Artificial Incorrectness: SMT and LLMs in Hardware Synthesis

Edward Wang[1 [0000−0001−9404−2954]], Joe Walston[2 [0009−0003−9697−9144]], Luca Daniel[1 [0000−0002−5880−3151]], Tony Tan[3 [0009−0005−8341−2004]], Yoni Zohar[4 [0000−0002−2972−6695]], and Clark Barrett[5 [0000−0002−9522−3084]]

[1] Massachusetts Institute of Technology, Cambridge, MA, USA
[2] Synopsys, Inc., Morrisville, NC, USA
[3] University of Liverpool, Liverpool, UK
[4] Bar-Ilan University, Ramat Gan, Israel
[5] Stanford University, Stanford, CA, USA
edwardw@csail.mit.edu

Abstract. The adoption of large language models (LLMs) in hardware design automation poses correctness risks for safety-critical applications. We systematically evaluate LLMs against Satisfiability Modulo Theories (SMT) solvers across three hardware synthesis tasks, revealing that LLMs achieve lower levels of functional correctness compared to SMT approaches in our benchmarks. Our findings reveal a crucial distinction: whilst SMT solvers can excel at direct synthesis and can exhaustively validate LLM outputs, their counterexample feedback fails to improve LLM performance. This demonstrates that effective validation does not translate to effective improvement guidance for LLMs, establishing formal methods as essential for direct synthesis and a need for better iterative refinement methods in reliable AI-assisted hardware design.

Keywords: Synthesis · Satisfiability Modulo Theories · Electronic Design Automation · Large Language Models · Artificial Intelligence · Formal Verification.

1 Introduction

Large language models (LLMs) promise to revolutionise hardware design automation, yet evidence suggests their reliability remains concerning [70]. Despite impressive capabilities in generating designs from natural language specifications, studies indicate that LLMs frequently produce subtle functional errors that can trigger catastrophic system failures [42, 69]. Such errors may evade initial testing yet emerge later as critical failures in production hardware.

Hardware design demands high reliability [28, 11], as functional errors can incur devastating costs. Silicon respins (re-designing and re-fabricating chips to fix errors) entail millions of dollars and months of delay [9, 61]. Though recent work has explored enhancing traditional formal synthesis methods like [6] with LLMs [35], current LLM-based approaches typically trade correctness guarantees

for natural language convenience [70, 66, 7]. Context window limitations further compound these challenges when handling larger, complex systems [21, 36, 58]. As a consequence, this reliability gap motivates systematic evaluation against established formal methods such as SMT (Satisfiability Modulo Theories) [12].

SMT solvers extend Boolean satisfiability (SAT) with theories including integer arithmetic, bit-vectors, arrays, and uninterpreted functions [12]. This evolution from pure Boolean SAT was partially driven by hardware verification requirements to more conveniently reason about higher-order structures such as integers and bit-vectors [30]. Through decades of research, modern solvers such as z3 [43], cvc5 [10], Bitwuzla [46], MathSAT [15], and Yices [19] have matured to find extensive application in hardware verification, software verification, and synthesis. In contemporary hardware contexts, these solvers have been effective for bounded model checking, equivalence checking, and synthesis [20, 30].

Our evaluation methodology differs from specification synthesis work by assuming designers provide both natural-language descriptions and formal specifications - enabling rigorous comparative evaluation of synthesis approaches. This reflects realistic industrial scenarios where designers specify formal correctness criteria but need implementation assistance. Natural-language descriptions guide LLM synthesis by capturing architectural constraints, optimisation preferences, and design patterns that resist formal specification. Formal specifications enable rigorous verification of LLM-generated implementations. This dual-input methodology aligns with intended requirements whilst delivering exhaustive correctness guarantees unavailable in existing LLM-based hardware synthesis approaches, as detailed in our related work comparison in Section 2.

On our benchmarks, SMT solvers [12] demonstrate distinct capabilities in two roles: as direct synthesis engines and as exhaustive validators of LLM outputs. However, our experiments reveal that validation does not equate to improvement - despite providing concrete counterexamples, SMT feedback failed to improve LLM correctness. Our evaluation encompasses three synthesis challenges: Arithmetic Logic Unit (ALU) microcode generation, Field-Programmable Gate Array (FPGA) function mapping, and protocol converter design. Results demonstrate that, on these benchmarks, formal encoding and contract-driven synthesis approaches [13, 24] outperform both direct LLM generation and iterative LLM refinement, suggesting that formal methods could be used for direct synthesis rather than as merely feedback sources to LLMs.

2 Related Work

LLMs have been recently used in various tasks including RTL code summarisation, generation, evaluation, debugging, and test vector generation [66, 7]. Specification synthesis, creating a formal specification from natural language or other stimuli, is another recent trend in applying LLMs in hardware design [48, 39].

On the other hand, classical design space exploration (DSE) methods involve systematically exploring numerous design options to identify optimal configurations [67, 55, 64]. These approaches typically evaluate trade-offs across multiple

objectives such as power, area, performance, and timing constraints. DSE methods can employ exhaustive enumeration, heuristic search algorithms, or machine learning-guided optimisation to navigate the exponentially large space of possible design choices. Optimisation-based methods use mathematical approaches to iteratively refine design decisions [38, 18]. While these classical approaches are considered more mature, recent LLM-based techniques face more fundamental challenges in correctness assurance.

Recent work on LLM-based hardware generation focuses on improving LLM performance through iterative refinement, but lacks rigorous comparison against established formal synthesis methods. Current verification approaches demonstrate significant limitations. Tang et al. [60] verify LLM-generated RTL using simulation-based testbenches, testing only 10–20 example inputs that cover merely 7.8% of the possible 256 input combinations in our 8-bit benchmarks. Such partial verification misses systematic errors: in our Mux function case study (Section 4.1), incorrect LLM solutions produce correct outputs for roughly 30% of inputs purely by coincidence whilst failing on the remaining 70%. Akyash et al. [4] rely solely on syntactic verification through Yosys compilation, which cannot detect functional errors like our ALU case study (Section 4.2) where syntactically valid microcode computes $f(x) = x - 1$ instead of the required $f(x) = x(x - 1)$.

Nonetheless, many current approaches assume that iterative feedback can compensate for these verification gaps. Despite the demonstrated inadequacy of current verification methods, the field operates under the assumption that feedback-driven approaches can reliably improve LLM synthesis performance. This assumption drives much current research effort. [51] develops discriminative guidance mechanisms for iterative Verilog refinement, whilst [68] proposes comprehensive multi-agent frameworks that employ feedback loops for automated processor design. Similarly, [23] integrates human-in-the-loop verification with agentic AI systems, assuming that iterative feedback enhances design quality. Related efforts extend this feedback paradigm to specialised domains: [22] evaluates LLM-generated designs for security vulnerabilities with the goal of improving LLM security awareness, [56] explores LLM potential for formal specification generation through iterative refinement, and [5] surveys feedback-based approaches across various hardware design applications.

A survey of over 360 papers by Abdollahi et al. [2] reveals that iterative, feedback-driven prompting - rather than re-training or fine-tuning - is the dominant approach for applying LLMs to hardware design and verification. Yet, across the 70 directly relevant works covered, the core assumption that prompting an LLM with error-based feedback improves its output is treated as conventional wisdom rather than an empirically tested hypothesis. Verification, where it exists, relies primarily on simulations that sample only a fraction of possible inputs.

To address this unanswered question, our work provides one of the first head-to-head comparisons of LLMs versus SMT solvers on identical synthesis tasks, offering both a new empirical vantage point and resolution of this fundamental assumption. We discover a fundamental limitation: whilst SMT provides exhaus-

tive validation of LLM outputs, SMT counterexample feedback fails to improve LLM performance (0% success in Function Mapping, 1.25% in ALU tasks). This validation-improvement distinction reveals that effective correctness checking does not translate to effective iterative refinement: a finding that challenges the purely-LLM feedback-driven approaches dominating current research. Our results work towards establishing empirical design principles for when to use formal methods directly versus as LLM assistance tools, addressing a core question that existing literature sidesteps.

