PyHDL-Eval: An LLM Evaluation Framework for Hardware Design Using Python-Embedded DSLs

Christopher Batten*
cbatten@cornell.edu
Cornell University
Ithaca, NY, USA

Nathaniel Pinckney
npinckney@nvidia.com
NVIDIA Corporation
Santa Clara, CA, USA

Mingjie Liu
mingjiel@nvidia.com
NVIDIA Corporation
Santa Clara, CA, USA

Haoxing Ren
haoxingr@nvidia.com
NVIDIA Corporation
Santa Clara, CA, USA

Brucek Khailany
bkhailany@nvidia.com
NVIDIA Corporation
Santa Clara, CA, USA

Abstract

Embedding hardware design frameworks within Python is a promising technique to improve the productivity of hardware engineers. At the same time, there is significant interest in using large-language models (LLMs) to improve key chip design tasks. This paper describes PyHDL-Eval, a new framework for evaluating LLMs on specification-to-RTL tasks in the context of Python-embedded domain-specific languages (DSLs). The framework includes 168 problems, Verilog reference solutions, Verilog test benches, Python test scripts, and workflow orchestration scripts. We use the framework to conduct a detailed case study comparing five LLMs (CodeGemma 7B, Llama3 8B/70B, GPT4, and GPT4 Turbo) targeting Verilog and five Python-embedded DSLs (PyMTL3, PyRTL, MyHDL, Migen, and Amaranth). Our results demonstrate the promise of in-context learning when applied to smaller models (e.g., pass rate for CodeGemma 7B improves from 14.9% to 32.7% on Verilog) and Python-embedded DSLs (e.g., pass rate for LLama3 70B improves from 0.6% to 33.0% on PyMTL3). We find LLMs perform better when targeting Verilog as compared to Python-embedded DSLs (e.g., pass rate for GPT4 Turbo is 72.2% on Verilog and 29.8–62.0% on the Python-embedded DSLs) despite using a popular general-purpose host language. PyHDL-Eval will serve as a useful framework for future research at the intersection of Python-embedded DSLs and LLMs.

CCS Concepts

• **Hardware** → **Hardware description languages and compilation**; • **Computing methodologies** → **Machine learning**.

Keywords

hardware description languages, Python-embedded domain-specific languages, large language models

*Also with NVIDIA Corporation.

MLCAD '24, September 9–11, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0699-8/24/09
https://doi.org/10.1145/3670474.3685948

ACM Reference Format:
Christopher Batten, Nathaniel Pinckney, Mingjie Liu, Haoxing Ren, and Brucek Khailany. 2024. PyHDL-Eval: An LLM Evaluation Framework for Hardware Design Using Python-Embedded DSLs . In *2024 ACM/IEEE International Symposium on Machine Learning for CAD (MLCAD '24), September 9–11, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3670474.3685948

1 Introduction

Novel domain-specific languages (DSLs) can improve hardware designer productivity by introducing new execution semantics, type systems, and/or custom compilation flows [14, 27, 28, 38]. Alternatively, embedded DSLs for hardware augment a software host language with a carefully designed library to provide the illusion of a new DSL while maintaining all of the features of the host language [3–6, 32–34]. Python-embedded DSLs for hardware design and verification capitalize on Python's popularity but also its unique mix of language features (e.g, lightweight syntax, dynamic typing, reflection, metaprogramming) and ecosystem (e.g., third-party libraries, package management) [2, 11–13, 17–19, 23, 38]. *Trend #1: Python-embedded DSLs are increasingly being used to improve the productivity of hardware design and verification.*

Large-language models (LLMs) can also improve hardware designer productivity by enabling engineers to express their intent as a natural-language prompt and using an LLM to generate low-level design or verification code [42]. Early work identified the key challenges and opportunities in using general-purpose LLMs for chip design [7, 10]. More recently, there has been work on new LLM benchmarks and fine-tuned LLMs for Verilog RTL code completion and specification-to-RTL tasks (e.g., VerilogEval [21], RTLLM [24], VeriGen [35, 36], RTLCoder [22], RTL-Repo [1]). Agent-based approaches use this recent work as one step in an automated and iterative workflow (e.g., RTLFixer [39], AutoChip [37]). LLMs are also being used more broadly to target other HDLs [40], generate hardware accelerators [15], generate dynamic test stimulus [41], generate formal assertions [31], and serve as a general chip-design AI assistant [20]. *Trend #2: LLMs are increasingly being used to improve the productivity of hardware design and verification.*

This paper describes PyHDL-Eval[1], a framework designed to enable research at the intersection of these two trends. Section 2 briefly reviews the Python-embedded DSLs that are considered in

[1]PyHDL-Eval is available at https://github.com/cornell-brg/pyhdl-eval

```python
from pymtl3 import *

class Top(Component):
 def construct(s):
  s.in_ = InPort (Bits8)
  s.en  = InPort ()
  s.out = OutPort(Bits8)

  # Sequential Logic

  s.in_r = Wire(Bits8)

  @update_ff
  def seq():
   if s.reset:
    s.in_r <<= 0
   elif s.en:
    s.in_r <<= s.in_

  # Combinational Logic

  @update
  def comb():
   s.out @= s.in_r + 1
```

(a) PyMTL3

```python
from pyrtl import *

def Top( in_, en ):
 out = WireVector(8)

 # Sequential Logic

 in_r = Register( 8,
         reset_value=0 )

 with conditional:
  with en:
   in_r.next |= in_

 # Combinational Logic

 out <<= in_r + 1

 return out
```

(b) PyRTL

```python
from myhdl import *

@block
def Top( clk, reset,
         in_, en, out ):

 # Sequential Logic

 in_r = Signal(modbv(0)[8:])

 @always( clk.posedge )
 def seq():
  if reset:
   in_r.next = 0
  elif en:
   in_r.next = in_

 # Combinational Logic

 @always_comb
 def comb():
  out.next = in_r + 1

 return seq, comb
```

(c) MyHDL

```python
from migen import *

@ResetInserter()
class Top(Module):
 def __init__(s):
  s.in_ = Signal(8)
  s.en  = Signal()
  s.out = Signal(8)

  # Sequential Logic

  in_r = Signal(8,reset=0)

  s.sync += \
   If( s.en,
    in_r.eq(s.in_) )

  # Combinational Logic

  s.comb += \
   s.out.eq( in_r + 1 )
```

(d) Migen

```python
from amaranth import *

class Top(Elaboratable):
 def __init__(s):
  s.in_ = Signal(8)
  s.en  = Signal()
  s.out = Signal(8)

def elaborate(s,platform):
 m = Module()

 # Sequential Logic

 in_r = Signal(8,reset=0)

 with m.If( s.en ):
  m.d.sync += in_r.eq(s.in_)

 # Combinational Logic

 m.d.comb += \
   s.out.eq( in_r + 1 )

 return m
```

(e) Amaranth

Figure 1: Python-Embedded Domain-Specific Languages (DSLs) – An 8-bit incrementer with a positive edge-triggered input register, an enable, and an active-high synchronous reset is implemented in five different Python-embedded DSLs; each with different syntax for expressing hardware modules, port-based interfaces, reset, sequential logic, and combinational logic.

this work. Section 3 describes the framework which includes 168 problems for a specification-to-RTL task developed using an ontological approach to cover 19 categories of RTL design. Our approach enables a finer grain understanding of LLM capabilities compared to prior work [21]. Each problem includes a natural-language specification, Verilog reference solution, Verilog test bench, and Python test script. The framework also includes workflow orchestration scripts to query LLMs, analyze compile- and simulation-time errors, and manage experiments. Section 4 describes a case study using PyHDL-Eval to compare five LLMs (CodeGemma 7B [16], Llama3 8B/70B [26], GPT4 [30], GPT4 Turbo [29]) targeting Verilog and five Python-embedded DSLs (PyMTL3 [18], PyRTL [11], MyHDL [13], Migen [19], Amaranth [2]). We find in-context learning (ICL) through few-shot examples [8] is critical when targeting Python-embedded DSLs. Our results also challenge conventional wisdom suggesting LLMs will perform better targeting Python-embedded DSLs compared to Verilog, since LLM training data includes orders-of-magnitude more Python vs. Verilog code [9].

The paper makes the following contributions: (1) to our knowledge, this is the first LLM evaluation framework for hardware design targeting multiple Python-embedded DSLs; (2) we evaluate the potential for ICL to improve the LLM pass rate for Python-embedded DSLs; and (3) we perform the first rigorous analysis of LLMs targeting both Verilog and five Python-embedded DSLs.

2 Background on Python-Embedded DSLs

We consider five popular Python-embedded DSLs (see Figure 1), which all include: hardware-centric fixed-bitwidth types; a pure-Python simulation engine; and support for translating designs from Python into Verilog RTL to drive either an ASIC or FPGA toolflow.

PyMTL3 has an emphasis on multi-level modeling with support for functional-, cycle-, and register-transfer-level modeling [18]. PyMTL3 uses classes to represent hardware modules and decorated closures with standard Python code to model sequential and combinational logic. PyMTL3 uses implicit clock and reset signals. Only positive-edge triggered sequential logic in a single clock domain with synchronous reset is supported.

PyRTL focuses on RTL modeling through explicit hardware construction [11]. PyRTL uses functions to represent hardware modules. Sequential logic must use explicit registers. Operator overloading constructs graphs of primitives for combinational logic with context managers for conditional operations. PyRTL uses implicit clock and reset signals. Only positive-edge triggered sequential logic in a single clock domain with synchronous reset is supported.

MyHDL is one of the oldest Python-embedded DSLs [13]. MyHDL uses decorated functions to represent hardware modules and decorated closures with standard Python code to model sequential and combinational logic. MyHDL uses explicit clock and reset signals. MyHDL supports event-driven execution semantics to enable multiple clock domains with synchronous/asynchronous reset.

Migen is the foundation for the LiteX framework that enables rapidly building FPGA-based SoCs [19]. **Amaranth** is a recent fork with similar syntax [2]. Migen/Amaranth use classes to represent hardware modules. Special Migen/Amaranth functions are required to model assignment and conditional operations. Migen and Amaranth have different syntax for modeling conditional operations. Migen/Amaranth use implicit clock and reset signals. They use event-driven execution semantics that enable multiple clock domains with synchronous/asynchronous reset.

3 PyHDL-Eval Framework

Figure 2 illustrates the PyHDL-Eval framework. In this section, we describe the framework's problem specifications and prompts, functional testing, and workflow orchestration scripts.

