.com
Harshit Sharma, Samsung Semiconductor India R&D Center, Bangalore, India
h4.sharma@samsung.com
Sachin Suresh Upadhya, Samsung Semiconductor India R&D Center, Bangalore, India
sachin1.s@samsung.com
Keerthi Kiran J, Samsung Semiconductor India R&D Center, Bangalore, India
keerthi.k@samsung.com

Abstract - *Design verification is a highly time consuming process for design as well as verification engineers. It involves huge manual effort even with the best of the tools available in the market. In today's world where NLP and LLMs are widely being used to solve problems in digital design verification, we propose TiDe model which can streamline the verification process at various levels. The model takes timing diagram as input directly from spec and generates: 1. Unit testcases for basic testing and 2. Assertions for Formal verification. When fine-tuned properly, TiDe model is highly effective in reducing the testbench development effort for both the design and verification engineers approximately 80% and 60% respectively.*

Keywords *: Design verification, RTL (Register Transfer Level), LLM (Large Language Model), LLaVa (Large Language and Vision Assistant), Deepseek, Formal verification, Unit testbench, Basic testbench, Timing diagram*

I. Introduction

Design verification is the process of ensuring that the intent of the specification is preserved in the implementation of RTL. It can be broadly classified into two categories: Functional verification and Formal verification. Functional verification is the task of verifying that the logic design conforms to the specification [1]. Formal verification is the methodology of verifying the correctness of the system with respect to properties that are derived from formal specifications [2]. Formal Verification is an important aspect of EDA (Electronic design automation) and is graded the highest EAL-7 (Evaluation Assurance Level) in the framework of common criteria for computer security certification. Formal verification examines the full space of possible simulation, rather than testing out only with specific values with random stimulus generation. It includes an exhaustive set of all possible values of parameters that can be used to uncover all possible bugs of the RTL and is highly profitable in target logic verification. Most of the formal verification implementation consists of SVA (System Verilog Assertions) which are assertion statements that generate error when a specified condition or sequence fails to comply with the specification.

On the other hand, before the actual functional and formal verification are done, the design engineers carry out unit testing, by executing a set of testcases referred to as Unit testbench or Basic testbench. These testcases cover basic functionalities of the RTL at module level that were intended to be implemented as per specification [3].

II. Problem Statement

All three forms of testing described in the Introduction section, require significant amount of time for development of the testcase code and stabilization. While the design engineers must take care of developing RTL as well as the unit testbench, the verification engineers must plan an exhaustive verification process.

In this paper, we present TiDe model which is a framework resulted as a combination of two popular LLMs : LLaVa and Deepseek bound with a custom script. The 2-LLM design was finalized after several experiments with single model failed to perfectly convert the timing diagrams to system verilog code. Vision transformers like Fuyu 8B, CongVLM and OCRs (Optical Character Recognition) like Tesseract, CLIP (Contrastive Language-Image Pretraining) like BLIP-2 could not describe the image elaborate enough to generate expected code. LLaVa and Deepseek were chosen after extensively experimenting with models like Qwen, Salesforce and Meta-Llama whose results were not as per expectation.

For the purpose of quantification of results, we have collected data from a team of Design experts and Formal verification experts. And the results and conclusions in this paper, have been drawn based on the information collected from them.

III. PROPOSED METHODOLOGY

The architecture of TiDe is represented in figures 1a and 1b. Our proposed TiDe model accepts timing diagram from spec/document as input and generates: 1. Unit testcases (Unit testbench generator block in figure 1c) and 2. System Verilog Assertions for Formal verification (Assertion generator block in figure 1c). The model uses a combination of two LLMs to achieve this along with a custom script, based on the execution environment and the RTL. A pre-trained LLaVa model takes the timing diagram as input with script-generated prompt and decodes the image embeddings and extracts all the information pertaining to the timing diagram. This information is used along with another prompt generated with the script to analyze and generate the unit testbench code and SVA statements in system verilog language.

TiDe has been designed such that its Unit testbench generator can generate almost all possible unit testcases to help the design engineers (~100% for simpler diagrams) and TiDe's Assertion generator can generate most of the properties and assertion statements for formal verification engineers (~90% for a highly descriptive diagram). The functional flow diagram of TiDe is represented in Figure 1c. Figure 1d details the testing process flow of TiDe.