In contrast to approaches relying on partial validation, our study employs SMT-based [12] formal encodings which provide mathematical assurances of functional correctness. SMT exhaustively checks all possible inputs simultaneously (for instance, all 256 possible 8-bit input combinations) rather than sampling subsets. This complete coverage reveals systematic errors that partial testing misses: whilst simulation might test 20 cases and miss failures on the remaining 236, SMT either finds a provably correct solution or proves no solution exists within the given constraints. However, our experiments reveal a key limitation: whilst SMT excels at validation (determining correctness), this does not translate to effective improvement guidance for LLMs, which cannot map concrete counterexamples to parameter corrections.

3 LLM and Prompting Methodology

Our prompt engineering approach is guided by two principles: clarity and reproducibility. We ensured that each prompt contained all the information a competent human designer would require and maintained consistency of prompts across experimental runs and case studies to enable unbiased comparisons. All prompts were pre-recorded, and the OpenAI API was used deterministically to guarantee reproducibility.

We did not employ chain-of-thought [63], self-consistency sampling [63], or majority voting [3] in our approach; these remain opportunities for future work.

3.1 Hyperparameters and Configuration

All LLM queries used OpenAI's `gpt-4o-2024-11-20` without fine-tuning. Whilst newer models have emerged since our November 2024 experiments, we maintain our original results to ensure reproducibility and as our findings demonstrate systematic challenges that likely persist across model versions. We used the `text` response format, no temperature adjustment (`temperature=1`, `top_p=1`), maximum token count of `16384`, and no penalty adjustments (`frequency_penalty=0`, `presence_penalty=0`).

3.2 Prompt Development and Templates

Our prompt development process required significant manual effort, with 10-20 hours per case study. This involved iterative testing typically across 10-20 trial

runs. This extensive effort focused on formulating clear, comprehensive English-language descriptions tailored for a designer-level audience, ensuring the LLM properly understood both the design context and specific requirements for each synthesis task.

The prompts follow a structured two-part design to maintain consistency whilst enabling flexibility. We hand-crafted a system prompt [45] in Markdown, offering a comprehensive description of the system and problem for a human designer. Specific task instructions (also in Markdown) were placed in the user prompt, with each description containing a placeholder[6] replaced at runtime with the precise objective as needed. This separation allows for consistent system-level context whilst enabling flexible task-specific customisation.

To illustrate this approach, we provide key excerpts from the Generalised ALU case study below:

System prompt excerpt:

```
We are designing microcode to compute one-input functions `y =
↪  f(x)` for an 8-bit, 2 register ALU [...]
* You may use AT MOST 7 instructions, as that is the size of this
↪  ALU's instruction space.
* Provide ONLY lines in the above format.
* Do NOT provide any explanations.
```

User prompt excerpt:

```
Your task is to implement the floored square root function `f(x) =
↪  floor(sqrt(x))`. We should make it INSERT_OBJECTIVE_HERE.
```

3.3 In-Context Learning

We used in-context learning [50] to improve LLM performance. For our design tasks, three key components were provided: a system/architecture description, the target function, and an example architecture mapping. The example architecture mapping is to illustrate the functioning of the system and is not necessarily example solutions to specific tasks being attempted.

For example, for the Function Mapping task in Section 4.1, we provide an example architecture mapping:

```
lut0_0 = 65511; lut0_1 = 45168
lut0_2 = 28416; lut0_3 = 512
lut0_4 = 45629; lut1_0 = 15391
lut1_1 = 15391; lutF = 2
```

We also provide multiple input-output examples of the target function. For instance, these examples illustrate a comparator function that checks whether the first input is greater than or equal to the second input:

[6] `INSERT_OBJECTIVE_HERE`

```
* `input1 = 5` (0101), `input0 = 0` => `output = 1`
* `input1 = 5` (0101), `input0 = 1` => `output = 0`
* `input1 = 5` (0101), `input0 = 2` => `output = 1`
* `input1 = 15` (1111), `input0 = 3` => `output = 1`
```

3.4 Iteration, Feedback, and Tool Use

We use an iterative refinement process where each LLM-generated solution undergoes verification, and the verification results guide subsequent iterations. The process operates as follows: (1) present the LLM with the initial prompt, (2) obtain a candidate solution, (3) verify the solution using an appropriate verification tool (SMT, Chisel testbench, and/or Yosys), (4) provide specific feedback from the verification tool to the LLM, and (5) repeat until either a correct solution is found or 20 iterations are reached.

We limited experiments to 20 iterations per test case, as LLMs beyond this point produced redundant candidates or increasingly complex solutions without meaningful progress.

For the SMT-based experiments (Function Mapping and Generalised ALU), the verification feedback includes:

- **Correctness status**: Whether the solution is functionally correct
- **Counterexamples**: When incorrect, specific input values that produce wrong outputs
- **Size/formatting errors**: When bit-widths or value ranges are violated

The raw output of the verification tool replaced the `TOOL_FEEDBACK_HERE` placeholder.

```
This module is not INSERT_OBJECTIVE_HERE. Please do better.

Tool feedback:
TOOL_FEEDBACK_HERE
```

For experiments without tool feedback, the "Tool feedback" section is omitted.

3.5 SMT as Validator vs. Improvement Mechanism

Our experiments reveal a critical distinction between SMT's dual roles in hardware synthesis:

SMT as Direct Synthesiser: SMT solvers achieve reasonable success in our benchmarks, either finding provably correct solutions or proving infeasibility (UNSAT) within reasonable runtime constraints.

SMT as Validator in LLM Feedback Loop: Despite providing exhaustive counter-example feedback to LLMs, the SMT-provided information yielded near-zero improvement rates: 0% (0/120) in Function Mapping tasks and 1.25% (2/160) in ALU tasks.

This stark disparity reveals that validation - determining whether a solution is correct - differs from improvement guidance (helping an LLM correct its errors). LLMs may possibly lack the constraint propagation mechanisms necessary to map concrete counterexamples (e.g., "input (5,2) produces wrong output") back to abstract configuration corrections (e.g., "modify LUT values X, Y, Z"). Whilst SMT can exhaustively verify all possible inputs and identify failures, this validation feedback proves ineffective for direct generative improvement.

4 Evaluation

We compare LLMs and SMT solvers on the Generalised ALU and Function Mapping tasks, where SMT achieves guaranteed correctness while LLMs consistently fail (Sections 4.1 and 4.2). Section 4.3 shows that LLMs can perform better with extensive guidance and verification, but still need substantial support.

Our evaluation focuses on the synthesis stages of the HDL-to-circuit compilation flow - from architectural configuration and microcode generation through logic synthesis - rather than backend physical implementation stages such as place-and-route.

Unless otherwise stated, all experiments were conducted on an Intel i7-1270P machine with 32GB RAM, along with a 20-minute timeout. The timeout strikes a practical balance, allowing sufficient time for solvers to make progress on hard synthesis problems whilst keeping experiment duration reasonable.

We used `cvc5` [10][7] amongst modern SMT solvers (including z3 [43], Bitwuzla [46], and MathSAT [15]) due to its comprehensive support for quantifier-free theories relevant to hardware synthesis: fixed-size bit-vectors, arrays, and uninterpreted functions. The `ALL` logic configuration enables cvc5's full theory combination capabilities, essential for encoding our synthesis constraints that mix Boolean logic with bit-vector arithmetic [12, 10].

Where RTL is present, we use Chisel [8] for RTL compilation and elaboration, the Mill build tool [34] for builds, ChiselTest [54] for throughput testbench simulation, and Yosys [65] for post-synthesis gate counts.

4.1 Function Mapping

In this case study, we evaluate whether LLMs can synthesise correct configurations for an FPGA/CGRA-like device that computes 4-bit functions using a layered lookup table (LUT) architecture.

Intuition: The device comprises programmable LUTs (truth tables) chained through a fixed two-layer topology. This architectural constraint renders certain functions implementable whilst others remain fundamentally unrealisable, irrespective of LUT programming. The synthesis task requires either: (1) determining LUT configurations that implement a target function, or (2) proving the architecture cannot express it.