Problem Category			Examples of Problems in Category	Num Probs	Spec Lines			Spec Tokens			Reference LOC			Num Test Cases		
					Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
1	comb	const	constant zero, one, lohi, 32-bit value	4	4	5.0	8	48	60	88	6	6.5	8	1	1.0	1
2	comb	wires	pass through, split, bit/byte reverse, sign extend	8	6	10.1	21	65	130	274	7	12.1	19	2	2.0	2
3	comb	gates	single logic gate, multiple logic gates, natural language	13	6	11.0	16	58	128	231	8	11.2	17	1	1.2	2
4	comb	bool	logic equations, truth table, K-map, waveform, natural language	15	8	19.6	52	72	231	746	9	16.8	29	1	1.0	1
5	comb	mux	various bitwidths and number of inputs, mux/demux	9	9	13.8	17	116	162	198	8	21.7	34	1	2.0	3
6	comb	codes	encoder, decoder, priority encoder, graycode, BCD, ASCII, parity	14	6	7.9	11	67	109	152	7	19.9	27	1	1.8	3
7	comb	arith	add/sub, popcount, multiplication, shift/rotate, comparison, ALU, sorter	17	7	10.7	20	85	130	293	8	13.4	32	2	4.1	9
8	comb	fsm	FSM tables and diagrams for next state and output logic	9	22	27.9	49	234	309	511	30	37.2	55	1	1.3	2
9	comb	param	parameterized: gates, mux, encoder, decoder, priority encoder, rotator	8	7	10.4	15	65	133	175	9	14.6	22	2	3.9	6
10	seq	gates	single gate, multiple gates, natural language	9	7	12.4	22	74	150	314	9	14.1	21	1	2.4	4
11	seq	bool	truth table, waveform	5	16	24.4	32	167	252	318	9	11.8	16	1	2.2	4
12	seq	sreg	shift registers: SISO, PISO, SIPO, universal, bidir, LFSR	7	15	21.1	28	177	280	459	17	20.6	25	3	4.7	7
13	seq	count	counters: binary, up/down, saturating, variable, decade, timer, clock	9	11	16.1	27	135	191	323	17	27.8	60	4	6.4	10
14	seq	edge	edge detect, edge capture, edge counters, max switching	5	24	26.8	29	331	356	369	11	19.4	33	4	6.0	9
15	seq	arb	arbiters: priority, rotational oblivious, round robin, grant-hold, weighted	5	34	42.6	60	423	565	812	47	60.6	93	6	7.8	12
16	seq	fsm	FSM tables and diagrams, natural language	13	22	28.5	49	260	317	503	34	43.4	63	4	4.3	6
17	seq	mem	1r1w, 2r1w register file (forwarding, reg zero, byte enables), 1s1w CAM	6	15	19.3	24	185	249	346	16	22.0	33	3	4.5	6
18	seq	arith	bit serial incrementer, bit serial adder, byte-serial adder, accumulator	4	23	26.8	34	264	327	404	17	27.3	31	6	6.3	7
19	seq	pipe	1/2/3-stage delay, 1/2-stage 2/3/4-input adder, 2-stage minmax	8	20	28.5	38	234	379	554	11	24.1	41	2	4.0	6
			Across All Categories:	**168**	**4**	**17.8**	**60**	**48**	**216**	**812**	**6**	**21.7**	**93**	**1**	**3.2**	**12**

Table 1: PyHDL-Eval Problem Categories – Spec = problem specification; lines/tokens are for just the problem specification without blank lines, in-context learning examples, delimiters/explanation suffix, or system message; specification lines exclude blank lines; specification tokens use GPT4 Turbo tokenizer; reference lines-of-code (LOC) are for Verilog reference excluding comments and blank lines.

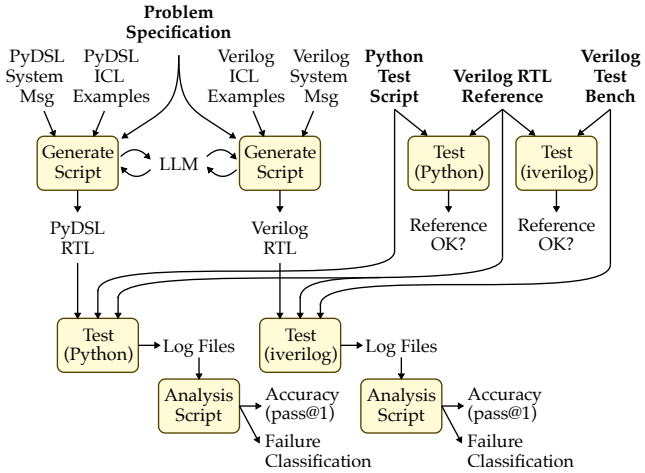

Figure 2: PyHDL-Eval Framework – ICL = in-context learning; iverilog = Icarus Verilog; PyDSL = one of five Python-embedded DSLs: PyMTL3, PyRTL, MyHDL, Migen, Amaranth.

3.1 Problem Specifications and Prompts

Table 1 illustrates the 19 categories of RTL design that can be evaluated using PyHDL-Eval (see Appendix 6 for complete problem list). The specification lines and tokens and the lines of code in the reference solution provide a very rough measure of problem complexity. Only problems that can be reasonably implemented in all five Python-embedded DSLs studied in this work are considered; thus problems covering latch-based design, asynchronous reset, and negative-edge triggered logic are excluded. Our ontological approach is a key feature of PyHDL-Eval, since it enables fine-grain evaluation across both different LLMs and Python-embedded DSLs.

```
Implement a hardware module named TopModule with the following
interface. All input and output ports are one bit unless otherwise
specified.

 - input  clk
 - input  ld
 - input  in_ (3 bits)
 - output out (3 bits)
 - output done

The module should implement a 3-bit binary down counter. When the
input ld is high, then the input in_ should be loaded into the
counter. The counter should then count down by one every cycle;
when the counter reaches zero it should stop counting and remain
at zero until a new value is loaded into the counter. The internal
counter register should be directly connected to the output port.
The output done should be (combinationally) set to one when the
counter is zero. Assume all sequential logic is triggered on the
positive edge of the clock.
```

Figure 3: PyHDL-Eval Problem 13.6 Specification

Figure 3 shows an example specification (see Appendix 7 for additional examples). Each specification includes the same two introductory sentences. Unlike prior work that includes Verilog-specific interface syntax [21, 24], PyHDL-Eval uses a high-level bulleted list of the module's port-based interface. This enables the exact same specification to target many different languages each with their own interface syntax (similar to HDLEval [40]). For parameterized modules, parameters are included in this list and port bitwidths are specified in terms of these parameters. Specifications avoid permitting undefined behavior when possible. All sequential problems include a final sentence specifying that sequential logic is triggered on the positive edge of the clock. For those problems with reset, the specification clearly describes the reset value and that the module should use an active-high synchronous reset.

Specifications that include truth tables, Karnaugh map tables, FSM transition tables, FSM diagrams, and/or waveforms use a text-

based format similar to VerilogEval [21]. Specifications that include a wiring diagram (e.g., LFSR problems) include a text-based diagram with an explanation. Precisely specifying the exact desired behavior for sequential problems is challenging. For example, what exact cycle should a rising edge detector change its output relative to the rising edge on the input? Does a two-stage pipeline include just registered inputs or registered inputs and outputs? To avoid ambiguity, specifications for problem categories 14, 15, 18, and 19 include an example execution trace that illustrates the exact desired input/output behavior over approximately ten cycles.

The problem specification is combined with 0–3 ICL examples, delimiter/explanation suffix, and a system message to create the full prompt. Each ICL example includes a specification and a solution in the exact same format as the actual problem. The same ICL examples are used across all problems. The ICL examples include: an 8-bit combinational incrementer; an 8-bit registered incrementer with active-high synchronous reset; and an N-bit registered incrementer parameterized by the bitwidth (see Appendix 8 for the exact ICL example specifications). While each ICL specification is exactly the same for Verilog and every Python-embedded DSL, the ICL solutions are specific to the target languages. The following delimiter/explanation suffix is appended to all problem specifications:

```
Enclose your code with <CODE> and </CODE>. Only output
the code snippet and do NOT output anything else.
```

The following system message template is used with *language* set to either `Verilog`, `PyMTL3`, `PyRTL`, `MyHDL`, `Migen`, or `Amaranth`:

```
You are a language RTL designer that only writes
code using correct language syntax.
```

3.2 Functional Testing

In addition to the specification, each problem also includes a Verilog reference solution, Verilog test bench, and Python test script. The exact same directed test cases and similar random test cases are used in both the Verilog test bench and Python test script (see Table 1 for the number of test cases per problem). Verilog test benches use a custom lightweight unit testing framework to compare the Verilog reference solution to an LLM-generated Verilog module. Python test scripts use PyHDL-Eval's new testing library (see Figure 4). An instance of the `Config` dataclass is used to specify the module's ports, parameters, and other configuration information. Each test case calls `run_sim` which applies a list of test vectors to both a Verilog reference and the LLM-generated module. The outputs are compared every cycle to ensure they match. The Python Hypothesis library is used for property-based testing with smart random search strategies and automatic failing test case minimization [17, 25].

The Verilog RTL reference is tested against itself to manually verify the test cases cover the specified behavior and the reference correctly implements the specification. The Verilog test bench and Python test scripts are then used to functionally verify the correctness of the LLM-generated modules compared to the Verilog reference solution. While all of the Python-embedded DSLs support translating Python into Verilog RTL, PyHDL-Eval currently only evaluates the LLM-generated modules using each DSL's pure-Python simulation engine. This means non-translatable Python-embedded DSL modules might pass functional testing, just as non-synthesizable Verilog modules might also pass functional testing.

```python
from pyhdl_eval.cfg  import Config, InputPort, OutputPort
from pyhdl_eval.core import run_sim
from pyhdl_eval      import strategies as pst
from hypothesis      import settings, given
from hypothesis      import strategies as st

config = Config(
  ports = [
    ( "clk",   InputPort (1) ),
    ( "ld",    InputPort (1) ),
    ( "in_",   InputPort (3) ),
    ( "out",   OutputPort(3) ),
    ( "done",  OutputPort(1) ),
  ],
  dead_cycles=1,
  dead_cycle_inputs=(1,0),
)

def test_case_done( pytestconfig ):
  run_sim( pytestconfig, __file__, config,
  [ (1,0), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), ])

...

def test_case_count7( pytestconfig ):
  run_sim( pytestconfig, __file__, config,
  [ (1,7), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), ])

def test_case_multi_ld( pytestconfig ):
  run_sim( pytestconfig, __file__, config,
  [ (1,5), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0),
    (0,0), (1,3), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), ])

@settings(deadline=None,max_examples=20)
@given(st.lists(st.tuples(pst.bits(1),pst.bits(3)),min_size=20))
def test_case_random( pytestconfig, test_vectors ):
  run_sim( pytestconfig, __file__, config, test_vectors )
```

Figure 4: PyHDL-Eval Problem 13.6 Python Test Script

A key challenge is functional testing of many different Python-embedded DSLs using the same Python test scripts. Each DSL has different syntax and semantics for hardware modeling, simulator construction, and test execution. PyHDL-Eval provides a unified DSL interface for constructing a model from a Python source file, resetting the simulator, reading/writing module ports, and ticking the simulator. Each DSL required only 60–141 lines of Python to implement this interface. This unified interface greatly simplifies extending the framework to support future Python-embedded DSLs.