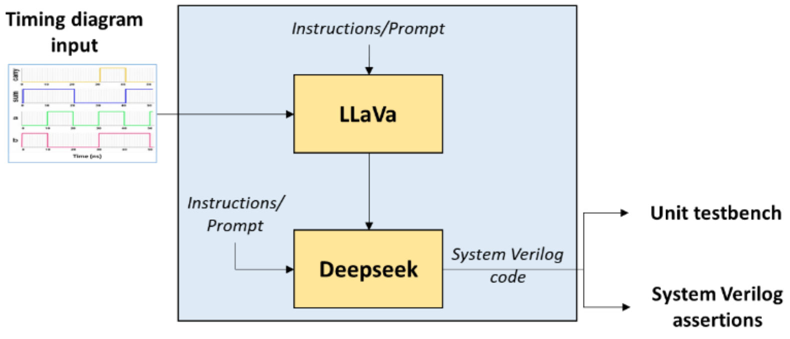


Figure 1a: Architecture of TiDe model

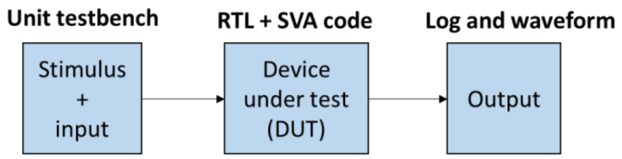


Figure 1b: Architecture of testing setup for TiDe model

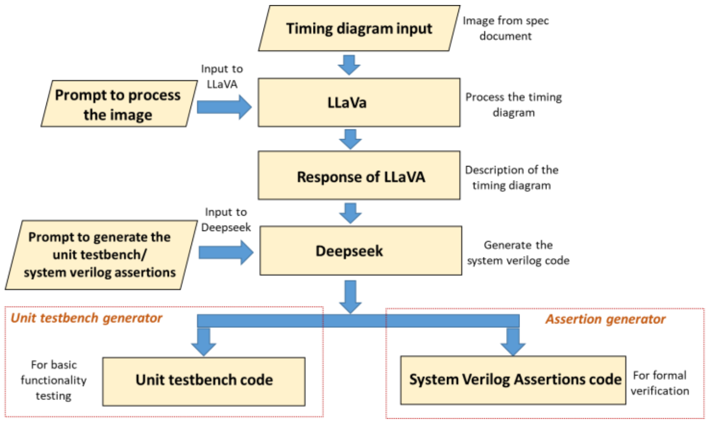

Timing diagram input
Image from spec document
Prompt to process the image
Input to LLaVA
LLaVa
Process the timing diagram
Response of LLaVA
Description of the timing diagram
Prompt to generate the unit testbench/ system verilog assertions
Input to Deepseek
Deepseek
Generate the system verilog code
Unit testbench generator
Assertion generator
For basic functionality testing
Unit testbench code
System Verilog Assertions code
For formal verification

Figure 1c: Code generation with TiDe model

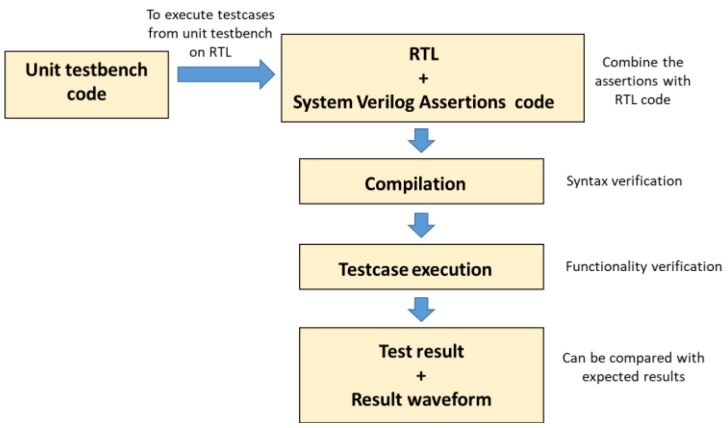

To execute testcases from unit testbench on RTL
Unit testbench code
RTL + System Verilog Assertions code
Combine the assertions with RTL code
Compilation
Syntax verification
Testcase execution
Functionality verification
Test result + Result waveform
Can be compared with expected results

Figure 1d: Verification of the code generated by TiDe model

Benchmarking

TiDe model's generated code is benchmarked on the basis of 2 parameters: **Syntax** and **Functionality** [7]. Syntax verification is done by checking the compilation of the unit testbench and SVA code on EDA playground [6]. Functionality verification is done by intently injecting error into the RTL. Tests from unit testbench are first executed on the RTL+SVA combination and the expected result would be for the testcase to fail. Similarly, assertion has to be hit in formal verification process in the property corresponding to the injected error. If these criteria are met, the functional verification is said to be accomplished.

IV. EXPERIMENTAL SETUP

The setup comprises of both hardware infrastructure and software components [4] [5]. All experiments were performed on a Quadro RTX 6000 GPU with 4608 CUDA cores and 24GB of GDDR6 memory. This GPU provides high computational power suitable for intensive tasks. To verify the syntax and functionality of the generated code, sample RTLs are implemented in EDA Playground [6]. The code is compiled and executed on EDA playground and verified for both parameters of benchmarking.