[7] Version 1.0.8-git (revision `cef9d27`)

LUT Semantics: Each LUT is a programmable truth table indexed by input value. An LUT with value v implements f(i) = bit i of v. For example, LUT value 5 (binary 0101) implements f(0)=1, f(1)=0, f(2)=1, f(3)=0. The fixed interconnect topology between LUTs constrains which composite functions the circuit can express.

Device Specification: The device accepts two 4-bit inputs (`input0`, `input1`) and produces a single 1-bit output. LUTs are 4-bit (16-entry) tables programmed via 16-bit unsigned integers. For instance, LUT value 514 (binary 0000001000000010) encodes a specific input-output mapping.

We combine `input0` and `input1` into a single `input` 8-bit bit-vector as {`input1, input0`} (concatenation), or expressed otherwise: `input[3:0] = input0` (bits 3 through 0) and `input[7:4] = input1` (bits 7 through 4).

We then have a first layer of LUTs as outlined below:

- LUT `lut0_0` implements $f($`input`$[3:0]) =$ `imm0`$[0]$
- LUT `lut0_1` implements $f($`input`$[4:1]) =$ `imm0`$[1]$
- LUT `lut0_2` implements $f($`input`$[5:2]) =$ `imm0`$[2]$
- LUT `lut0_3` implements $f($`input`$[6:3]) =$ `imm0`$[3]$
- LUT `lut0_4` implements $f($`input`$[7:4]) =$ `imm0`$[4]$

Following this, we have a second layer of LUTs:

- LUT `lut1_0` implements $f($`imm0`$[3:0]) =$ `imm1`$[0]$
- LUT `lut1_1` implements $f($`imm0`$[4:1]) =$ `imm1`$[1]$

Finally, a single 2-bit LUT:

- LUT `lutF` implements $f($`imm1`$[1:0]) =$ `output`

Each LUT is defined by the following LUT relationship `lutRelation(lut, input, output)`:[8]:

```
output = (lutValue << concat(zero, input))[0:0]
```

The device function is defined as `circuitRelation(input0, input1, output)`, shown in Figure 1.

Finally, we assert that the LUT values chosen are correct for some desired function $f(x, y)$ as follows:

$$\forall x \in \mathbb{B}^N, \forall y \in \mathbb{B}^N, \text{circuitRelation}(x, y, f(x, y))$$

This encoding leverages SMT's ability to reason directly about bit-vectors and their operations, avoiding the manual Boolean encoding required by pure SAT approaches [12].

To synthesise, we query the solver for LUT values that satisfy these constraints. To validate a candidate LLM solution, we add assertions of the form lut$K$ = L_K, where L_K is the LLM's answer for the K LUT, converting the synthesis problem into a verification query.

[8] `zero` is an appropriately sized bit-vector whose value is 0; `x[0:0]` takes the least significant bit; `<<` marks the left shift operator; `concat` denotes concatenation of two bit-vectors.

```
input = input1 || input0
lutRelation(lut0_0, input[3:0], imm0[0]) &&
lutRelation(lut0_1, input[4:1], imm0[1]) &&
[...]
lutRelation(lut0_4, input[7:4], imm0[4]) &&
lutRelation(lut1_0, imm0[3:0], imm1[0]) &&
lutRelation(lut1_1, imm0[4:1], imm1[1]) &&
lutRelation(lutF, imm1[1:0], output)
```

Fig. 1. The definition of `circuitRelation`.

4.1.1 Results We evaluate whether six different functions can be implemented using our layered LUT architecture. SMT finds working LUT configurations for three functions but shows that the remaining three cannot be implemented.

- `Comparator`: $f(x, y) = x >= y$
- `Mux`: $f(x, y) = (x >> y)\&1$ (select the yth bit of x)
- `BitInversion`: $f(x, y) = (x ==\sim y)$
- `Equality`: $f(x, y) = x == y$
- `Subset`: $f(x, y) = (x|y) == y\&1$ (if x is a bit-set subset of y)
- `UnsignedOverflow`: $f(x, y) = ((x + y) > 15)$ (if $x + y$ will overflow)

Testcase	**SMT runtime (ms)**	**SMT res.**	**Correct solns. (of 20)**	**Size errs. (of 20)**	**Logic errs. (of 20)**
Comparator	257	UNSAT	0	2	18
Mux	179	SAT	0	1	19
BitInversion	127	SAT	0	4	16
Equality	127	SAT	0	2	18
Subset	310	UNSAT	0	1	19
Unsigned Overflow	257	UNSAT	0	1	19

Table 1. Function mapping case study results. SMT columns show synthesis runtime and feasibility. LLM columns show the number of attempts (out of 20 iterations per test case) that resulted in correct solutions, bit-width/formatting violations, or functional logic errors, respectively.

Table 1 shows that across 20 iterative refinement attempts per test case, the LLM failed to produce any correct function mappings. The SMT results column indicates synthesis feasibility: `SAT` means the SMT solver found a valid implementation, whilst `UNSAT` means the solver mathematically proved[9] that no

[9] Strictly speaking, formal proof certificates (e.g., DRAT proofs [26]) would be required to verify unsatisfiability claims, but we treat solver `UNSAT` results as reliable for our evaluation purposes.

implementation exists within the given architectural constraints. Unlike incomplete search methods that might simply fail to find a solution, `UNSAT` represents an exhaustive mathematical proof of impossibility for the specific architecture. Out of 120 total attempts across all test cases, the LLM generated zero functionally correct solutions.

Interpretation and Patterns: Despite receiving specific counter-examples showing exactly which inputs produced wrong outputs, LLMs achieved 0% correctness across all test cases. The iterative feedback mechanism proved entirely ineffective: across 120 total refinement attempts (6 test cases $\times$ 20 iterations), not a single LLM-generated solution achieved functional correctness. This failure persisted despite SMT providing concrete counterexamples - for instance, informing the LLM that "input0=0, input1=3 produces output=1 instead of the required output=0". In contrast, SMT solvers either found correct solutions within milliseconds or mathematically proved that no solution exists within the architectural constraints. This stark difference reveals that LLMs fundamentally struggle with constraint propagation: they cannot translate specific input-output mismatches into the systematic LUT value adjustments required for correction.

Limitations: This evaluation covers six functions, limiting broader conclusions. Different prompting strategies or tool-assisted approaches might yield better results, though our setup produced uniformly incorrect outputs.

Qualitative Illustration: Here we present examples demonstrating the contrast between SMT-generated correct mappings and LLM-generated incorrect mappings for the `Mux` function:

1. Correct SMT-generated LUT mapping for `Mux`:

```
lut0_0 = 38233; lut0_1 = 43645; lut0_2 = 51967; lut0_3 = 252;
↪  lut0_4 = 13
lut1_0 = 16226; lut1_1 = 45774
lutF = 6
```

2. Syntactically invalid LLM-generated LUT mapping for `Mux`: This mapping violates bit-width constraints as `lut0_0` is assigned a value of 69905, which exceeds the 16-bit limit (maximum value 65535).

```
lut0_0 = 69905; lut0_1 = 54016; lut0_2 = 36880; lut0_3 = 20496;
↪  lut0_4 = 8208
lut1_0 = 65280; lut1_1 = 65280
lutF = 1
```

3. Syntactically valid but functionally incorrect LLM-generated LUT mapping for `Mux`: Whilst this mapping adheres to bit-width constraints, SMT verification reveals it produces incorrect outputs. Recall that `Mux` should implement $f(x, y) = (x >> y)\&1$, which selects the yth bit of x. The LLM-generated mapping fails to implement this logic correctly in multiple ways. For example, when `input0`=0 (i.e., $x = 0$), all bits of x are zero, so the function should return 0 regardless of the bit position selected by y (i.e., `input1`). However, this mapping incorrectly produces an output of 1 when `input0`=0 and

`input1`=3, contradicting the expected behaviour. Conversely, when `input0`=15 (i.e., $x = 15 = 1111_2$), all four bits are set to 1, so selecting any bit position should yield 1. Yet this mapping incorrectly produces an output of 0 for `input0`=15 when `input1`$\in \{1, 2\}$, failing to recognise that bits 1 and 2 of the value 15 are both set to 1. These systematic errors demonstrate the LLM's inability to correctly implement the bit-selection logic required by the `Mux` function.

```
lut0_0 = 42405; lut0_1 = 33825; lut0_2 = 25245; lut0_3 = 16665;
↪  lut0_4 = 8070
lut1_0 = 65280; lut1_1 = 65280
lutF = 1
```

4.2 Generalised ALU

In this case study, we are tasked with designing microcode to compute one-input functions $y = f(x)$ for an 8-bit, 2-register ALU for low-power, embedded, Digital Signal Processing (DSP) applications. An Arithmetic Logic Unit (ALU) performs basic arithmetic and logical operations. Our 2-register design constrains computation to two working values (`r0`, `r1`) to minimise area in embedded applications. The two registers are `r0` and `r1`. We represent the state as a pair (a, b), which means `r0` $= a \wedge$ `r1` $= b$. We consider microcode up to 7 instructions long.