3.3 Workflow Orchestration Scripts

The PyHDL-Eval framework also includes workflow orchestration scripts to automate the process of evaluating LLMs for Python-embedded DSLs. A *generate script*: (1) creates full prompts from a specification, ICL examples, delimiter/explanation suffix, and system message; (2) uses the `langchain` library to send a query to various cloud-based LLM services; and (3) post-processes the response to extract the LLM-generated module based on `<CODE>` delimiters or triple back-ticks. An *analysis script* processes log files from functional testing to classify errors and outputs a concise summary in both plain text and CSV formats (see Appendix 10). Automated error classification enables much deeper insights into LLM understanding compared to simple pass/fail analysis. A *Makefile* automates the entire generation, testing, and analysis process.

4 PyHDL-Eval Case Study

In this section, we use PyHDL-Eval to evaluate five LLMs targeting Verilog and the five Python-embedded DSLs described in Section 2. We evaluate the instruction-tuned variants of CodeGemma 7B (a smaller open LLM trained on code), Llama3 8B (a smaller general-purpose open LLM); Llama3 70B (one of the most capable general-purpose open LLMs); GPT4 (version 0613; a strong closed LLM); and GPT4 Turbo (1106-preview; one of the most capable general-purpose closed LLMs). This case study demonstrates the PyHDL-Eval framework and evaluates the state-of-the-art for LLMs on specification-to-RTL tasks targeting Python-embedded DSLs.

4.1 Evaluating the Effect of In-Context Learning

We first focus on using the PyHDL-Eval framework to understand the effect of ICL targeting Verilog and PyMTL3. We evaluate all five LLMs with 0–3 ICL examples using the approach described in Section 3.1. Similar to prior work [21], we generate 20 samples per problem and calculate the pass@1 rate which is simply the fraction of samples that successfully pass all functional tests using the testing methodology described in Section 3.2. We then average these pass rates across all 168 problems to calculate an overall pass rate. Table 2 shows the results and Figure 5 shows the trends in pass rate vs. the number of ICL examples.

ICL for Verilog – ICL improves the pass rate for smaller LLMs targeting Verilog; CodeGemma 7B and Llama3 8B improve from 14.9% and 21.5% without any ICL examples to 32.7% and 36.6% with three ICL examples, respectively. Much of the benefit comes from the first two ICL examples. However, the third ICL example (i.e., a parameterized module) does have a significant impact on the smaller LLMs' performance on problem category 9 (see Appendix 9 for detailed data). ICL also produces significant improvements in pass rate for two larger LLMs; Llama3 70B and GPT4 improve from 57.5% and 60.2% without any ICL examples to 63.2% and 69.9% with three ICL examples, respectively. With these two LLMs, the second ICL example enables these LLMs to significantly improve their pass rate on sequential problems (see Appendix 9 for detailed data). However, the third ICL example has little impact on problem category 9, likely because these LLMs are already performing relatively well on parameterized problems.

ICL for PyMTL3 – Without ICL, CodeGemma 7B, Llama3 8B/70B, and GPT4 Turbo have a pass rate of ≤2%. Examining the samples reveals that these LLMs have a conceptual understanding of PyMTL3, but consistently make basic mistakes (e.g., import a non-existent PyMTL3 library or use deprecated syntax). Even one ICL example reduces the occurrences of these basic mistakes and enables these LLMs to significantly improve their pass rate to 15.7–33.2%. Additional ICL examples have a relatively modest impact on the smaller LLMs and a larger impact on the larger LLMs including GPT4. The second ICL example has a noticeable impact on the ability of GPT4 and GPT4 Turbo to solve sequential problems (see Appendix 9 for detailed data). These results perhaps illustrate the larger models' stronger instruction following capabilities.

ICL Summary – ICL either has no impact or improves the pass rate for LLMs targeting Verilog. ICL is critical to enabling reasonable pass rates for LLMs targeting PyMTL3. Thus the remainder of the case study will always use three ICL examples.

DSL	Num ICL Ex	Code Gemma 7B	Llama3 8B	Llama3 70B	GPT4	GPT4 Turbo
Verilog	0	14.9%	21.5%	57.5%	60.2%	71.0%
Verilog	1	26.6%	33.0%	56.7%	62.4%	70.1%
Verilog	2	32.2%	34.7%	62.4%	69.3%	72.7%
Verilog	3	32.7%	36.6%	63.2%	69.9%	72.2%
PyMTL3	0	0.0%	0.0%	0.6%	13.2%	2.0%
PyMTL3	1	20.0%	15.7%	27.3%	33.1%	33.2%
PyMTL3	2	23.5%	14.3%	31.9%	43.5%	39.2%
PyMTL3	3	23.9%	15.9%	33.0%	47.0%	43.6%
PyRTL	3	14.7%	6.9%	24.4%	22.9%	29.8%
MyHDL	3	32.8%	23.1%	53.8%	61.0%	62.0%
Migen	3	22.9%	21.4%	40.2%	46.0%	47.3%
Amaranth	3	22.4%	20.7%	45.2%	49.4%	56.0%

Table 2: Average Pass Rate – Pass rate is averaged across 20 samples per problem across all 168 PyHDL-Eval problems.

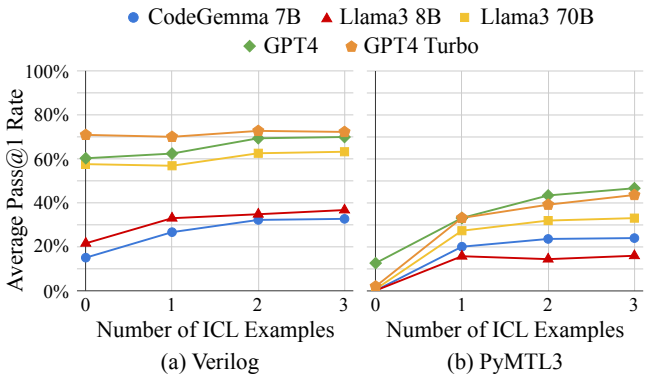

Figure 5: Average Pass Rate vs. Number of ICL Examples

4.2 Evaluating LLMs for Verilog and Python-Embedded DSLs

We now use the PyHDL-Eval framework to evaluate the five LLMs targeting Verilog, PyMTL3, PyRTL, MyHDL, Migen, and Amaranth with three ICL examples. Table 2 and Figure 6 show the results. Table 3 shows the pass rate for each LLM targeting each language organized by problem category.

Verilog – As expected, larger models have higher pass rates compared to smaller models with GPT4 Turbo achieving the highest pass rate of 72.2%. Llama3 8B outperforms CodeGemma 7B even though CodeGemma 7B is specifically trained on code, perhaps reflecting the key challenge of applying LLMs trained using software programming languages to tasks requiring hardware description languages. Table 3 shows that all LLMs struggle on sequential arithmetic and arbiter problems. A more detailed analysis shows all LLMs also struggle on boolean logic problems (truth tables, K-maps, waveforms) involving more than two inputs and the more complicated sequential FSM problems. Smaller LLMs additionally struggle on encoder/decoder problems, all FSM problems, and all sequential boolean logic problems. These results confirm that the PyHDL-Eval framework targeting Verilog serves as a reasonable baseline when exploring Python-embedded DSLs.

	Problem Category		Num Probs	CodeGemma 7B						Llama3 8B						Llama3 70B						GPT4						GPT4 Turbo					
				Ver	PyM	PyR	MyH	Mig	Ama	Ver	PyM	PyR	MyH	Mig	Ama	Ver	PyM	PyR	MyH	Mig	Ama	Ver	PyM	PyR	MyH	Mig	Ama	Ver	PyM	PyR	MyH	Mig	Ama
1	comb	const	4	86	100	69	0	100	98	85	95	25	0	100	100	100	100	76	0	100	100	100	98	81	0	100	100	100	100	99	0	100	100
2	comb	wires	8	49	36	26	24	49	52	58	28	24	27	43	29	67	44	49	39	73	61	92	41	48	46	64	56	89	76	46	58	86	76
3	comb	gates	13	62	49	35	53	57	53	65	45	5	45	54	48	96	80	42	84	84	90	99	92	63	83	93	91	99	88	57	76	96	90
4	comb	bool	15	31	40	8	36	28	32	35	38	0	44	32	31	49	55	33	54	47	51	57	52	46	60	56	55	56	58	46	61	57	60
5	comb	mux	9	58	82	7	84	23	8	53	19	0	45	13	8	96	83	63	100	51	73	98	95	12	99	47	66	85	92	52	98	62	89
6	comb	codes	14	6	17	0	26	10	9	16	4	0	9	3	4	38	23	2	32	26	13	64	34	3	63	29	46	68	48	46	65	21	54
7	comb	arith	17	47	24	25	48	32	31	46	14	16	41	24	28	68	37	36	67	56	59	76	30	38	65	49	48	84	27	46	76	62	66
8	comb	fsm	9	0	1	0	7	0	0	3	9	0	9	1	0	41	66	0	75	27	20	81	78	0	81	44	53	83	43	61	78	26	66
9	comb	param	8	23	31	11	20	24	23	40	21	13	10	20	4	62	49	24	42	44	57	83	45	16	55	37	44	88	43	33	64	49	67
10	seq	gates	9	73	13	28	67	48	51	81	2	0	41	50	56	95	0	23	87	67	74	89	59	19	87	77	66	99	42	0	89	57	76
11	seq	bool	5	0	1	0	8	0	0	8	0	0	11	4	1	33	2	3	22	10	6	43	53	11	58	47	42	52	11	1	51	31	32
12	seq	sreg	7	41	0	0	47	0	0	20	0	0	6	0	0	64	0	0	3	2	49	59	2	0	18	24	36	71	0	0	41	22	37
13	seq	count	9	39	18	25	49	17	22	46	3	14	11	14	23	82	2	8	82	27	29	83	46	6	82	29	38	89	39	0	90	61	46
14	seq	edge	5	11	6	9	2	5	4	22	8	11	3	17	28	41	1	32	41	40	39	46	50	25	67	43	41	62	51	4	66	56	54
15	seq	arb	5	4	0	0	0	0	0	3	0	0	0	0	0	15	0	0	0	0	0	17	0	0	1	0	0	10	0	0	2	0	0
16	seq	fsm	13	0	0	0	1	0	0	1	0	0	0	0	0	51	3	2	52	6	17	30	22	0	42	6	5	29	12	0	36	3	12
17	seq	mem	6	36	4	0	46	2	0	36	0	0	28	0	0	91	0	0	68	16	26	92	15	0	81	58	69	90	2	0	75	13	13
18	seq	arith	4	25	0	0	0	0	0	24	0	0	0	0	0	25	0	5	4	1	0	35	3	3	3	5	14	21	16	0	6	5	3
19	seq	pipe	8	24	16	36	27	26	25	47	12	31	36	33	33	63	23	58	45	42	51	58	55	39	68	47	54	63	46	7	41	53	61
	Average Pass Rate			33	24	15	33	23	22	37	16	7	23	21	21	63	33	24	54	40	45	70	47	23	61	46	49	72	44	30	62	47	56

Table 3: Average Pass Rate with Three ICL Examples Organized by Problem Category – Values are average pass rate as a percentage for corresponding LLM, language, and problem category. Ver = Verilog; PyM = PyMTL3; MyH = MyHDL; Mig = Migen; Ama = Amaranth.