Table 1: Experimental setup

Hardware	Quadro RTX 6000 GPU	4608 CUDA cores + 24GB of GDDR6
Software	LLaVa LLM	llava-hf/llava-v1.6-mistral-7b-hf
	DeepSeek LLM	deepseek-ai/deepseek-coder-6.7b-instruct
Verification	EDA Playground	https://edaplayground.com/

For the purpose of explanation in this paper, we consider a simple binary adder circuit as represented in figure 2.

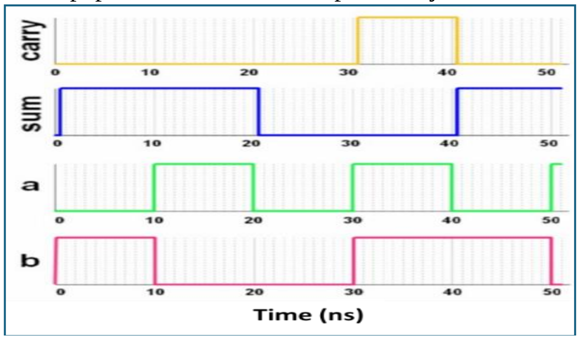
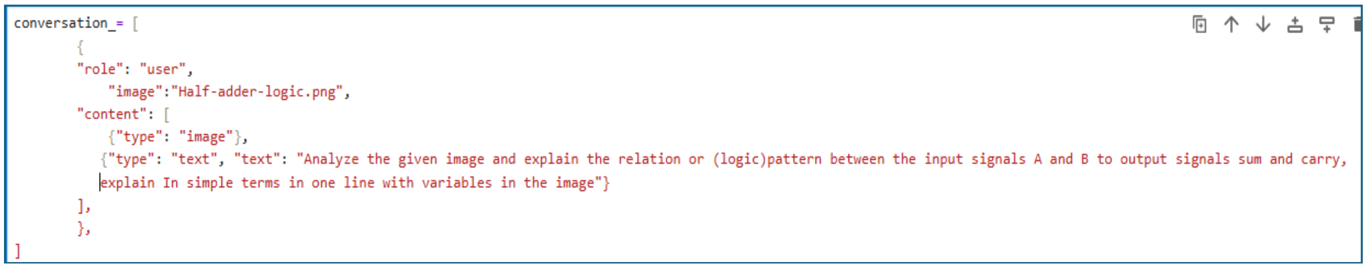


Figure 2: Input timing diagram

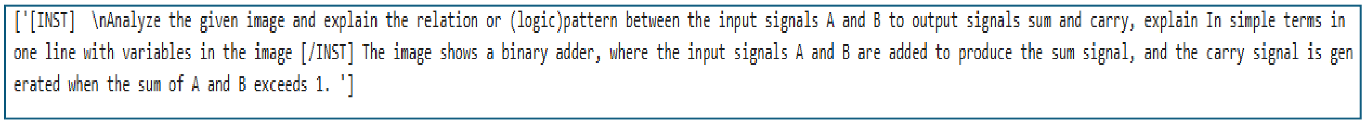
V. Results

When the binary adder timing diagram (figure 2) is input to LLaVa along with the prompt (represented in figure 3) to extract the information out of it, we get an output as represented in figure 4.



```
conversation_= [
        {
        "role": "user",
            "image":"Half-adder-logic.png",
        "content": [
            {"type": "image"},
           {"type": "text", "text": "Analyze the given image and explain the relation or (logic)pattern between the input signals A and B to output signals sum and carry,
           explain In simple terms in one line with variables in the image"}
        ],
        },
]
```

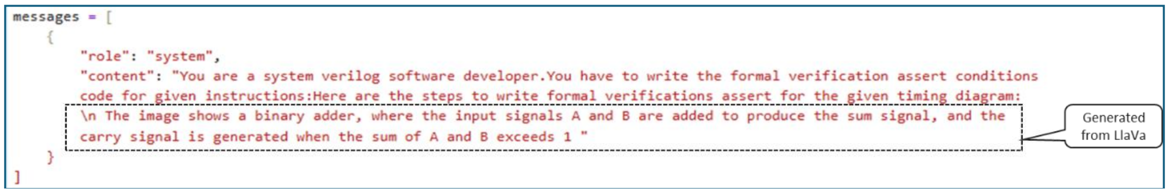
Figure 3: Prompt for LLaVA



```
['[INST]  \nAnalyze the given image and explain the relation or (logic)pattern between the input signals A and B to output signals sum and carry, explain In simple terms in
one line with variables in the image [/INST] The image shows a binary adder, where the input signals A and B are added to produce the sum signal, and the carry signal is gen
erated when the sum of A and B exceeds 1. ']
```