Here are the available instructions:

- `INSTR_NOOP` - does nothing. $(a, b) \to (a, b)$
- `INSTR_SWAP` - swaps the two values of the registers. $(a, b) \to (b, a)$
- `INSTR_ADD` - adds the two values of the registers and stores the result in `r0`. $(a, b) \to (a + b, b)$
- `INSTR_SUB` - subtracts `r1` from `r0` and stores the result in `r0`. $(a, b) \to (a-b, b)$
- `INSTR_MUL` - multiplies the two values of the registers and stores the result in `r0`. $(a, b) \to (a \times b, b)$
- `INSTR_NEG` - negates the value in `r0`. $(a, b) \to (-a, b)$
- `INSTR_INCR` - increments the value in `r0`. $(a, b) \to (a + 1, b)$
- `INSTR_DOUBLE` - doubles the value in `r0`. $(a, b) \to (2a, b)$

The program is initialised such that x, the input, is in `r0`, while `r1` is set to 0.

We create an SMT encoding for this problem as follows.

First, we alias each instruction to an integer, e.g., `INSTR_NOOP` = 0, `INSTR_SWAP` = 1, `INSTR_ADD` = 2, etc.

Then, we define small-step semantics relating the state of each instruction to the next:

```
function InstrRelation(instr, input, output):
    if (instr = INSTR_NOOP) then output = (input.r0, input.r1)
    else if (instr = INSTR_SWAP) then output = (input.r1, input.r0)
    else if (instr = INSTR_ADD) then output = (input.r0 + input.r1,
input.r1)
```

```
5:      else if ...                                   ▷ additional instructions
6:      else return false                             ▷ if the instruction is illegal
```

Then, we define a whole-program relation as follows, by defining intermediate states and chaining the small-step semantics:

```
1: function FullRelation(input, output):
2:      InstrRelation(instr0, (input, 0), s0)
3:      and InstrRelation(instr1, s0, s1)
4:      and InstrRelation(instr2, s1, s2)
5:      ...                                           ▷ additional instructions
6:      and InstrRelation(instrN, s(N-1), sN)
7:      and output = sN.r0
```

Finally, we assert that the set of instructions chosen is correct for some desired function $f(x)$ as follows:

$$\forall x \in \mathbb{B}^N : \text{FullRelation}(x, f(x))$$

This formulation exemplifies SMT's ability to encode program synthesis as constraint satisfaction [6, 1]. The solver systematically explores the combinatorial space of instruction sequences to find valid implementations. To synthesise, we query this constraint system directly. To validate LLM solutions, we add constraints fixing the instruction values to the LLM's choices.

4.2.1 Results We define eight functions spanning from basic arithmetic to complex operations like factorial and square root to test synthesis across diverse problem domains. Using SMT, we find valid LUT configurations for four functions whilst proving no implementations exist within the architectural constraints for four others.

- `abs`: $f(x) = |x|$
- `bitinversion`: $f(x) = \sim x$ (bit inversion)
- `cubing`: $f(x) = x^3$
- `factorial`: $f(x) = x!$
- `picktwo`: $f(x) = P(x, 2) = x(x-1)$ (2-permutations of x, or the number of ways to pick 2 objects from x)
- `rrot`: $f(x) = x[0] \mid\mid x[7:1]$ (rotate rightwards by one bit)
- `sqrt`: $f(x) = \lfloor \sqrt{x} \rfloor$
- `squaring`: $f(x) = x^2$

We summarise the results of this case study in Table 2.

Interpretation and Patterns: LLMs produced syntactically valid microcode without hallucinating instructions, but consistently failed to implement target functions correctly. Across 160 attempts (8 test cases, 20 iterations), only 2 solutions achieved correctness (both for the squaring function) - yielding 1.25% success despite iterative SMT feedback.[10] The feedback mechanism proved ineffective: whilst SMT identified that a program computed $x-1$ instead of $x(x-1)$

[10] The two successful LLM solutions required more instructions than SMT's optimal sequences.

Testcase	SMT runtime (min)	SMT result	Correct solns. (of 20)	Syntactic errs. (of 20)	Logic errs. (of 20)
abs	5	UNSAT	0	0	20
bitinversion	<1	SAT	0	0	20
cubing	<1	SAT	0	0	20
factorial	1.5	UNSAT	0	0	20
picktwo	<1	SAT	0	0	20
rrot	3	UNSAT	0	0	20
sqrt	4	UNSAT	0	0	20
squaring	<1	SAT	2	0	18

Table 2. A variety of functions to implement on the ALU and their respective synthesis results and SMT runtimes (in minutes for readability; cf. Table 1 which uses milliseconds for sub-second runtimes). LLM columns show the number of attempts (out of 20 iterations per test case) that resulted in correct solutions, syntactic violations, or functional logic errors, respectively.

for `picktwo`, LLMs could not translate functional discrepancies into corrective instruction modifications. Failures included off-by-one errors and operand confusion, demonstrating that `gpt-4o` in our experiments lacks symbolic reasoning to map functional errors to instruction adjustments. Conversely, SMT consistently found correct implementations or proved impossibility (UNSAT) within minutes.

Limitations: We tested only 8 ALU functions, limiting statistical significance. Better prompting, task selection, or intermediate verification might improve outcomes, though LLMs showed both creativity and unreliability.

Qualitative Illustration: Here are example programs generated in this test case:

- *Correct LLM-generated* `squaring` *program*: `INSTR_SWAP, INSTR_ADD, INSTR_SWAP, INSTR_MUL`
- *Correct SMT-generated* `squaring` *program*: `INSTR_SWAP, INSTR_ADD, INSTR_MUL`

It is noteworthy that SMT coincidentally found a shorter program whereas the LLM-generated program is longer, even though it is also correct.

- *Correct SMT-generated* `picktwo` *program*: `INSTR_SWAP, INSTR_ADD, INSTR_MUL, INSTR_SUB`
- ***Incorrect*** *LLM-generated* `picktwo` *program*: `INSTR_SWAP, INSTR_INCR, INSTR_SWAP, INSTR_SUB, INSTR_SWAP, INSTR_MUL`

The SMT-generated program is correct-by-construction due to synthesis [20, 40, 53, 49]. To demonstrate this concretely, we now analyse both the correct SMT solution and the incorrect LLM solution for the `picktwo` function, which computes $f(x) = x(x-1)$ (the number of 2-permutations of x objects).

SMT-generated solution (correct): Walking through each instruction step-by-step: $(x, 0) \xrightarrow{\text{SWAP}} (0, x) \xrightarrow{\text{ADD}} (x, x) \xrightarrow{\text{MUL}} (x^2, x) \xrightarrow{\text{SUB}} (x^2 - x, x)$.

Since the final result is stored in `r0`, we obtain $x^2 - x = x(x-1)$, which is correct.