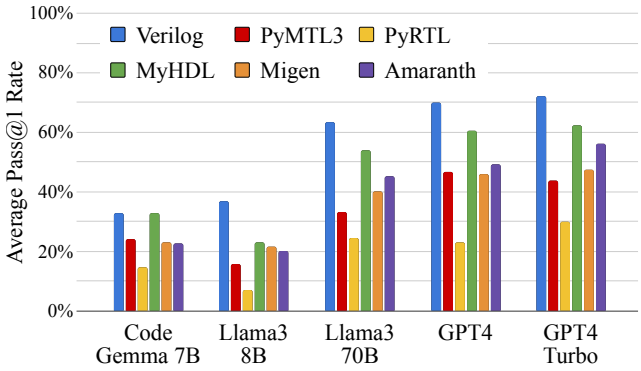

Figure 6: Average Pass Rate with Three ICL Examples

Python-Embedded DSLs – Again, larger models have higher pass rates compared to smaller models. Notice that CodeGemma 7B now outperforms Llama3 8B on all five Python-embedded DSLs, perhaps reflecting CodeGemma 7B's strength in targeting Python code generation tasks. Table 3 shows that although all LLMs perform better on the combinational problems compared to the sequential problems, any given LLM demonstrates different conceptual understanding for each DSL. For example, the LLMs struggle on the constant logic problems for MyHDL, but have no issue for the other DSLs. Llama3 70B and GPT4 struggle on combinational FSM problems for PyRTL, but have more success for the other DSLs. All five LLMs achieve the highest pass rate when targeting MyHDL with GPT4 Turbo achieving a pass rate of 62.0%. The PyHDL-Eval framework's ontological approach and automated failure classification can help explain this result. For PyMTL3, PyRTL, Migen, and Amaranth, LLMs frequently declare explicit clock and reset signals even though the ICL examples clearly demonstrate these ports are implicit. MyHDL uses explicit clock and reset signals and thus is perhaps a better match for the LLMs' conceptual understanding of hardware modeling (likely grounded in its Verilog training data).

Python-Embedded DSLs vs. Verilog – One might assume that LLMs will perform better targeting Python-embedded DSLs, since LLM training data includes significantly more Python vs. Verilog training data. This case-study provides rigorous data to challenge this assumption. All five LLMs achieve higher pass rates on Verilog compared to all five DSLs (except for CodeGemma7B on MyHDL). Table 3 shows that LLMs generally struggle on the same problem categories (e.g., arbiters, sequential arithmetic), but ultimately struggle more when targeting DSLs. What the DSLs gain by using a popular host language, they then loose through their use of a domain-specific library with unique syntax and semantics. MyHDL comes the closest to Verilog (62.0% vs. 72.2% using GPT4 Turbo). Some of this gap is due to constant logic problems but GPT4 Turbo is also making relatively basic mistakes related to bit slicing and reduction operators (see Appendix 10 for detailed results).

5 Conclusions

PyHDL-Eval is the first LLM evaluation framework for hardware design targeting multiple Python-embedded DSLs. PyHDL-Eval includes 168 problem specifications, Verilog reference solutions, Verilog test benches, and Python test scripts, and workflow orchestration scripts. We have used PyHDL-Eval to demonstrate the critical need for ICL when generating Python-embedded DSL modules, and we have also conducted the first rigorous comparative analysis of LLMs targeting both Verilog and various DSLs. PyHDL-Eval can serve as a framework for future research potentially exploring the impact of more advanced ICL and/or supervised fine-tuning to close the gap between Verilog and Python-embedded DSLs.

Acknowledgments

The authors thank Teodor-Dumitru Ene for his guidance on effectively querying LLMs and his valuable feedback on this work. The authors also thank the NVIDIA Applied Deep Learning Research (ADLR) and the NVIDIA Inference Microservices (NIM) teams for supporting the infrastructure that enabled access to the large-language models used in this paper.

References

[1] A. Allam and M. Shalan. RTL-Repo: A Benchmark for Evaluating LLMs on Large-Scale RTL Design Projects. *Computing Research Repository (CoRR)*, arXiv:2405.17378, May 2024.

[2] Amaranth HDL. Online Webpage, 2024 (accessed May 2024). https://github.com/amaranth-lang/amaranth.

[3] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards. Cλash: Structural Descriptions of Synchronous Hardware Using Haskell. *Euromicro Conf. on Digital System Design (DSD)*, Sep 2010.

[4] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanović. Chisel: Constructing Hardware in a Scala Embedded Language. *Design Automation Conf. (DAC)*, Jun 2012.

[5] P. Bellows and B. Hutchings. JHDL: An HDL for Reconfigurable Systems. *Symp. on FPGAs for Custom Computing Machines (FCCM)*, Apr 1998.

[6] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware Design in Haskell. *Int'l Conf. on Functional Programming (ICFP)*, Sep 1998.

[7] J. Blocklove, S. Garg, R. Karri, and H. Pearce. Chip-Chat: Challenges and Opportunities in Conversational Hardware Design. *Int'l Symp. on Machine Learning for CAD (MLCAD)*, Sep 2023.

[8] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language Modles are Few-Shot Learners. *Conf. on Neural Information Processing Systems (NeurIPS)*, Dec 2020.

[9] K. Chang, K. Wang, N. Yang, Y. Wang, D. Jin, W. Zhu, Z. Chen, C. Li, H. Yan, Y. Zhou, Z. Zhao, Y. Cheng, Y. Pan, Y. Liu, M. Wang, S. Liang, Y. Han, H. Li, and X. Li. Data is All You Need: Finetuning LLMs for Chip Design via an Automated Design-Data Augmentation Framework. *Design Automation Conf. (DAC)*, Jun 2024.

[10] K. Chang, Y. Wang, H. Ren, M. Wang, S. Liang, Y. Han, H. Li, and X. Li. ChipGPT: How Far Are We From Natural Language Hardware Design? *Computing Research Repository (CoRR)*, arXiv:2305.14019, May 2023.

[11] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood. A Pythonic Approach for Rapid Hardware Prototyping and Instrumentation. *Int'l Conf. on Field Programmable Logic (FPL)*, Sep 2017.

[12] cocotb: A Coroutine-Based Cosimulation Library for Writing VHDL and Verilog Testbenches in Python. Online Webpage, 2024 (accessed May 2024). https://github.com/cocotb/cocotb.

[13] J. Decaluwe. MyHDL: A Python-Based Hardware Description Language. *Linux Journal*, Nov 2004.

[14] D. Durst, M. Feldman, D. Huff, D. Akeley, R. Daly, G. L. Bernstein, M. Patrignani, K. Fatahalian, and P. Hanrahan. Type-Directed Scheduling of Streaming Accelerators. *Conf. on Programming Language Design and Implementation (PLDI)*, Jun 2020.

[15] Y. Fu, Y. Zhang, Z. Yu, S. Li, Z. Ye, C. Li, C. Wan, and Y. C. Lin. GPT4AIChip: Towards Next-Generation AI Accelerator Design Automation via Large Language Models. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2023.

[16] Google. CodeGemma: Open Code Models Based on Gemma. Google White Paper, May 2024. https://goo.gle/codegemma.

[17] S. Jiang, Y. Ou, P. Pan, K. Cheng, Y. Zhang, and C. Batten. PyH2: Using PyMTL3 to Create Productive and Open-Source Hardware Testing Methodologies. *IEEE Design and Test of Computers*, 40(4):53–61, Apr 2021.

[18] S. Jiang, P. Pan, Y. Ou, and C. Batten. PyMTL3: A Python Framework for Open-Source Hardware Modeling, Generation, Simulation, and Verification. *IEEE Micro*, 40(4):58–66, Jul/Aug 2020.

[19] F. Kermarrec, S. Bourdeauducq, J.-C. L. Lann, and H. Badier. LiteX: An Open-Source SoC Builder and Library Based on Migen Python DSL. *Workshop on Open-Source Design Automation (OSDA)*, Mar 2019.

[20] M. Liu, T.-D. Ene, R. Kirby, C. Cheng, N. Pinckney, R. Liang, J. Alben, H. Anand, S. Banerjee, I. Bayraktaroglu, B. Bhaskaran, B. Catanzaro, A. Chaudhuri, S. Clay, B. Dally, L. Dang, P. Deshpande, S. Dhodhi, S. Halepete, E. Hill, J. Hu, S. Jain, A. Jindal, B. Khailany, G. Kokai, K. Kunal, X. Li, C. Lind, H. Liu, S. Oberman, S. Omar, G. Pasandi, S. Pratty, J. Raiman, A. Sarkar, Z. Shao, H. Sun, P. P. Suthar, V. Tej, W. Turner, K. Xu, and H. Ren. ChipNeMo: Domain-Adapted LLMs for Chip Design. *Computing Research Repository (CoRR)*, arXiv:2311.00176, Oct 2023.

[21] M. Liu, N. Pinckney, B. Khailany, and H. Ren. VerilogEval: Evaluating Large Language Models for Verilog Code Generation. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2023.

[22] S. Liu, W. Fang, Y. Lu, Q. Zhang, H. Zhang, and Z. Xie. RTLCoder: Outperforming GPT-3.5 in Design RTL Generation with Our Open-Source Dataset and Lightweight Solution. *Computing Research Repository (CoRR)*, arXiv:2312.08617, Dec 2023.

[23] D. Lockhart, G. Zibrat, and C. Batten. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014.

[24] Y. Lu, S. Liu, Q. Zhang, and Z. Xie. RTLLM: An Open-Source Benchmark for Design RTL Generation with Large Language Models. *Asia and South Pacific Design Automation Conf. (ASP-DAC)*, Jan 2024.

[25] D. R. MacIver, Z. Hatfield-Dodds, and many other contributors. Hypothesis: A New Approach to Property-Based Testing. *Journal of Open-Source Software (JOSS)*, 4(43), Nov 2019.

[26] Meta. Introducing Meta Llama 3: The Most Capable Openly Available LLM to Date. Online Webpage, Apr 2024 (accessed May 2024). https://ai.meta.com/blog/meta-llama-3.

[27] R. Nigam, P. H. A. de Amorim, and A. Sampson. Modular Hardawre Design with Timeline Types. *Conf. on Programming Language Design and Implementation (PLDI)*, Jun 2023.

[28] N. Nikhil. Bluespec System Verilog: Efficient, Correct RTL from High-Level Specifications. *Int'l Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*, Jun 2004.