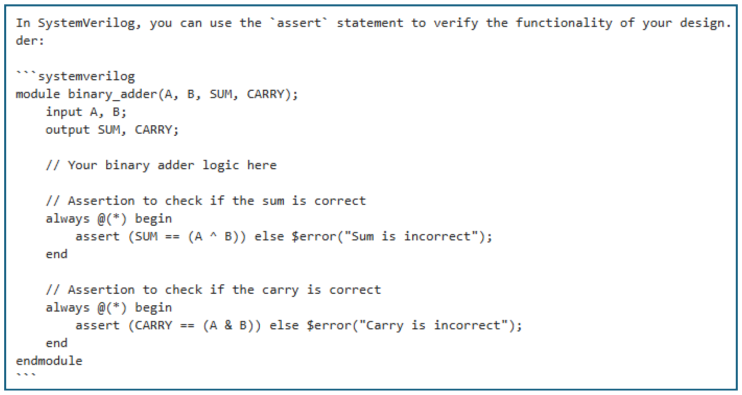
Figure 4: Output of LLaVa

This output (figure 4) becomes the input to the Deepseek along with another prompt to trigger the generation of the code: SVA (figure 5) or unit testbench (figure 7) resulting in the codes represented in figure 6 and figure 8 respectively.



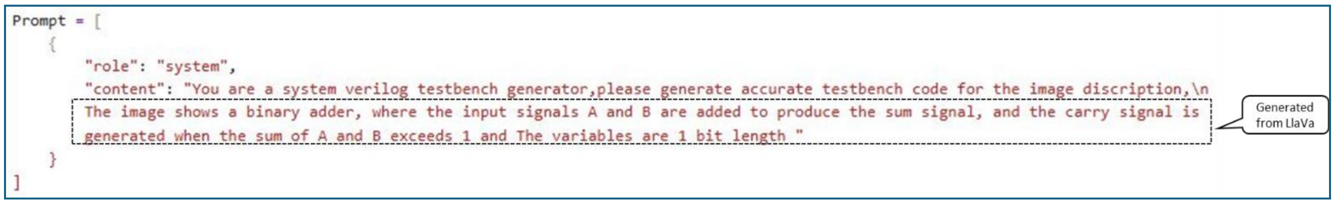
```
messages = [
    {
        "role": "system",
        "content": "You are a system verilog software developer.You have to write the formal verification assert conditions
        code for given instructions:Here are the steps to write formal verifications assert for the given timing diagram:
        \n The image shows a binary adder, where the input signals A and B are added to produce the sum signal, and the
        carry signal is generated when the sum of A and B exceeds 1 "
    }
]
```

Figure 5: Prompt for Deepseek (SVA code)



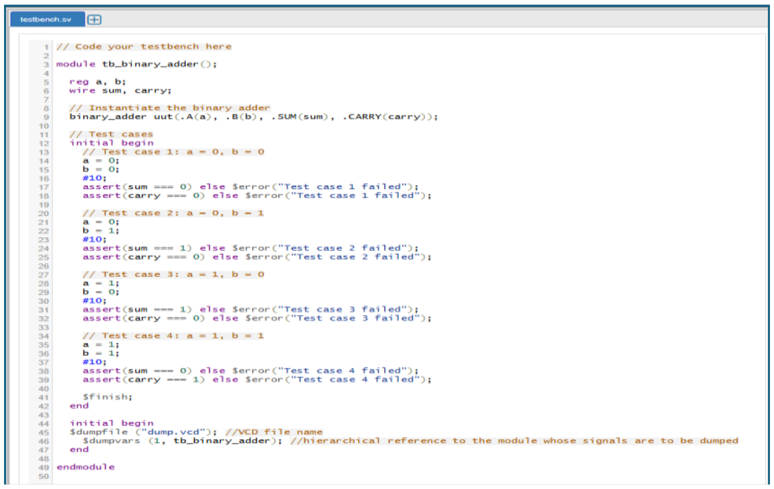
~~~
In SystemVerilog, you can use the `assert` statement to verify the functionality of your design.
der:

```systemverilog
module binary_adder(A, B, SUM, CARRY);
    input A, B;
    output SUM, CARRY;

    // Your binary adder logic here

    // Assertion to check if the sum is correct
    always @(*) begin
        assert (SUM == (A ^ B)) else $error("Sum is incorrect");
    end

    // Assertion to check if the carry is correct
    always @(*) begin
        assert (CARRY == (A & B)) else $error("Carry is incorrect");
    end
endmodule
```
~~~

Figure 6: Output of Deepseek (SVA code)



```
Prompt = [
    {
        "role": "system",
        "content": "You are a system verilog testbench generator,please generate accurate testbench code for the image discription,\n
        The image shows a binary adder, where the input signals A and B are added to produce the sum signal, and the carry signal is
        generated when the sum of A and B exceeds 1 and The variables are 1 bit length "
    }
]
```