LLM-generated solution (incorrect): Walking through the LLM's instruction sequence step-by-step: $(x, 0) \xrightarrow{\text{SWAP}} (0, x) \xrightarrow{\text{INCR}} (1, x) \xrightarrow{\text{SWAP}} (x, 1) \xrightarrow{\text{SUB}} (x-1, 1) \xrightarrow{\text{SWAP}} (1, x-1) \xrightarrow{\text{MUL}} (1 \times (x-1), x-1)$. The final result in `r0` is $1 \times (x-1) = x - 1$, which is incorrect. The LLM obtained $x - 1$ instead of the required $x(x-1)$, demonstrating its failure to maintain the correct relationship between operands throughout the computation.

4.3 Protocol Converter

We now explore a case study comparing direct vs modular LLM approaches. The task involves designing a protocol converter connecting a core's 128-bit input bus to two peripherals, each consuming 128-bit messages via 32-bit buses. All ports use ready-valid/decoupled protocols [27, 17], comparable to "read/write blocking" dataflow process networks (DPNs) [32][11].

The protocol converter should route messages to output port 0 or 1 based on the message's least significant bit (LSB), incorporate FIFOs to prevent input blocking, and consider area/power/throughput trade-offs.

The converter uses three component types to achieve message routing and format conversion:

- `Demuxer`: This is a **combinational** demuxer which routes the input message to the particular output port depending on its LSB. If LSB=0, route to port 0; otherwise, if LSB=1, route to port 1.
- `FIFO`: A FIFO queue buffers data to prevent blocking upon data receipt from the input. The trade-off is that enlarging the FIFO will result in less input blocking and higher throughput, but increased area and power consumption.
- `Serializer`: Divides a 128-bit message into 32-bit segments. It sends out 32 bits per cycle. For an input to be fully processed, all four parts of the output must be consumed.

4.3.1 SMT Scalability Boundary Unlike our SMT-tractable case studies - Function Mapping (8-bit inputs, 256 combinations) and ALU (7 bounded instructions, ∼50K configurations) - the Protocol Converter crosses a scalability boundary:

- **Scale**: 128-bit inputs (2^{128} possible values) versus 8-bit inputs (256 values)
- **State complexity**: Multi-cycle stateful execution with FIFO queues and ready-valid handshaking creates $\sim 2^{200+}$ possible states [30]
- **Non-determinism**: Ready-valid protocols introduce combinatorial explosion in handshake interleavings

[11] Classical Kahn DPNs are write non-blocking as they assume infinitely large queues which is unrealistic for hardware.

This could exceed current solver capabilities [52, 14, 30]. We employ industry-standard simulation-based verification to confirm correctness in tested scenarios. Potential formal verification approaches are discussed in Section 6.2.

Optimisation Goal	**Approach**	**Tool Feed-back?**	**Best Gate Count**	**Best Cycle Count**
Size	Modular	No	545	24
Size	Modular	Yes	1086	24
Size	Raw RTL	No	N/A	N/A
Size	Raw RTL	Yes	475*	24
Throughput	Modular	No	1690	16
Throughput	Modular	Yes	1690	16
Throughput	Raw RTL	No	N/A	N/A
Throughput	Raw RTL	Yes	384	25

Table 3. LLM synthesis results for the protocol converter

We tested LLMs under varying conditions: optimisation objectives (size vs. throughput), synthesis approaches (modular vs. raw RTL), and tool feedback availability. We elaborated generated Chisel [8] designs to Verilog, tested functionality via testbench simulation, and synthesised using Yosys for gate counts. For this case study, we verified correct routing logic through a simulation-based testbench rather than exhaustive formal validation, checking that messages were properly demultiplexed based on their LSB and transmitted through the appropriate serialisers. We excluded SMT baselines due to scalability limitations for complex RTL circuits [30, 12, 31], focusing instead on LLM-testbench comparisons, with formal verification extensions discussed in Section 6.2.

Table 3 shows that modular approaches outperformed raw RTL, with the latter failing to produce compilable designs in two scenarios (marked "N/A").[12] Tool feedback proved essential for raw RTL success whilst showing minimal impact on modular approaches. Importantly, LLMs successfully distinguished between optimisation objectives, producing designs tailored for size or throughput.

For illustration, the correct modular approach produced clean component instantiation and connectivity:

```
val demuxer = Module(new Demuxer(width=128))
val fifo0 = Module(new FIFO(width=128, entries=8))
val fifo1 = Module(new FIFO(width=128, entries=8))
val serializer0 = Module(new Serializer(inputWidth=128,
↪  outputWidth=32))
```

[12] One instance where raw RTL achieved a smaller gate count may reflect a fortuitous initial guess requiring additional verification.

```
val serializer1 = Module(new Serializer(inputWidth=128,
↪  outputWidth=32))

input <> demuxer.in
demuxer.out0 <> fifo0.in
demuxer.out1 <> fifo1.in
fifo0.out <> serializer0.input
fifo1.out <> serializer1.input
serializer0.output <> output0
serializer1.output <> output1
```

In contrast, the failed raw RTL approach exhibited fundamental syntax errors and misconceptions:

```
// Split input into four 32-bit chunks
val chunks = VecInit(Seq(
  input.bits(31, 0), input.bits(63, 32), input.bits(95, 64),
  ↪  input.bits(127, 96)
))

// FIFOs for buffering output data
val fifo0 = Module(new Queue(UInt(32.W), 16))
val fifo1 = Module(new Queue(UInt(32.W), 16))

// Track which chunk index is currently being processed
val chunkCounter = RegInit(0.U(2.W))

// Determine the output port based on the LSB of the message
val targetFifo = if (input.bits(0) === 1.U) fifo1 else fifo0

// Determine if input should be accepted for processing
val canQueueAllChunks = fifo0.io.enq.ready && fifo1.io.enq.ready
input.ready := canQueueAllChunks
when(input.valid.ready)
```

5 Discussion

Our experimental results reveal a performance gap between LLMs and SMT solvers in hardware synthesis tasks. LLM failures were systematic across all test cases, with successes remaining superficial, improving syntactic correctness without achieving functional goals. Despite prompt engineering (10-20 hours per case study, Section 3) and iterative tool feedback (up to 20 iterations, Section 3.4), LLMs achieved 0% correctness in Function Mapping and 1.25% in ALU tasks (2 of 160 attempts).

Crucially, SMT-based counterexample feedback (providing specific failing input-output pairs) failed to improve LLM performance, revealing that exhaustive validation does not equate to effective improvement guidance. SMT solvers,

conversely, consistently produced correct, verifiable solutions within reasonable runtime when used for direct synthesis. Primary LLM failure modes were: (1) incorrect logic despite counterexample feedback, (2) inability to propagate constraints from concrete failures to parameter corrections, and (3) invalid syntax (wrong bit-widths, invalid Chisel syntax).

These patterns suggest a fundamental mismatch between `gpt-4o`'s natural-language training and the constraint propagation required for digital design synthesis in our tasks. Whilst LLMs parse counterexamples and understand that "input (5,2) produces wrong output", they lack explicit mechanisms to map concrete failures to parameter corrections. This explains why the naive LLM-SMT feedback loops in our evaluation prove ineffective: validation identifies errors but cannot guide correction without explicit constraint propagation.

SMT solvers, by contrast, leverage decades of progress in constraint solving [12] to provide systematic exploration of the solution space with mathematical guarantees. Their practicality is already demonstrated through deployment in industrial processor verification workflows [30]. For safety-critical applications, our results suggest using SMT for direct synthesis rather than purely as iterative LLM refinement, as validation capabilities do not necessarily translate to improvement effectiveness.

Rather than precluding LLM involvement entirely, these complementary strengths suggest hybrid methodologies using both approaches. Future work might improve LLM-SMT integration by: (1) using LLMs for high-level architectural decisions whilst delegating low-level constraint satisfaction to SMT, or (2) fine-tuning LLMs on counter-example-to-correction mappings to develop constraint propagation capabilities. Such methods could address two major EDA challenges: heuristic complexity and verification difficulty. LLMs excel in heuristics and natural language understanding but lack correctness guarantees, whilst formal methods ensure correctness but struggle with heuristic exploration. Incorporating formal specifications could avoid many correctness issues in LLM applications [42], leveraging natural language capabilities whilst ensuring formal guarantees, provided constraint propagation limitations are addressed.