[29] OpenAI. New Models and Developer Products Announced at DevDay. Online Webpage, Nov 2024 (accessed May 2024). https://openai.com/index/new-models-and-developer-products-announced-at-devday.

[30] OpenAI et al. GPT-4 Technical Report. *Computing Research Repository (CoRR)*, arxiv:2303.08774, Mar 2023.

[31] M. Orenes-Vera, M. Martonosi, and D. Wentzlaff. Using LLMs to Facilitate Formal Verification of RTL. *Computing Research Repository (CoRR)*, arXiv:2309.09437, Sep 2023.

[32] O. Port and Y. Etsion. DFiant: A Dataflow Hardware Description Language. *Int'l Conf. on Field Programmable Logic (FPL)*, Sep 2017.

[33] A. Ray, B. Devlin, F. Y. Quah, and R. Yesantharao. HardCaml: An OCaml Hardware Domain-Specific Languaeg for Efficient and Robust Design. *Computing Research Repository (CoRR)*, arXiv:1509.02058, Dec 2023.

[34] SpinalHDL: A Scala-based HDL. Online Webpage, 2024 (accessed May 2024). https://github.com/SpinalHDL/SpinalHDL.

[35] S. Thakur, B. Ahmad, Z. Fan, H. Pearce, B. Tan, R. Karri, B. Dolan-Gavitt, and S. Garg. Benchmarking Large Language Models for Automated Verilog RTL Code Generation. *Design, Automation, and Test in Europe (DATE)*, Apr 2023.

[36] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, and S. Garg. VeriGen: A Large Language Model for Verilog Code Generation. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 29(3):1–31, Apr 2024.

[37] S. Thakur, J. Blocklove, H. Pearce, B. Tan, S. Garg, and R. Karri. AutoChip: Automating HDL Generation Using LLM Feedback. *Computing Research Repository (CoRR)*, arXiv:2311.04887, Nov 2023.

[38] L. Truong and P. Hanrahan. A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity. *Summit on Advances in Programming Languages (SNAPL)*, May 2019.

[39] Y.-D. Tsai, M. Liu, and H. Ren. RTLFixer: Automatically Fixing RTL Syntax Errors with Large Language Models. *Computing Research Repository (CoRR)*, arXivv:2311.16543, Nov 2023.

[40] M. Zakharov, F. R. Kashanaki, and J. Renau. HDLEval Benchmarking LLMs for Multiple HDLs. *Int'l Workshop on LLM-Aided Design (LAD)*, Jun 2024.

[41] Z. Zhang, G. Chadwick, H. McNally, Y. Zhao, and R. Mullins. LLM4DV: Using Large Language Models for Hardware Test Stimuli Generation. *Computing Research Repository (CoRR)*, arXiv:2310.04535, Oct 2023.

[42] R. Zhong, X. Du, S. Kai, Z. Tang, S. Xu, H.-L. Zhen, J. Hao, Q. Xu, M. Yuan, and J. Yan. LLM4EDA: Emerging Progress in Large Language Models for Electronic Design Automation. *Computing Research Repository (CoRR)*, arXiv:2401.12224, Dec 2023.