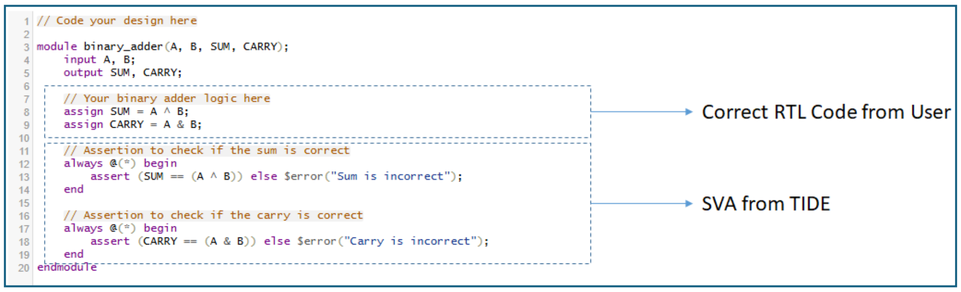
Figure 7: Prompt for Deepseek (Unit testbench code)



```
// Code your testbench here

module tb_binary_adder();

  reg a, b;
  wire sum, carry;

  // Instantiate the binary adder
  binary_adder uut(.A(a), .B(b), .SUM(sum), .CARRY(carry));

  // Test cases
  initial begin
    // Test case 1: a = 0, b = 0
    a = 0;
    b = 0;
    #10;
    assert(sum === 0) else $error("Test case 1 failed");
    assert(carry === 0) else $error("Test case 1 failed");

    // Test case 2: a = 0, b = 1
    a = 0;
    b = 1;
    #10;
    assert(sum === 1) else $error("Test case 2 failed");
    assert(carry === 0) else $error("Test case 2 failed");

    // Test case 3: a = 1, b = 0
    a = 1;
    b = 0;
    #10;
    assert(sum === 1) else $error("Test case 3 failed");
    assert(carry === 0) else $error("Test case 3 failed");

    // Test case 4: a = 1, b = 1
    a = 1;
    b = 1;
    #10;
    assert(sum === 0) else $error("Test case 4 failed");
    assert(carry === 1) else $error("Test case 4 failed");

    $finish;
  end

  initial begin
  $dumpfile ("dump.vcd"); //VCD file name
    $dumpvars (1, tb_binary_adder); //hierarchical reference to the module whose signals are to be dumped
  end

endmodule
```

Figure 8: Output of Deepseek (Unit testbench code)

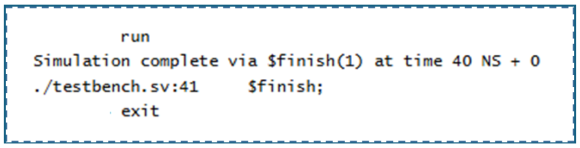
To verify whether the generated code is syntactically and functionally correct, an RTL is programmed for binary adder as represented in figure 9. It is appended by the SVA code generated by TiDe.



```
// Code your design here

module binary_adder(A, B, SUM, CARRY);
    input A, B;
    output SUM, CARRY;

    // Your binary adder logic here
    assign SUM = A ^ B;
    assign CARRY = A & B;

    // Assertion to check if the sum is correct
    always @(*) begin
        assert (SUM == (A ^ B)) else $error("Sum is incorrect");
    end

    // Assertion to check if the carry is correct
    always @(*) begin
        assert (CARRY == (A & B)) else $error("Carry is incorrect");
    end
endmodule
```

Figure 9: Execution code : RTL + SVA

When the testcases were executed on the RTL + SVA, the output log (figure 10a) and waveform (figure 10b) were generated which were exactly as per expectation.



```
           run
Simulation complete via $finish(1) at time 40 NS + 0
./testbench.sv:41      $finish;
           exit
```

Figure 10a: Output log

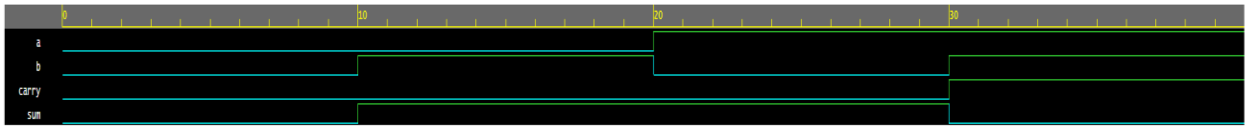
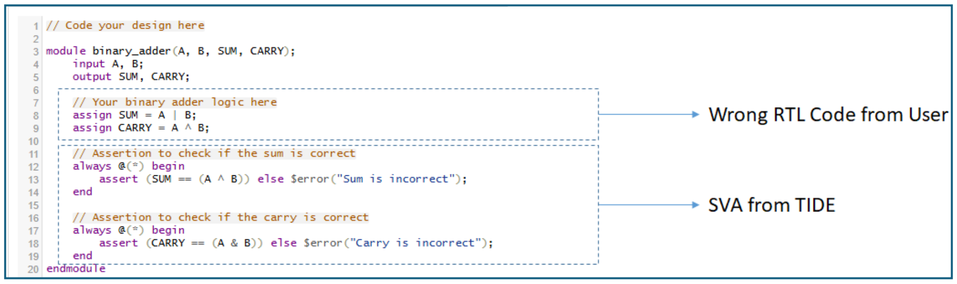