However, realising such hybrid approaches introduces significant practical challenges. They require explicit natural-language descriptions and formal specifications of system modules, plus models to evaluate design costs (power, area, and cell count). Previous works such as Clover [57] and [35] suggest using agents to automatically generate natural-language descriptions for multi-agent integration. Approaches like [33] could offer efficient initial cost evaluation without full EDA tool execution. Future hybrid approaches must balance designer accessibility (formal specifications demand greater expertise than natural language) against correctness guarantees. Moreover, correctness checking and cost evaluation [61, 62] impose substantial computational overhead.

These implementation challenges notwithstanding, our evaluation establishes principles for reliable AI-assisted hardware design. Beyond the specific validation-improvement gap identified earlier, understanding when traditional heuristic approaches underperform remains crucial for effective deployment [29]. Our focus

on synthesis stages leaves backend implementation challenges like place-and-route optimisation unaddressed, where physical constraints and timing closure introduce additional complexity that merits separate investigation. Whilst hybrid methods cannot address all synthesis challenges - particularly those with significant abstraction gaps between high-level specifications and low-level implementations [59] - our systematic evaluation provides empirical guidance for practitioners: leverage LLMs for architectural exploration and natural language interpretation, but rely on formal methods for correctness-critical synthesis rather than iterative refinement.

6 Future Work

6.1 Proposed Case Study: LLM-Advantaged Tasks

Whilst our current evaluation examines domains where SMT excels (Boolean logic synthesis, instruction mapping), we acknowledge that the presented benchmarks do not showcase scenarios where LLMs might demonstrate advantages. Within the scope of this study, we prioritised establishing a rigorous baseline evaluation in domains amenable to formal verification, leaving LLM-advantaged case studies for future work where equivalent verification methodologies can be developed. To address this gap, we propose some future case studies which might lend themselves to LLMs in areas where SMT-based approaches face significant challenges.

Unbounded-Loop Microcode Generation: Unlike the bounded 7-instruction ALU sequences in our current evaluation, generating microcode for algorithms requiring unbounded iteration presents challenges for SMT-based approaches. Whilst SMT excels at bounded problems, unbounded verification requires either explicit loop unrolling (limiting completeness) or inductive invariant synthesis (computationally expensive) [31, 52]. This limitation suggests investigating whether LLMs' pattern recognition from training on iterative algorithms could complement SMT's verification capabilities [41].

Architecture Extraction from Narrative Specifications: Consider "Design a memory hierarchy for a mobile graphics processor prioritising power efficiency, supporting texture streaming for 4K displays, and degrading gracefully under thermal constraints". Such specifications contain implicit trade-offs and domain terminology that resist formalisation. LLMs' natural language understanding enables parsing these narratives, identifying architectural components (cache hierarchies, power management, thermal throttling), and proposing system-level configurations - capabilities that SMT solvers lack, as they require explicit mathematical constraints and cannot handle semantic ambiguity.

6.2 Formal Verification of Complex RTL Designs

As detailed in Section 4.3.1, the Protocol Converter represents a fundamental scalability boundary where current SMT techniques face limitations due to

state explosion from 128-bit inputs, stateful multi-cycle execution, and non-deterministic ready-valid handshaking.

Future work could investigate compositional verification and abstraction techniques. Word-level model checking using the BTOR2 format [47] offers a promising direction. SMT-based bounded model checking tools like Pono [37] and CBMC [16] could verify bounded executions of protocol converters, whilst abstraction-refinement techniques could handle larger state spaces [30]. Additionally, modular SMT encodings that verify individual components before composition could scale better than monolithic approaches, following assume-guarantee reasoning patterns established in hardware verification [13].

6.3 Other Directions

Future work could expand support for additional agents, such as testbenches and fuzzers, to enhance testing robustness and versatility. The framework could generate not only architecture instances but also generators themselves, offering greater design adaptability. Another promising direction involves integrating formal verification collateral generation, as suggested by existing research [44, 25], to ensure rigorous validation and reliability.

Further exploration could focus on domain-specific design, including clock gating, power gating, and dynamic voltage and frequency scaling (DVFS), with custom agents and optimisers. Developing new building blocks and abstractions will enhance modularity and expand application scope. Additionally, traditional heuristic optimisation engines guided by LLMs could yield novel insights and improved efficiency.

Extending our synthesis-focused evaluation to consider backend steps such as place-and-route would provide a more complete assessment of LLM capabilities across the full implementation flow. Such integration could leverage LLMs' pattern recognition for floorplanning heuristics whilst maintaining formal verification for timing closure validation.

6.4 Artefact Availability

A public artefact available at `https://zenodo.org/records/19326189` contains machine-readable prompt templates and SMT-LIB benchmarks.

For further reproducibility and completeness, we plan to provide an extended technical report on arXiv including additional prompt documentation and detailed experimental setup.

Acknowledgements

The authors would like to thank Synopsys Inc for their generous support through research internship funds. The authors also extend their gratitude to Stelios Diamantidis for his invaluable guidance and support throughout this project. Author Tony Tan was supported by the UKRI Engineering and Physical Sciences Research Council (EPSRC) grant number UKRI4035.

References

[1] Abate, A., David, C., Kesseli, P., Kroening, D., Polgreen, E.: Counterexample guided inductive synthesis modulo theories. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification. pp. 270–288. Springer International Publishing, Cham (2018), http://www.kroening.com/papers/cav2018-synthesis.pdf

[2] Abdollahi, M., Yeganli, S.F., Baharloo, A., Baniasadi, A.: Hardware design and verification with large language models: A scoping review, challenges, and open issues. Electronics **14**(1) (2025). https://doi.org/10.3390/electronics14010120, http://www.mdpi.com/2079-9292/14/1/120

[3] Aggarwal, P., Madaan, A., Yang, Y., Mausam: Let's sample step by step: Adaptive-consistency for efficient reasoning and coding with LLMs (2023), http://arxiv.org/abs/2305.11860

[4] Akyash, M., Azar, K., Kamali, H.: RTL++: Graph-enhanced LLM for RTL code generation (2025), http://arxiv.org/abs/2505.13479

[5] Alsaqer, S., Alajmi, S., Ahmad, I., Alfailakawi, M.: The potential of LLMs in hardware design. Journal of Engineering Research **13**(3), 2392–2404 (2025). https://doi.org/10.1016/j.jer.2024.08.001

[6] Alur, R., Bodik, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: 2013 Formal Methods in Computer-Aided Design. pp. 1–8 (2013). https://doi.org/10.1109/FMCAD.2013.6679385

[7] Baartmans, R., Ensinger, A., Agostinelli, V., Chen, L.: ML for hardware design interpretability: Challenges and opportunities. arXiv (2025)

[8] Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., Asanović, K.: Chisel: constructing hardware in a Scala embedded language, pp. 1216–1225. ACM (2012). https://doi.org/10.1145/2228360.2228584, http://people.eecs.berkeley.edu/~krste/papers/chisel-dac2012.pdf

[9] Bailey, B.: What will that chip cost? http://web.archive.org/web/20240719072950/https://semiengineering.com/what-will-that-chip-cost/ (Oct 2023), http://web.archive.org/web/20240719072950/https://semiengineering.com/what-will-that-chip-cost/

[10] Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 415–442. Springer International Publishing, Cham (2022), http://www-cs.stanford.edu/~preiner/publications/2022/BarbosaBBKLMMMN-TACAS22.pdf

[11] Barendregt, H.P.: The quest for correctness. Amsterdam: Stichting Mathematisch Centrum (1996), http://repository.ubn.ru.nl/bitstream/handle/2066/17273/13361.pdf

[12] Barrett, C., Sebastiani, R., Seshia, S., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Second Edition, Frontiers in Artificial Intelligence and Applications, vol. 336, chap. 33, pp. 825–885. IOS Press (Feb 2021). https://doi.org/10.3233/FAIA201017, http://theory.stanford.edu/~barrett/pubs/BSST21.pdf