6 PyHDL-Eval Problem List

Num	Problem Category		Name	Description	SL	ST	RL	TC
1.1	comb	const	zero	output a constant zero	4	48	6	1
1.2	comb	const	one	output a constant one	4	48	6	1
1.3	comb	const	lohi	output LOW and HIGH values	8	88	8	1
1.4	comb	const	32b value	output a constant 0xdeadbeef value	4	56	6	1
2.1	comb	wires	8b passthru	pass through 8-bit signal	6	65	7	2
2.2	comb	wires	16b split	split 16-bit input into 8-bit lower and 8-bit upper	9	125	9	2
2.3	comb	wires	8b bit rev	8-bit bit reverse	7	104	14	2
2.4	comb	wires	100b bit rev	100-bit bit reverse	7	104	11	2
2.5	comb	wires	64b byte rev	64-bit byte reverse	9	120	14	2
2.6	comb	wires	4x2b passthru	pass through 4x 2-bit signals	14	155	16	2
2.7	comb	wires	5x3b to 4x4b	wire 5 3-bit inputs to 4 4-bit inputs	21	274	19	2
2.8	comb	wires	sext	8-bit sign extend to 32-bit output	8	96	7	2
3.1	comb	gates	and	single 2-input AND gate	6	58	8	1
3.2	comb	gates	nor	single 2-input NOR gate	6	58	8	1
3.3	comb	gates	aoi21	single AOI21 gate	9	105	9	1
3.4	comb	gates	oai22	single OAI22 gate	8	83	10	1
3.5	comb	gates	hadd	1-bit half adder	7	63	10	1
3.6	comb	gates	fadd	1-bit full adder	8	68	11	1
3.7	comb	gates	4 input	4-input AND, NAND, OR, NOR	14	129	16	1
3.8	comb	gates	100 input	100-input AND, NAND, OR, NOR	11	117	13	2
3.9	comb	gates	bitwise	4-bit bitwise AND, NAND, OR, NOR	14	168	14	2
3.10	comb	gates	4b pairwise	4-bit compare neighboring bits	16	225	17	1
3.11	comb	gates	100b pairwise	100-bit compare neighboring bits	16	231	11	2
3.12	comb	gates	nl0	natural lang description of ~(in0 ^ in1) & in2	12	153	9	1
3.13	comb	gates	nl1	natural lang description of (~in0 | in1) & (in2 | ~in3)	16	206	10	1
4.1	comb	bool	logic eq0	boolean function f = (~a & b & ~c) | (a & b & c)	8	81	9	1
4.2	comb	bool	logic eq1	boolean function f = ((NOT a) AND b AND (NOT c)) ...	8	84	9	1
4.3	comb	bool	logic eq2	boolean function f = a'bc' + abc	8	72	9	1
4.4	comb	bool	truth tbl0	truth table with 2 inputs (NAND gate)	13	110	15	1
4.5	comb	bool	truth tbl1	truth table with 3 inputs	18	168	20	1
4.6	comb	bool	truth tbl2	truth table with 4 inputs	27	294	29	1
4.7	comb	bool	kmap0	karnaugh map with 2 inputs (NAND gate)	12	114	15	1
4.8	comb	bool	kmap1	karnaugh map with 3 inputs	13	140	20	1
4.9	comb	bool	kmap2	karnaugh map with 4 inputs	16	180	29	1
4.10	comb	bool	waveform0	waveform with 2 inputs (NAND gate)	30	334	15	1
4.11	comb	bool	waveform1	waveform with 3 inputs	41	520	20	1
4.12	comb	bool	waveform2	waveform with 4 inputs	52	746	29	1
4.13	comb	bool	nl lights	natural lang description of a home lighting system	15	185	9	1
4.14	comb	bool	nl ringer	natural lang description of a cellphone ringer	13	174	10	1
4.15	comb	bool	nl thermostat	natural lang description of a thermostat	20	261	14	1
5.1	comb	mux	1b 2to1	1-bit 2-to-1 mux	10	116	9	1
5.2	comb	mux	1b 8to1	1-bit 8-to-1 mux	16	158	26	2
5.3	comb	mux	1b 1to8	1-bit 1-to-8 demux	17	168	34	1
5.4	comb	mux	4b 8to1	4-bit 8-to-1 mux	16	184	26	2
5.5	comb	mux	4b 1to8	4-bit 1-to-8 demux	17	198	34	2
5.6	comb	mux	4b 5to1	4-bit 5-to-1 mux	14	174	21	3
5.7	comb	mux	4b 1to5	4-bit 1-to-5 demux	15	190	26	3
5.8	comb	mux	1b 128to1	1-bit 128-to-1 mux	9	131	8	2
5.9	comb	mux	1b 1to128	1-bit 1-to-128 demux	10	139	11	2
6.1	comb	codes	enc 4to2	one-hot 4-to-2 binary encoder	8	119	15	2
6.2	comb	codes	enc 16to4	one-hot 16-to-4 binary encoder	8	120	27	3
6.3	comb	codes	dec 2to4	binary 2-to-4 one-hot decoder	7	101	14	1
6.4	comb	codes	dec 4to16	binary 4-to-16 one-hot decoder	7	102	26	1
6.5	comb	codes	penc 4to2	one-hot 4-to-2 binary priority encoder	9	138	26	1
6.6	comb	codes	penc 16to4	one-hot 16-to-4 binary priority encoder	9	139	27	2
6.7	comb	codes	bin2gcode	4-bit binary to gray code encoder	6	72	26	1
6.8	comb	codes	gcode2bin	4-bit gray code to binary decoder	6	72	26	1
6.9	comb	codes	bin2bcd	4-bit binary to 2-digit BCD encoder	8	110	20	1
6.10	comb	codes	bcd2bin	2-digit BCD to 4-bit binary decoder	11	152	18	3
6.11	comb	codes	bin2ascii	4-bit binary to 2-digit ASCII encoder	9	128	21	1
6.12	comb	codes	ascii2bin	2-digit ASCII to 4-bit binary decoder	10	144	19	3
6.13	comb	codes	8b parity	8-bit even parity generator	6	67	7	2
6.14	comb	codes	100b parity	100-bit even parity generator	6	67	7	3
7.1	comb	arith	8b add	8-bit adder without carry in/out	8	95	8	4
7.2	comb	arith	8b add carry	8-bit adder with carry in/out	12	140	10	4
7.3	comb	arith	8b sub	8-bit subtractor	8	97	8	4
7.4	comb	arith	8b popcount	8-bit popcount	7	85	8	2
7.5	comb	arith	100b popcount	100-bit popcount	7	87	11	3
7.6	comb	arith	8b shifter	8-bit left/right shifter w/ op input	12	143	14	3
7.7	comb	arith	8b sra	8-bit arithmetic right shift	8	99	8	3
7.8	comb	arith	8b rotator	8-bit left/right rotator w/ op input	12	133	19	3
7.9	comb	arith	8b umul	8-bit unsigned multiplier (16-bit output)	8	95	8	3
7.10	comb	arith	8b smul	8-bit signed multiplier (16-bit output)	8	97	19	5
7.11	comb	arith	8b fxp umul	8-bit UQ4.4 fixed point multplication (8-bit output)	14	180	12	5
7.12	comb	arith	8b madd	8-bit multiply add unit (16-bit add input)	10	122	9	4
7.13	comb	arith	8b ucmp	8-bit unsigned comparator (lt, eq, gt)	13	141	12	4
7.14	comb	arith	8b scmp	8-bit signed comparator (lt, eq, gt)	13	143	12	6
7.15	comb	arith	8b alu	8-bit alu (add, sub, srl, sra, sll, lt, lteq, eq, gteq, gt)	20	293	20	8
7.16	comb	arith	2x8b minmax	2x 8-bit inputs, output the min and max	9	109	18	2
7.17	comb	arith	4x8b sorter	4x 8-bit inputs, sort in increasing order	13	148	32	4
8.1	comb	fsm	4s1i1o mo tbl0	4-state, 1-in, 1-out FSM Moore table (1-hot encoding)	24	269	30	2
8.2	comb	fsm	4s1i1o mo tbl1	4-state, 1-in, 1-out FSM Moore table (bin encoding)	22	234	30	1
8.3	comb	fsm	4s1i1o me tbl	4-state, 1-in, 1-out FSM Mealy table	24	252	30	1
8.4	comb	fsm	4s1i2o mo tbl	4-state, 1-in, 2-out FSM Moore table	23	262	34	1
8.5	comb	fsm	4s2i1o mo tbl	4-state, 2-in, 1-out FSM Moore table	22	274	45	1
8.6	comb	fsm	6s2i2o mo tbl	6-state, 2-in, 2-out FSM Moore table	29	383	55	2
8.7	comb	fsm	4s1i1o mo dia	4-state, 1-in, 1-out FSM Moore diagram	29	298	30	1
8.8	comb	fsm	4s1i1o me dia	4-state, 1-in, 1-out FSM Mealy diagram	29	298	30	1
8.9	comb	fsm	6s2i2o mo dia	6-state, 1-in, 1-out FSM Moore diagram	49	511	51	2
9.1	comb	param	const	parameterized by constant bitwidth and value	7	65	9	2
9.2	comb	param	bit rev	parameterized bitwidth bit reverse	9	131	13	4
9.3	comb	param	nor	parameterized NOR gate by number of inputs	7	74	9	3
9.4	comb	param	2to1 mux	parameterized bitwidth 2-to-1 mux	12	144	11	3
9.5	comb	param	enc	parameterized n-to-m encoder	10	142	22	6
9.6	comb	param	dec	parameterized m-to-n decoder	12	168	13	3
9.7	comb	param	penc	parameterized n-to-m priority encoder	11	161	19	4
9.8	comb	param	rotator	parameterized rotator	15	175	21	6
10.1	seq	gates	1b dff	single 1-bit DFF	7	74	9	1
10.2	seq	gates	1b dffe	single 1-bit DFF w/ enable	9	98	12	2
10.3	seq	gates	1b dffr	single 1-bit DFF w/ reset	10	101	14	3
10.4	seq	gates	8b dff	8-bit DFF	7	79	9	2
10.5	seq	gates	8b dffre	8-bit DFF w/ enable and reset	12	139	15	4
10.6	seq	gates	16b dff byte	16-bit DFF w/ byte enables	15	154	14	4
10.7	seq	gates	nl0	natural lang descr of 2x DFF with in0 & in1	12	144	15	2
10.8	seq	gates	nl1	natural lang descr of 3x DFF with ~(in0 ^ in1) & in2	18	249	18	2
10.9	seq	gates	nl2	natural lang descr of 4x DFF with (~in0 | in1) & ...	22	314	21	2
11.1	seq	bool	truth dff	truth table for D flip-flop	16	167	9	1
11.2	seq	bool	truth tff	truth table for toggle flip-flop	21	229	16	4
11.3	seq	bool	truth jkff	truth table for JK flip-flop	21	229	16	2
11.4	seq	bool	waveform0	waveform for f <= a	32	318	9	2
11.5	seq	bool	waveform1	waveform for f <= ~a	32	318	9	2
12.1	seq	sreg	8b siso	8-bit shift register with serial input, serial output	15	177	17	5
12.2	seq	sreg	8b piso	8-bit shift register with parallel input, serial output	18	218	20	5
12.3	seq	sreg	8b sipo	8-bit shift register with serial input, parallel output	15	178	17	5
12.4	seq	sreg	8b universal	8-bit universal shift register (serial/parallel I/O)	23	269	23	5
12.5	seq	sreg	8b bidir	8-bit bidirectional shift register	22	251	25	7
12.6	seq	sreg	5b lfsr	5-bit Galois linear feedback shift register	27	407	20	3
12.7	seq	sreg	7b lfsr	7-bit Galois linear feedback shift register	28	459	22	3
13.1	seq	count	3b bin up	3-bit binary up counter	11	139	17	4
13.2	seq	count	3b bin up en	3-bit binary up counter with enable	13	157	18	6
13.3	seq	count	3b bin dn	3-bit binary down counter	11	136	17	4
13.4	seq	count	3b bin up dn	3-bit binary up/down counter	17	181	23	8
13.5	seq	count	2b bin sat	2-bit binary saturating up/down counter	19	212	26	9
13.6	seq	count	3b bin var dn	3-bit variable binary down counter	15	182	25	5
13.7	seq	count	4b dec up	4-bit decade up counter	11	135	22	4
13.8	seq	count	timer	minute/second timer	21	255	42	10
13.9	seq	count	clock	hour/minute/second/pm clock	27	323	60	8
14.1	seq	edge	8b any detect	8-bit detect any edge	26	369	11	5
14.2	seq	edge	8b pos detect	8-bit detect just positive edges	24	331	11	5
14.3	seq	edge	8b capture	8-bit capture positive edges	26	351	25	7
14.4	seq	edge	8b any count	8-bit count number of transitions across all bits	29	359	33	9
14.5	seq	edge	8b max switch	8-bit detect maximum switching event	29	368	17	4
15.1	seq	arb	4in priority	4-input variable priority arbiter	39	543	48	9
15.2	seq	arb	4in rotating	4-input oblivious rotational priority arbiter	34	426	47	6
15.3	seq	arb	4in roundrobin	4-input round robin arbiter	34	423	52	6
15.4	seq	arb	4in granthold	4-input grand/hold round-robin arbiter	46	623	63	6
15.5	seq	arb	4in weighted	4-input weighted round robin arbiter	60	812	93	12
16.1	seq	fsm	4s1i1o mo tbl0	4-state, 1-in, 1-out FSM Moore table (1-hot encoding)	25	274	38	4
16.2	seq	fsm	4s1i1o mo tbl1	4-state, 1-in, 1-out FSM Moore table (bin encoding)	25	260	38	4
16.3	seq	fsm	4s1i1o me tbl	4-state, 1-in, 1-out FSM Mealy table	26	278	38	4
16.4	seq	fsm	4s1i2o mo tbl	4-state, 1-in, 2-out FSM Moore table	26	288	42	4
16.5	seq	fsm	4s2i1o mo tbl	4-state, 2-in, 1-out FSM Moore table	25	300	53	4
16.6	seq	fsm	6s2i2o mo tbl	6-state, 2-in, 2-out FSM Moore table	30	378	63	4
16.7	seq	fsm	4s1i1o mo dia	4-state, 1-in, 1-out FSM Moore diagram	31	322	38	4
16.8	seq	fsm	4s1i1o me dia	4-state, 1-in, 1-out FSM Mealy diagram	31	323	38	4
16.9	seq	fsm	6s2i2o mo dia	6-state, 1-in, 1-out FSM Moore diagram	49	503	59	4
16.10	seq	fsm	pattern1	4-state, detect 101	25	267	37	5
16.11	seq	fsm	pattern2	4-state, detect 1110	25	269	40	5
16.12	seq	fsm	stop light	traffic light controller with starting yellow	22	268	46	4
16.13	seq	fsm	ps2	ps2 mouse message detector	30	388	34	6
17.1	seq	mem	8x8b 1r1w rf	1-read, 1-write, 8x8-bit entry reg file	15	185	16	3
17.2	seq	mem	8x8b 1r1w rf fw	1-read, 1-write, 8x8-bit entry reg file with forwarding	16	209	21	4
17.3	seq	mem	8x8b 1r1w rf z	1-read, 1-write, 8x8-bit entry reg file with zero reg	18	218	21	4
17.4	seq	mem	8x8b 1r1w rf pw	1-read, 1-write, 8x8-bit entry reg file with partial writes	20	249	18	4
17.5	seq	mem	8x8b 2r1w rf fwz	2-read, 1-write, 8x8-bit entry reg file with fwd, zero reg	23	289	33	6
17.6	seq	mem	8x8b 1s1w cam	1-search, 1-write, 8x8-bit entry content addr mem	24	346	23	6
18.1	seq	arith	4x1b incr	4-bit bit-serial incrementer	34	404	30	6
18.2	seq	arith	4x1b add	4-bit bit-serial 2-input adder	26	328	31	6
18.3	seq	arith	2x4b add	8-bit nibble-serial 2-input adder	24	310	31	7
18.4	seq	arith	8b accum	8-bit accumulator	23	264	17	6
19.1	seq	pipe	delay 1stage	8-bit input delayed for 1 cycle	21	235	11	2
19.2	seq	pipe	delay 2stage	8-bit input delayed for 2 cycles	22	245	14	2
19.3	seq	pipe	delay 3stage	8-bit input delayed for 3 cycles	24	256	17	2
19.4	seq	pipe	add2 1stage	8-bit 2-input adder with 1 stage	20	234	15	5
19.5	seq	pipe	add2 2stage	8-bit 2-input adder with 2 stages (4-bit add, pipe carry)	38	538	32	6
19.6	seq	pipe	add3 2stage	8-bit 3-input adder with 2 stages	31	442	28	5
19.7	seq	pipe	add4 2stage	8-bit 4-input adder with 2 stages	34	527	35	5
19.8	seq	pipe	minmax4 2stage	8-bit 4-input find min/max with 2 stages	38	554	41	5

SL = Specification lines-of-code for just the problem specification without blank lines, ICL examples, delimiters/explanation suffix, or system message; ST = Specification tokens using the GPT4 Turbo tokenizer without ICL examples, delimiters/explanation suffix, or system message; RL = Verilog reference lines-of-code (LOC) excluding comments and blank lines; TC = number of test cases.