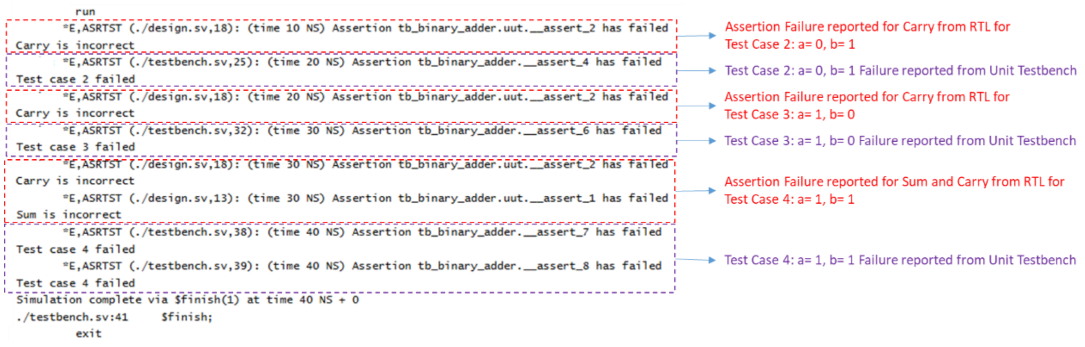
Figure 10b: Output waveform

To verify negative scenario, error injection has been done as represented in figure 11 and the result of the test execution is captured in figures 12a and 12b respectively.



```
// Code your design here

module binary_adder(A, B, SUM, CARRY);
    input A, B;
    output SUM, CARRY;

    // Your binary adder logic here
    assign SUM = A | B;
    assign CARRY = A ^ B;

    // Assertion to check if the sum is correct
    always @(*) begin
        assert (SUM == (A ^ B)) else $error("Sum is incorrect");
    end

    // Assertion to check if the carry is correct
    always @(*) begin
        assert (CARRY == (A & B)) else $error("Carry is incorrect");
    end
endmodule
```

Figure 11: Fault-injected code



```
        run
        *E,ASRTST (./design.sv,18): (time 10 NS) Assertion tb_binary_adder.uut.__assert_2 has failed
Carry is incorrect
        *E,ASRTST (./testbench.sv,25): (time 20 NS) Assertion tb_binary_adder.__assert_4 has failed
Test case 2 failed
        *E,ASRTST (./design.sv,18): (time 20 NS) Assertion tb_binary_adder.uut.__assert_2 has failed
Carry is incorrect
        *E,ASRTST (./testbench.sv,32): (time 30 NS) Assertion tb_binary_adder.__assert_6 has failed
Test case 3 failed
        *E,ASRTST (./design.sv,18): (time 30 NS) Assertion tb_binary_adder.uut.__assert_2 has failed
Carry is incorrect
        *E,ASRTST (./design.sv,13): (time 30 NS) Assertion tb_binary_adder.uut.__assert_1 has failed
Sum is incorrect
        *E,ASRTST (./testbench.sv,38): (time 40 NS) Assertion tb_binary_adder.__assert_7 has failed
Test case 4 failed
        *E,ASRTST (./testbench.sv,39): (time 40 NS) Assertion tb_binary_adder.__assert_8 has failed
Test case 4 failed
Simulation complete via $finish(1) at time 40 NS + 0
./testbench.sv:41     $finish;
        exit
```

Assertion Failure reported for Carry from RTL for Test Case 2: a= 0, b= 1

Test Case 2: a= 0, b= 1 Failure reported from Unit Testbench

Assertion Failure reported for Carry from RTL for Test Case 3: a= 1, b= 0

Test Case 3: a= 1, b= 0 Failure reported from Unit Testbench

Assertion Failure reported for Sum and Carry from RTL for Test Case 4: a= 1, b= 1