[13] Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.B., Reinkemeier, P., Sangiovanni-Vincentelli, A., Damm, W., Henzinger, T., Larsen,

K.G.: Contracts for Systems Design: Theory. Research Report RR-8759, Inria Rennes Bretagne Atlantique ; INRIA (Jul 2015), http://inria.hal.science/hal-01178467

[14] Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 193–207. Springer Berlin Heidelberg, Berlin, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0˙14

[15] Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., Van Rossum, P., Schulz, S., Sebastiani, R.: Mathsat: Tight integration of sat and mathematical decision procedures. Journal of Automated Reasoning **35**(1), 265–293 (2005). https://doi.org/10.1007/s10817-005-9004-z, http://brenta.dit.unitn.it/~rseba/papers/jar_sat05.pdf

[16] Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 168–176. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)

[17] Cook, H., Terpstra, W., Lee, Y.: Diplomatic design patterns: A TileLink case study. In: 1st Workshop on Computer Architecture Research with RISC-V. vol. 23 (2017), http://carrv.github.io/2017/papers/cook-diplomacy-carrv2017.pdf

[18] Donda, K., Brahmkhatri, P., Zhu, Y., Dey, B., Slesarenko, V.: Machine learning for inverse design of acoustic and elastic metamaterials. Current Opinion in Solid State and Materials Science **35**, 101218 (2025). https://doi.org/http://doi.org/10.1016/j.cossms.2025.101218, http://www.sciencedirect.com/science/article/pii/S1359028625000051

[19] Dutertre, B., De Moura, L.: The yices smt solver. SRI International Computer Science Laboratory Technical Reports (2006), http://yices.csl.sri.com/papers/tool-paper.pdf

[20] Faghih, F., Bonakdarpour, B.: SMT-based synthesis of distributed self-stabilizing systems. ACM Trans. Auton. Adapt. Syst. **10**(3) (Oct 2015). https://doi.org/10.1145/2767133, http://doi.org/10.1145/2767133

[21] Fei, W., Niu, X., Zhou, P., Hou, L., Bai, B., Deng, L., Han, W.: Extending context window of large language models via semantic compression. In: Ku, L.W., Martins, A., Srikumar, V. (eds.) Findings of the Association for Computational Linguistics: ACL 2024. pp. 5169–5181. Association for Computational Linguistics, Bangkok, Thailand (Aug 2024). https://doi.org/10.18653/v1/2024.findings-acl.306, http://aclanthology.org/2024.findings-acl.306/

[22] Gadde, D.N., Kumar, A., Nalapat, T., Rezunov, E., Cappellini, F.: All artificial, less intelligence: GenAI through the lens of formal verification (2024), http://arxiv.org/abs/2403.16750

[23] Gadde, D.N., Radhakrishna, K.K., Viswambharan, V.N., Kumar, A., Lettnin, D., Kunz, W., Simon, S.: Hey AI, generate me a hardware code, agentic AI-based hardware design and verification. In: 2025 38th SBC/SBMicro/IEEE Symposium on Integrated Circuits and Systems Design (SBCCI). pp. 1–5 (2025). https://doi.org/10.1109/SBCCI66862.2025.11218681

[24] Gao, Z., Boning, D.S.: A review of Bayesian methods in electronic design automation (2023), http://arxiv.org/abs/2304.09723

[25] Greenman, B., Prasad, S., Di Stasio, A., Zhu, S., De Giacomo, G., Krishnamurthi, S., Montali, M., Nelson, T., Zizyte, M.: Misconceptions in finite-trace and infinite-trace linear temporal logic. In: Platzer, A., Rozier, K.Y., Pradella, M., Rossi, M. (eds.) Formal Methods. pp. 579–599. Springer Nature Switzerland, Cham (2025)

[26] Hitarth, S., Codel, C., Lachnitt, H., Dutertre, B.: Extending DRAT to SMT. In: 2024 Formal Methods in Computer-Aided Design (FMCAD). pp. 01–11 (2024). https://doi.org/10.34727/2024/isbn.978-3-85448-065-5˙8
[27] Jayarama, K.: A case study on effective pipeline design in digital systems (Jun 2024), http://web.archive.org/web/20250522085640/https://verilog-meetup.com/2024/06/20/a-case-study-on-effective-pipeline-design-in-digital-system/
[28] Kahng, A.B., Lienig, J., Markov, I.L., Hu, J.: VLSI physical design: from graph partitioning to timing closure, vol. 312. Springer (2011)
[29] Karimi, P., Pirelli, S., Kakarla, S.K.R., Beckett, R., Segarra, S., Li, B., Namyar, P., Arzani, B.: Towards safer heuristics with XPlain. In: Proceedings of the 23rd ACM Workshop on Hot Topics in Networks. p. 68–76. HotNets '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3696348.3696884, http://doi.org/10.1145/3696348.3696884
[30] Kern, C., Greenstreet, M.R.: Formal verification in hardware design: a survey. ACM Trans. Des. Autom. Electron. Syst. **4**(2), 123–193 (Apr 1999). https://doi.org/10.1145/307988.307989, http://doi.org/10.1145/307988.307989
[31] Konnov, I., Kukovec, J., Tran, T.H.: TLA+ model checking made symbolic. Proc. ACM Program. Lang. **3**(OOPSLA) (Oct 2019). https://doi.org/10.1145/3360549, http://doi.org/10.1145/3360549
[32] Lee, E., Parks, T.: Dataflow process networks. Proceedings of the IEEE **83**(5), 773–801 (1995). https://doi.org/10.1109/5.381846
[33] Levy, A., Walston, J., Samanta, S., Raina, P., Diamantidis, S.: Fastpase: An ai-driven fast ppa speculation engine for rtl design space optimization. In: 2024 25th International Symposium on Quality Electronic Design (ISQED). pp. 1–8 (2024). https://doi.org/10.1109/ISQED60706.2024.10528750
[34] Li, H.: Mill: A fast, scalable JVM build tool (2018), http://mill-build.org/
[35] Li, Y., Parsert, J., Polgreen, E.: Guiding enumerative program synthesis with large language models. In: Gurfinkel, A., Ganesh, V. (eds.) Computer Aided Verification. pp. 280–301. Springer Nature Switzerland, Cham (2024)
[36] Makharev, V., Ivanov, V.: Code summarization beyond function level (2025)
[37] Mann, M., Irfan, A., Lonsing, F., Yang, Y., Zhang, H., Brown, K., Gupta, A., Barrett, C.: Pono: A flexible and extensible SMT-based model checker. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification. pp. 461–474. Springer International Publishing, Cham (2021)
[38] Melus, B., Suelzer, J., Hagedon, M., Reano, R.: Investigation of adjoint method inverse design applied to photonic integrated circuit fiber-to-chip edge couplers. In: Schröder, H., Chen, R.T. (eds.) Optical Interconnects XXIV. p. 10. SPIE (Mar 2024). https://doi.org/10.1117/12.3001211, http://dx.doi.org/10.1117/12.3001211
[39] Mendoza, D., Hahn, C., Trippel, C.: Translating natural language to temporal logics with large language models and model checkers. In: 2024 Formal Methods in Computer-Aided Design (FMCAD). pp. 1–11 (2024). https://doi.org/10.34727/2024/isbn.978-3-85448-065-5˙17
[40] Meng, B., Debnath, J., Varanasi, S.C., Manoloios, E., Durling, M., Paul, S., Prince, D., Alsabbagh, S., Haadsma, R., McMillan, C., Zhang, C., Oates, T.: Towards a correct-by-construction design of integrated modular avionics. In: 2023 Formal Methods in Computer-Aided Design (FMCAD). pp. 221–227 (2023). https://doi.org/10.34727/2023/isbn.978-3-85448-060-0˙30

[41] Mohsin, A., Janicke, H., Wood, A., Sarker, I.H., Maglaras, L., Janjua, N.: Can we trust large language models' generated code? a framework for in-context learning, security patterns, and code evaluations across diverse LLMs (2024), http://arxiv.org/abs/2406.12513