7 PyHDL-Eval Problem Examples

Problem 4.13 Specification

```
Implement a hardware module named TopModule with the following
interface. All input and output ports are one bit unless otherwise
specified.

 - input  dark
 - input  movement
 - input  force_on
 - output turn_on_lights

The module should implement a controller for a home lighting system.
There are three inputs: input dark is one if it is dark outside and
zero if it is light outside; input movement is one if the security
system has detected movement and is otherwise zero; and input
force_on is one if a manual override switch is activated. The lights
should be turned on if it is dark outside and the security system
has detected movement. The lights should always be turned off if it
is light outside regardless of whether the security system has
detected movement. The lights should always be turned on if the
manual override switch is activated.
```

Problem 4.13 Verilog Reference Solution

```
module RefModule
(
  input  logic dark,
  input  logic movement,
  input  logic force_on,
  output logic turn_on_lights
);

  assign turn_on_lights = (dark & movement) | force_on;

endmodule
```

Problem 9.8 Specification

```
Implement a hardware module named TopModule with the following
interface. All input and output ports are one bit unless otherwise
specified.

 - parameter nbits
 - input  in_ (nbits)
 - input  amt (log2(nbits))
 - input  op
 - output out (nbits)

The module should implement a variable rotator which takes as input
a value to rotate (in_) and the rotation amount (amt) and writes the
rotated result to the output (out). The module should be
parameterized by the bitwidth (nbits) of the input and output ports;
nbits can be assumed to be a power of two. The input op specifies
what kind of rotate to perform using the following encoding:

 - 0 : rotate left
 - 1 : rotate right
```

Problem 9.8 Verilog Reference Solution

```
module RefModule
#(
  parameter nbits
)(
  input  logic [        nbits-1:0] in_,
  input  logic [$clog2(nbits)-1:0] amt,
  input  logic                     op,
  output logic [        nbits-1:0] out
);

  logic [(2*nbits)-1:0] temp;

  always @(*) begin
   if ( op == 1'd0 ) begin
     temp = { in_, in_ } << amt;
     out  = temp[(2*nbits)-1:nbits];
   end
   else begin
     temp = { in_, in_ } >> amt;
     out  = temp[nbits-1:0];
   end
  end

endmodule
```

Problem 15.5 Specification

Implement a hardware module named TopModule with the following interface. All input and output ports are one bit unless otherwise specified.

```
 - input  clk
 - input  reset
 - input  reqs   (4 bits)
 - output grants (4 bits)
```

The module should implement a 4-input round-robin priority arbiter with an active-high synchronous reset. Each bit of the input reqs corresponds to one of the four requesters; if reqs[i] is high then this means requester i has a valid request. The output grants is either all zeros (meaning no requesters had a valid request on this cycle) or is a one-hot bit vector indicating which requester won the arbitration.

The internal priority register should be reset to 0001 (i.e., requester 0 has the highest priority) when the reset input is one. When a request wins arbitration it should have the lowest priority in the next cycle of arbitration. For example, if requester 2 wins arbitration, then on the next cycle the priority register should be 1000 (i.e., requester 3 has the highest priority and requester 2 has the lowest priority). If there are no valid requests then the priority should not change. Here is an example execution trace.

```
   cycle | reqs grants
  -------+--------------------
     0   | 1111 0001
     1   | 1111 0010
     2   | 1111 0100
     3   | 1111 1000
     4   | 1111 0001
     5   | 0000 0000
     6   | 1111 0010
     7   | 0000 0000
     8   | 0000 0000
     9   | 1111 0100
```

Assume all sequential logic is triggered on the positive edge of the clock.

Problem 15.5 Verilog Reference Solution

```
module RefModule
(
  input  logic       clk,
  input  logic       reset,
  input  logic [3:0] reqs,
  output logic [3:0] grants
);

  // Register for priority

  logic [3:0] priority_reg, priority_reg_next;

  always @( posedge clk ) begin
    if ( reset )
      priority_reg <= 4'b0001;
    else
      priority_reg <= priority_reg_next;
  end

  // Combinational arbitration logic

  always @(*) begin
    grants = 4'b0000;
    case ( priority_reg )

      4'b0001: begin
        if      ( reqs[0] ) grants = 4'b0001;
        else if ( reqs[1] ) grants = 4'b0010;
        else if ( reqs[2] ) grants = 4'b0100;
        else if ( reqs[3] ) grants = 4'b1000;
      end

      4'b0010: begin
        if      ( reqs[1] ) grants = 4'b0010;
        else if ( reqs[2] ) grants = 4'b0100;
        else if ( reqs[3] ) grants = 4'b1000;
        else if ( reqs[0] ) grants = 4'b0001;
      end

      4'b0100: begin
        if      ( reqs[2] ) grants = 4'b0100;
        else if ( reqs[3] ) grants = 4'b1000;
        else if ( reqs[0] ) grants = 4'b0001;
        else if ( reqs[1] ) grants = 4'b0010;
      end

      4'b1000: begin
        if      ( reqs[3] ) grants = 4'b1000;
        else if ( reqs[0] ) grants = 4'b0001;
        else if ( reqs[1] ) grants = 4'b0010;
        else if ( reqs[2] ) grants = 4'b0100;
      end

      default:
        grants = 4'b0000;

    endcase
  end

  // Combinational priority logic

  always @(*) begin
    priority_reg_next = priority_reg;
    if ( |grants ) begin
      priority_reg_next = { grants[2:0], grants[3] };
    end
  end

endmodule
```

Problem 16.8 Specification

Implement a hardware module named TopModule with the following interface. All input and output ports are one bit unless otherwise specified.

- input clk
- input reset
- input in_
- output state (2 bits)
- output out

The module should implement a Mealy finite-state machine (FSM) with four states. The FSM will be described as a set of FSM transitions with each transition specified using the following format.

```
X (0) --1--> Y
```

In this example, the FSM transitions from state X to state Y when the input is 1 and the output of the FSM should be 0 during this transition. Implement the FSM with the following transitions.

```
A (0) --0--> A
A (1) --1--> B
B (1) --0--> C
B (0) --1--> B
C (0) --0--> A
C (1) --1--> D
D (0) --0--> C
D (0) --1--> B
```

The state machine should use the following state encoding.

- A : 0
- B : 1
- C : 2
- D : 3

The FSM should reset to state A when the reset input is one. The internal state register should be directly connected to the output port state so that this output always reflects the current state of the FSM. Assume all sequential logic is triggered on the positive edge of the clock.

Problem 16.8 Verilog Reference Solution

```
module RefModule
(
  input  logic       clk,
  input  logic       reset,
  input  logic       in_,
  output logic [1:0] state,
  output logic       out
);

  // State Encoding

  parameter A = 2'd0;
  parameter B = 2'd1;
  parameter C = 2'd2;
  parameter D = 2'd3;

  // State Register

  logic [1:0] state_next;

  always @( posedge clk ) begin
    if ( reset )
      state <= A;
    else
      state <= state_next;
  end

  // State Transition Logic

  always @(*) begin
    case ( state )
      A       : state_next = ( in_ == 0 ) ? A : B;
      B       : state_next = ( in_ == 0 ) ? C : B;
      C       : state_next = ( in_ == 0 ) ? A : D;
      D       : state_next = ( in_ == 0 ) ? C : B;
      default : state_next = 0;
    endcase
  end

  // State Output Logic

  always @(*) begin
    case ( state )
      A       : out = ( in_ == 0 ) ? 0 : 1;
      B       : out = ( in_ == 0 ) ? 1 : 0;
      C       : out = ( in_ == 0 ) ? 0 : 1;
      D       : out = ( in_ == 0 ) ? 0 : 0;
      default : out = 0;
    endcase
  end

endmodule
```

Problem 19.6 Specification

Implement a hardware module named TopModule with the following interface. All input and output ports are one bit unless otherwise specified.

```
 - input  clk
 - input  in0   (8 bits)
 - input  in1   (8 bits)
 - input  in2   (8 bits)
 - output out01 (8 bits)
 - output out   (8 bits)
```

The module should implement a two-stage pipelined three-input adder. Just the first addition of in0 and in1 should occur during the first stage, and then the second addition of this first sum and input in2 should occur during the second stage. The final result should be written to the output out at the end of the second stage. The module should write output out01 with the result of the first addition at the end of the first stage.

All transactions should have a two cycle latency (i.e., inputs on cycle i will produce the corresponding sum on cycle i+2). The module should be able to achieve full throughput, i.e., it should be able to accept new inputs every cycle and produce a valid output every cycle. Here is an example execution trace. An X indicates that the output is undefined because there is no reset signal for the pipeline registers.

```
  cycle | in0 in1 in2 out01 out
 -------+----------------------
    0   | 00  00  00  XX    XX
    1   | 01  02  04  00    XX
    2   | 02  03  04  03    00
    3   | 03  04  05  05    07
    4   | 00  00  00  07    09
    5   | 00  00  00  00    0c
    6   | 00  00  00  00    00
```

Assume all sequential logic is triggered on the positive edge of the clock.

Problem 19.6 Verilog Reference Solution

```
module RefModule
(
  input  logic       clk,
  input  logic [7:0] in0,
  input  logic [7:0] in1,
  input  logic [7:0] in2,
  output logic [7:0] out01,
  output logic [7:0] out
);

  // --- Stage 0 ---------------------

  logic [7:0] in0_X0;
  logic [7:0] in1_X0;
  logic [7:0] in2_X0;

  always @( posedge clk ) begin
    in0_X0 <= in0;
    in1_X0 <= in1;
    in2_X0 <= in2;
  end

  logic [7:0] sum01_X0;

  assign sum01_X0 = in0_X0 + in1_X0;
  assign out01 = sum01_X0;

  // --- Stage 1 ---------------------

  logic [7:0] sum01_X1;
  logic [7:0] in2_X1;

  always @( posedge clk ) begin
    sum01_X1 <= sum01_X0;
    in2_X1   <= in2_X0;
  end

  assign out = sum01_X1 + in2_X1;

endmodule
```

8 PyHDL-Eval ICL Examples

ICL Example 1 Specification

```
Implement a hardware module named TopModule with the following
interface. All input and output ports are one bit unless otherwise
specified.

 - input  in_ (8 bits)
 - output out (8 bits)

The module should implement an incrementer which increments the
input by one and writes the result to the output. Assume all values
are encoded as two's complement binary numbers.
```

ICL Example 2 Specification

```
Implement a hardware module named TopModule with the following
interface. All input and output ports are one bit unless otherwise
specified.

 - input  clk
 - input  reset
 - input  in_ (8 bits)
 - output out (8 bits)

The module should implement an 8-bit registered incrementer with an
active-high synchronous reset. The 8-bit input is first registered
and then incremented by one on the next cycle. The internal state
should be reset to zero when the reset input is one. Assume all
values are encoded as two's complement binary numbers. Assume all
sequential logic is triggered on the positive edge of the clock.
```

ICL Example 3 Specification

```
Implement a hardware module named TopModule with the following
interface. All input and output ports are one bit unless otherwise
specified.

 - parameter nbits
 - input     clk
 - input     reset
 - input     in_ (nbits)
 - output    out (nbits)

The module should implement an N-bit registered incrementer with an
active-high synchronous reset. The bitwidth is specified by the
parameter nbits. The N-bit input is first registered and then
incremented by one on the next cycle. The internal state should be
reset to zero when the reset input is one. Assume all values are
encoded as two's complement binary numbers. Assume all sequential
logic is triggered on the positive edge of the clock.
```