Test Case 4: a= 1, b= 1 Failure reported from Unit Testbench

Figure 12a: Output log

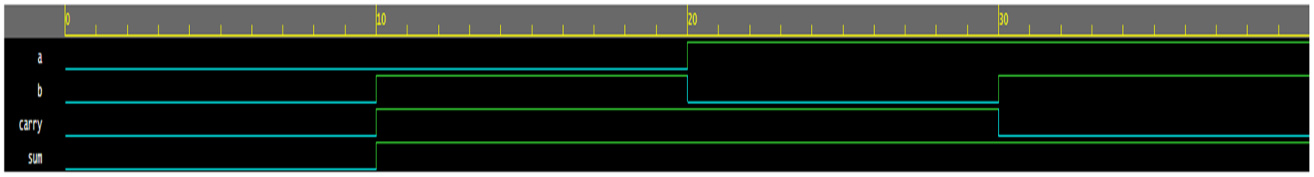


Figure 12b: Output waveform

TiDe has been tested with over 20 timing diagrams from various specifications as inputs. These timing diagrams vary in complexity w.r.t number of pins, number of delay parameters, dependency of the signals on each other, labels, difference in timescale between pins etc.
For diagrams with less complexity, TiDe was found to be highly accurate i.e, the unit testbench and SVA code were as per expectation. For diagrams with higher complexity, the output prompt of LLaVa was found to be inaccurate. The deviation in its output prompt was approximately 10%-30% from the expected prompt. This was solved by manual intervention. When this prompt was fixed, Deepseek was able to generate the code perfectly. To eliminate this manual intervention. fine-tuning of LLaVa is being carried out. This has been determined to result in more accurate code.

VI. Conclusion

As TiDe can be used to generate testbench from timing diagram, it can be highly useful for Design engineers to do unit testing and Formal verification engineers to do functionality verification. On consultation with design experts and verification experts and assortment of the data collected during an RTL project development life cycle, this has been found to reduce unit testing effort for design engineers by upto 80% and atleast 60% for verification engineers. As TiDe expects only the timing diagram from the spec as input, it can be used in commercial systems where code confidentiality and integrity are top priorities. As unit testbench development and verification testbench development processes can be parallelized, TiDe can help save the time on overall project development life cycle as well.
But the most important factor is to ensure maximum possible clarity of the signals/pins, timing parameters given in the timing diagram. The accuracy of the testcases depends upon the diagram. With the present level of training and fine-tuning, TiDe is able to convert timing diagrams with relatively less complexity. Hence, it can be considered a pilot deployment.

VII. Scope for future work

TiDe has to be polished and fine-tuned to be more effective on complicated timing diagrams. These cover scenarios such as higher number of IO pins, missing significant titles/tags/labels in the diagram, lack of information related to delay parameters etc. Along with timing diagrams, architecture diagrams and textual descriptions of the figures can be input and used to effectively generate the SVA for a better formal verification engine.

VIII. References

1. https://en.wikipedia.org/wiki/Functional_verification
2. https://en.wikipedia.org/wiki/Formal_verification#:~:text=Formal%20verification%20can%20be%20helpful%20in%20proving,as%20source%20code%20in%20a%20programming%20language
3. M. Abrahams, J. Barkley , “RTL verification strategies”, Wescon/98 IEEE Conference
4. https://huggingface.co/llava-hf/llava-v1.6-mistral-7b-hf
5. https://huggingface.co/deepseek-ai/deepseek-coder-6.7b-instruct
6. https://edaplayground.com/
7. Yao Lu et al, “RTLLM: An Open-Source Benchmark for Design RTL Generation with Large Language Model” 29th Asia and South Pacific Design Automation Conference (ASP-DAC) 2024

TABLE I
TYPE SIZES FOR CAMERA-READY PAPERS

Type size (pts.)	Appearance		
	Regular	Bold	Italic
6	Table captions,[a] table superscripts		
8	Section titles, [a] references, tables, table names,[a] first letters in table captions,[a] figure captions, footnotes, text subscripts, and superscripts		
9		Abstract	
10	Authors' affiliations, main text, equations, first letters in section titles[a]		Subheading
11	Authors' names		
24	Paper title		

[a]Uppercase

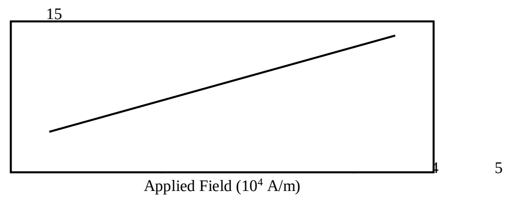


Figure 1. Magnetization as a function of applied field.
Note how the caption is centered in the column.

C. References

Number citations consecutively in square brackets [1]. Punctuation follows the bracket [2]. Refer simply to the reference number, as in [3]. Use "Ref. [3]" or Reference [3]" at the beginning of a sentence: "Reference [3] was the first …"

Number footnotes separately in superscripts. Place the actual footnote at the bottom of the column in which it was cited. Do not put footnotes in the reference list. Use letters for table footnotes (see Table I). *IEEE Transactions* no longer use a journal prefix before the volume number. For example, use "IEEE *Trans. Magn*., vol. 25," not "vol. MAG-25.