[42] Mondal, R., Tang, A., Beckett, R., Millstein, T., Varghese, G.: What do LLMs need to synthesize correct router configurations? In: Proceedings of the 22nd ACM Workshop on Hot Topics in Networks. p. 189–195. HotNets '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3626111.3628194, http://doi.org/10.1145/3626111.3628194

[43] de Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008), http://algos.inesc.pt/projects/nanotime/ref12.pdf

[44] Nelson, T., Greenman, B., Prasad, S., Dyer, T., Bove, E., Chen, Q., Cutting, C., Del Vecchio, T., LeVine, S., Rudner, J., Ryjikov, B., Varga, A., Wagner, A., West, L., Krishnamurthi, S.: Forge: A tool and language for teaching formal methods. Proc. ACM Program. Lang. **8**(OOPSLA1) (Apr 2024). https://doi.org/10.1145/3649833, http://doi.org/10.1145/3649833

[45] Neumann, A., Kirsten, E., Zafar, M.B., Singh, J.: Position is power: System prompts as a mechanism of bias in LLMs. In: Proceedings of the 2025 ACM Conference on Fairness, Accountability, and Transparency. p. 573–598. FAccT '25, Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3715275.3732038, http://doi.org/10.1145/3715275.3732038

[46] Niemetz, A., Preiner, M.: Bitwuzla. In: Enea, C., Lal, A. (eds.) Computer Aided Verification (CAV). pp. 3–17. Springer Nature Switzerland, Cham (2023), http://cs.stanford.edu/~preiner/publications/2023/NiemetzP-CAV23.pdf

[47] Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2, BtorMC and Boolector 3.0. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification. pp. 587–595. Springer International Publishing, Cham (2018)

[48] Norman, C., Godbole, A., Manerkar, Y.A.: PipeSynth: Automated synthesis of microarchitectural axioms for memory consistency. In: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3. p. 513–527. ASPLOS 2023, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3582016.3582056, http://doi.org/10.1145/3582016.3582056

[49] Parker, J., Vazou, N., Hicks, M.: LWeb: information flow security for multi-tier web applications. Proc. ACM Program. Lang. **3**(POPL) (Jan 2019). https://doi.org/10.1145/3290388, http://doi.org/10.1145/3290388

[50] Patel, A., Reddy, S., Bahdanau, D., Dasigi, P.: Evaluating in-context learning of libraries for code generation (2024), http://arxiv.org/abs/2311.09635

[51] Pei, Z., Zhen, H.L., Yuan, M., Huang, Y., Yu, B.: BetterV: controlled Verilog generation with discriminative guidance. In: Proceedings of the 41st International Conference on Machine Learning. ICML'24, JMLR.org (2024)

[52] Roy, T., Hsiao, M.: Reachability analysis in RTL circuits using k-induction bounded model checking. In: 2017 IEEE International High Level Design Validation and Test Workshop (HLDVT). pp. 9–16 (2017). https://doi.org/10.1109/HLDVT.2017.8167457

[53] Sadigova, P.: SMT-based verification of building management systems. Ph.D. thesis, King's College London (2022), http://kclpure.kcl.ac.uk/ws/portalfiles/portal/270830485/2024_Sadigova_Parvin_1272461_ethesis.pdf
[54] Schoeberl, M., Andersen, S.T., Rasmussen, K.J.H., Lin, R.: Towards an open-source verification method with Chisel and Scala. In: Proc. 3rd Workshop Open-Source EDA Technol.(WOSET) (2020), http://woset-workshop.github.io/PDFs/2020/a02.pdf
[55] Schubert, M.F., Cheung, A.K.C., Williamson, I.A.D., Spyra, A., Alexander, D.H.: Inverse design of photonic devices with strict foundry fabrication constraints. ACS Photonics **9**(7), 2327–2336 (Jun 2022). https://doi.org/10.1021/acsphotonics.2c00313, http://dx.doi.org/10.1021/acsphotonics.2c00313
[56] Srikumar, P.: Fast and wrong: The case for formally specifying hardware with LLMs. In: WACI at ASPLOS (2023), http://asplos-conference.org/wp-content/uploads/2023/waci/6-Fast-and-wrong-priya-srikumarpdf.pdf
[57] Sun, C., Sheng, Y., Padon, O., Barrett, C.: Clover: Closed-loop verifiable code generation. In: Avni, G., Giacobbe, M., Johnson, T.T., Katz, G., Lukina, A., Narodytska, N., Schilling, C. (eds.) AI Verification. pp. 134–155. Springer Nature Switzerland, Cham (2024)
[58] Sun, W., Zhang, Y., Zhu, J., Wang, Z., Fang, C., Zhang, Y., Feng, Y., Huang, J., Wang, X., Jin, Z., Liu, Y.: Commenting higher-level code unit: Full code, reduced code, or hierarchical code summarization (2025)
[59] Synopsys: DesignWare library - datapath and building block IP, http://www.synopsys.com/dw/buildingblock.php
[60] Tang, J., Qin, J., Thorat, K., Zhu-Tian, C., Cao, Y., Yang, Zhao, Ding, C.: HiVeGen - hierarchical LLM-based Verilog generation for scalable chip design (2024), http://arxiv.org/abs/2412.05393
[61] Wang, E., Daniel, L., Zohar, Y., Barrett, C.: How I learned to stop worrying and love physical design. In: 2024 Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE) (2024), http://capra.cs.cornell.edu/latte24/paper/5.pdf
[62] Wang, E., Schmidt, C., Izraelevitz, A., Wright, J., Nikolić, B., Alon, E., Bachrach, J.: A methodology for reusable physical design. In: 2020 21st International Symposium on Quality Electronic Design (ISQED). pp. 243–249 (2020). https://doi.org/10.1109/ISQED48828.2020.9136999
[63] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E.H., Le, Q.V., Zhou, D.: Chain-of-thought prompting elicits reasoning in large language models. In: Proceedings of the 36th International Conference on Neural Information Processing Systems. NIPS '22, Curran Associates Inc., Red Hook, NY, USA (2022). https://doi.org/10.5555/3600270.3602070, http://dl.acm.org/doi/10.5555/3600270.3602070
[64] Weng, J., Liu, S., Dadu, V., Wang, Z., Shah, P., Nowatzki, T.: DSAGEN: Synthesizing programmable spatial accelerators. In: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). pp. 268–281 (2020). https://doi.org/10.1109/ISCA45697.2020.00032
[65] Wolf, C., Glaser, J., Kepler, J.: Yosys - a free Verilog synthesis suite. In: Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip). vol. 97 (2013), http://yosyshq.net/yosys/files/yosys-austrochip2013.pdf
[66] Xu, K., Qiu, R., Zhao, Z., Zhang, G.L., Schlichtmann, U., Li, B.: LLM-aided efficient hardware design automation. arXiv (2024)

[67] Xu, R., Xiao, Y., Luo, J., Liang, Y.: HECTOR: A multi-level intermediate representation for hardware synthesis methodologies. In: Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design. ICCAD '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3508352.3549370, http://doi.org/10.1145/3508352.3549370

[68] Zhang, R., Wen, Y., Cheng, S., Huang, D., Peng, S., Guo, J., Jin, P., Zhao, J., Ma, T., Zhu, Y., Hao, Y., Zhao, Y., Liang, S., Wang, Y., Hu, X., Du, Z., Cui, H., Li, L., Guo, Q., Chen, Y.: QiMeng: Fully automated hardware and software design for processor chip (2025), http://arxiv.org/abs/2506.05007

[69] Zheng, K., Han, J.M., Polu, S.: MiniF2F: a cross-system benchmark for formal olympiad-level mathematics. arXiv preprint arXiv:2109.00110 (2021), http://openreview.net/pdf?id=9ZPegFuFTFv

[70] Zhong, R., Du, X., Kai, S., Tang, Z., Xu, S., Zhen, H.L., Hao, J., Xu, Q., Yuan, M., Yan, J.: LLM4EDA: Emerging progress in large language models for electronic design automation (2023)