Give all authors' names; use "et al." if there are six authors or more. Papers that have not been published, even if they have been submitted for publication, should be cited as "unpublished" [4]. Papers that have been accepted for publication should be cited as "in press" [5]. In a paper title, capitalize the first word and all other words except for conjunctions, prepositions less than seven letters, and prepositional phrases.

For papers published in translated journals, first give the English citation, then the original foreign-language citation [6].

D. Abbreviations and Acronyms

Define abbreviations and acronyms the first time they are used in the text, even if they have been defined in the abstract. Abbreviations such as IEEE, SI, MKS, CGS, sc, dc, and rms do not have to be defined. Do not use abbreviations in the title unless they are unavoidable.

E. Equations

Number equations consecutively with equation numbers in parentheses flush with the right margin, as in (1). To make your equations more compact, you may use the solidus (/), the exp function, or appropriate exponents. Italicize Roman symbols for quantities and variables, but not Greek symbols. Use an en dash (–) rather than a hyphen for a minus sign. Use parentheses to avoid ambiguities in denominators. Punctuate equations with commas or periods when they are part of a sentence, as in

$$a + b = c. \tag{1}$$

Symbols in your equation should be defined before the equation appears or immediately following. Use "(1)," not "Eq. (1)" or "equation (1)," except at the beginning of a sentence: "Equation (1) is …"

Other Recommendations

The Roman numerals used to number the section headings are optional. If you do use them, do not number ACKNOWLEDGMENT and REFERENCES, and begin Subheadings with letters. Use two spaces after periods (full stops). Hyphenate complex modifiers: "zero-field-cooled magnetization." Avoid dangling participles, such as, "Using (1), the potential was calculated." Write instead, "The potential was calculated using (1)," or "Using (1), we calculated the potential."

Use a zero before decimal points: "0.25," not ".25." Use "cm^3," not "cc." Do not mix complete spellings and abbreviations of units: "Wb/m^2" or "webers per square meter," not "webers/m^2." Spell units when they appear in text: "…a few henries," not "…a few H." If your native language is not English, try to get a native English-speaking colleague to proofread your paper. Do not add page numbers.

III. UNITS

Use either SI (MKS) or CGS as primary units. (SI units are encouraged.) English units may be used as secondary units (in parentheses). An exception would be the use of English units as identifiers in trade, such as "3.5-inch disk drive."

Avoid combining SI and CGS units, such as current in amperes and magnetic field in oersteds. This often leads to confusion because equations do not balance dimensionally. If you must use mixed units, clearly state the units for each quantity that you use in an equation.

IV. SOME COMMON MISTAKES

The word "data" is plural, not singular. The subscript for the permeability of $vacuum_0$ is zero, not a lowercase letter "o." In American English, periods and commas are within quotation marks, like "this period." A parenthetical statement at the end of a sentence is punctuated outside of the closing parenthesis (like this). (A parenthetical *sentence* is punctuated within the parentheses.) A graph within a graph is an "inset," not an "insert." The word alternatively is preferred to the word "alternately" (unless you mean something that alternates). Do not use the word "essentially" to mean "approximately" or "effectively." Be aware of the different meanings of the homophones "affect" and "effect," "complement" and "compliment," "discreet" and "discrete," "principal" and "principle." Do not confuse "imply" and "infer." The prefix "non" is not a word; it should be joined to the word it modifies, usually without a hyphen. There is no period after the "et" in the Latin abbreviation "et al." The abbreviation "i.e." means "that is," and the abbreviation "e.g." means "for example." An excellent style manual for science writers is [7].

ACKNOWLEDGMENT

The preferred spelling of the word "acknowledgment" in America is without an "e" after the "g." Try to avoid the stilted expression, "One of us (R. B. G.) thanks …" Instead, try "R.B.G. thanks …" Put sponsor acknowledgments in the unnumbered footnote on the first page.

REFERENCES

[1] G. Eason, B. Noble, and I.N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529-551, April 1955.
[2] J. Clerk Maxwell, *A Treatise on Electricity and Magnetism,* 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68-73.
[3] I.S. Jacobs and C.P. Bean, "Fine particles, thin films and exchange anisotropy," in *Magnetism,* vol. III, G.T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271-350.
[4] K. Elissa, "Title of paper if known," unpublished.
[5] R. Nicole, "Title of paper with only first word capitalized," *J. Name Stand. Abbrev.,* in press.
[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," *IEEE Transl. J. Magn. Japan,* vol. 2, pp. 740-741, August 1987 [*Digests 9th Annual Conf. Magnetics Japan,* p. 301, 1982].
[7] M. Young, *The Technical Writer's Handbook.* Mill Valley, CA: University Science, 1989.