9 Results for ICL Organized by Problem Category

Problem			Num	CodeGemma 7B				Llama3 8B				Llama3 70B				GPT4				GPT4 Turbo			
Category			Probs	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
1	comb	const	4	8	71	89	86	30	90	89	85	100	100	100	100	96	100	100	100	100	100	100	100
2	comb	wires	8	9	44	49	49	33	51	57	58	74	71	71	67	83	80	90	92	87	89	88	89
3	comb	gates	13	40	57	57	62	51	56	65	65	99	100	97	96	99	98	100	99	96	99	97	99
4	comb	bool	15	15	28	30	31	24	34	33	35	53	47	47	49	62	58	60	57	59	53	54	56
5	comb	mux	9	15	52	61	58	23	52	50	53	86	95	97	96	73	89	96	98	96	88	89	85
6	comb	codes	14	1	8	8	6	15	18	16	16	39	49	39	38	47	60	60	64	63	70	70	68
7	comb	arith	17	11	43	46	47	32	48	43	46	65	65	66	68	68	77	76	76	86	83	86	84
8	comb	fsm	9	0	1	0	0	0	1	3	3	29	38	36	41	54	67	78	81	91	80	85	83
9	comb	param	8	0	0	0	23	38	32	11	40	68	74	62	62	69	83	84	83	92	91	89	88
10	seq	gates	9	44	40	68	73	44	69	84	81	99	87	98	95	84	78	88	89	99	92	99	99
11	seq	bool	5	16	3	0	0	0	8	13	8	44	39	9	33	39	42	53	43	58	64	53	52
12	seq	sreg	7	21	21	51	41	17	14	19	20	59	35	62	64	54	22	59	59	64	68	71	71
13	seq	count	9	32	28	41	39	18	38	44	46	72	47	85	82	73	64	78	83	88	82	89	89
14	seq	edge	5	9	13	24	11	2	0	23	22	42	42	40	41	41	59	53	46	57	61	61	62
15	seq	arb	5	0	0	3	4	0	1	4	3	0	8	15	15	3	1	12	17	2	7	9	10
16	seq	fsm	13	0	0	1	0	1	4	0	1	10	11	53	51	8	8	19	30	11	9	28	29
17	seq	mem	6	0	30	37	36	16	44	34	36	88	82	88	91	81	84	93	92	84	87	93	90
18	seq	arith	4	23	25	25	25	11	23	23	24	24	20	26	25	23	33	40	35	26	20	23	21
19	seq	pipe	8	33	28	28	24	9	26	44	47	27	48	63	63	57	51	61	58	56	65	63	63
Average Pass Rate				15	27	32	33	21	33	35	37	58	57	62	63	60	62	69	70	71	70	73	72

Verilog Results for ICL Organized by Category – Each value is the average pass rate as a percentage for that LLM with a specific number of ICL examples on the given problem category.

Problem			Num	CodeGemma 7B				Llama3 8B				Llama3 70B				GPT4				GPT4 Turbo			
Category			Probs	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
1	comb	const	4	0	100	98	100	0	99	99	95	4	100	100	100	34	98	100	98	6	94	96	100
2	comb	wires	8	0	38	39	36	0	30	31	28	4	36	42	44	27	47	43	41	8	79	76	76
3	comb	gates	13	0	53	50	49	0	48	39	45	0	81	82	80	45	92	94	92	3	86	91	88
4	comb	bool	15	0	27	41	40	0	37	35	38	0	53	55	55	28	53	55	52	4	55	59	58
5	comb	mux	9	0	73	82	82	0	32	23	19	0	65	81	83	23	94	92	95	1	94	92	92
6	comb	codes	14	0	14	15	17	0	3	2	4	0	22	28	23	14	30	32	34	4	46	53	48
7	comb	arith	17	0	25	24	24	0	19	14	14	1	32	38	37	15	35	28	30	3	29	27	27
8	comb	fsm	9	0	0	2	1	0	6	6	9	0	62	65	66	6	66	71	78	0	36	47	43
9	comb	param	8	0	35	31	31	0	15	0	21	5	4	26	49	14	44	38	45	3	41	43	43
10	seq	gates	9	0	0	13	13	0	0	2	2	0	0	0	0	0	0	49	59	0	0	7	42
11	seq	bool	5	0	0	2	1	0	0	0	0	0	0	0	2	2	0	38	53	0	0	3	11
12	seq	sreg	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0
13	seq	count	9	0	0	13	18	0	0	5	3	0	0	0	2	0	0	36	46	0	0	6	39
14	seq	edge	5	0	0	9	6	0	0	0	8	0	0	1	1	1	0	49	50	0	0	43	51
15	seq	arb	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	seq	fsm	13	0	0	0	0	0	0	0	0	0	0	0	3	0	0	17	22	0	0	4	12
17	seq	mem	6	0	0	1	4	0	0	0	0	0	0	0	0	0	0	1	15	0	0	0	2
18	seq	arith	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	3	0	0	9	16
19	seq	pipe	8	0	0	9	16	0	0	16	12	0	0	23	23	3	0	50	55	0	0	38	46
Average Pass Rate				0	20	24	24	0	16	14	16	1	27	32	33	13	33	44	47	2	33	39	44

PyMTL3 Results for ICL Organized by Category – Each value is the average pass rate as a percentage for that LLM with a specific number of ICL examples on the given problem category.

10 Results per Problem for GPT4 Turbo Targeting Verilog vs. MyHDL

Num	Problem Category		Name	GPT4 Turbo Targeting Verilog With 3 ICL Examples PR	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	GPT4 Turbo Targeting MyHDL With 3 ICL Examples PR	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1.1	comb	const	zero	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0%	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
1.2	comb	const	one	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0%	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
1.3	comb	const	lohi	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0%	R	R	R	R	R	R	p	R	R	R	R	R	R	R	R	R	R	R	R	R
1.4	comb	const	32b value	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0%	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
2.2	comb	wires	16b split	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0%	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
9.1	comb	param	const	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0%	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
3.8	comb	gates	100 input	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	5%	n	n	n	n	n	n	n	n	n	n	n	n	n	.	n	n	n	T	n	
3.11	comb	gates	100b pairwise	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	5%	v	v	v	v	v	v	v	v	v	v	v	v	v	.	v	v	v	v	v	v
2.8	comb	wires	sext	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	15%	R	t	R	R	R	.	R	R	R	.	R	R	.	R	R	R	R	R	R	R
7.2	comb	arith	8b add carry	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	15%	t	t	t	t	.	t	.	t	t	t	.	t	v	t	t	t	t	t	t	t
3.10	comb	gates	4b pairwise	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	20%	v	v	v	.	v	v	v	v	v	.	v	v	v	v	.	a	v	a	.	v
12.2	seq	sreg	8b piso	95%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	R	.	.	.	15%	t	R	t	.	R	t	t	R	t	R	t	t	t	.	t	t	t	.	t	
19.3	seq	pipe	delay 3stage	95%	.	.	.	w	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	15%	R	R	R	.	R	R	R	R	R	.	R	.	R	R	R	R	R	R	R	R
10.6	seq	gates	16b dff byte	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	25%	v	.	v	v	v	v	.	v	.	v	R	v	.	v	v	v	v	t	v	.
18.1	seq	arith	4x1b incr	80%	.	.	.	.	.	.	.	C	.	i	.	.	.	i	.	.	.	i	.	.	5%	t	t	R	R	a	R	t	R	t	R	R	R	R	R	v	R	R	R	R	.
6.13	comb	codes	8b parity	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	40%	n	R	R	n	.	R	.	.	n	R	a	.	.	n	n	.	n	R	.	.
9.3	comb	param	nor	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	40%	.	T	T	.	T	T	T	.	.	n	T	R	.	T	.	.	T	T	T	.
6.2	comb	codes	enc 16to4	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	45%	.	R	.	t	.	R	R	R	.	.	R	R	.	R	.	.	R	.	R	R
17.4	seq	mem	8x8b 1r1w rf pw	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	45%	.	R	R	v	a	v	.	.	.	.	.	v	v	v	v	v	.	.	a	
2.5	comb	wires	64b byte rev	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	50%	R	.	R	.	.	v	v	.	v	.	.	.	v	v	v	.	.	R	.	v
6.11	comb	codes	bin2ascii	90%	.	R	.	.	.	.	.	.	.	.	.	.	R	.	.	.	.	.	.	.	40%	t	R	t	.	t	R	.	.	.	.	R	R	t	.	.	R	.	t	v	t
12.3	seq	sreg	8b sipo	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	55%	R	.	.	.	v	R	R	.	R	.	R	.	.	.	R	R	.	.	R	.
12.4	seq	sreg	8b universal	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	55%	R	.	.	R	.	R	.	R	.	R	.	.	.	R	.	.	R	R	.	.
14.5	seq	edge	8b max switch	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	60%	.	.	.	R	.	.	.	.	.	.	R	R	R	.	R	R	R	.	R	.
7.10	comb	arith	8b smul	85%	.	.	.	.	.	.	.	R	.	.	R	.	.	R	.	.	.	.	.	.	45%	R	.	R	R	.	R	.	a	.	R	R	R	a	.	.	R	.	.	R	.
17.5	seq	mem	8x8b 2r1w rf fwz	70%	R	.	.	.	.	.	.	.	.	.	.	R	R	.	R	p	R	.	.	.	35%	.	R	R	R	R	R	.	.	p	.	R	.	.	R	R	R	R	R	.	.
7.14	comb	arith	8b scmp	40%	.	R	R	.	R	R	R	R	R	.	R	.	R	R	.	.	.	R	.	R	10%	R	R	R	.	R	R	R	R	R	R	R	R	R	R	R	R	.	R	R
12.1	seq	sreg	8b siso	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	70%	.	.	.	.	R	.	.	.	.	.	.	R	.	.	.	R	v	v	.	R	.
16.5	seq	fsm	4s2i1o mo tbl	80%	e	.	.	.	.	.	.	e	.	.	.	e	.	.	.	.	.	e	.	.	50%	t	t	R	.	.	R	R	.	.	.	R	.	.	R	R	.	t	.	.	R
19.8	seq	pipe	minmax4 2stage	55%	.	.	R	R	R	.	.	.	R	R	R	.	R	R	.	.	.	.	S	.	25%	R	R	R	R	.	R	n	.	R	R	R	R	R	R	.	R	R	.	.	R
15.1	seq	arb	4in priority	35%	.	S	.	S	S	S	S	S	S	S	R	.	.	S	S	S	S	.	.	.	5%	R	R	a	R	R	R	R	R	.	T	T	R	T	R	R	p	a	R	a	R
16.3	seq	fsm	4s1i1o me tbl	50%	e	e	.	R	e	r	e	.	C	.	C	.	.	.	.	.	.	e	e	.	20%	.	t	t	R	R	.	R	t	R	R	R	R	.	t	R	t	R	R	R	.
7.15	comb	arith	8b alu	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	75%	.	.	.	.	.	.	.	.	v	.	v	.	v	.	.	v	.	.	.	v
8.8	comb	fsm	4s1i1o me dia	70%	.	.	R	.	R	R	.	R	.	.	S	.	R	.	.	.	.	.	.	.	45%	R	.	R	.	.	.	R	.	.	R	.	.	R	R	R	R	R	.	.	
17.6	seq	mem	8x8b 1s1w cam	90%	w	.	.	.	.	.	.	.	.	.	.	.	.	.	.	w	.	.	.	.	70%	.	.	t	t	.	t	.	.	.	.	R	.	t	R	.	.	.	.	.	.
13.8	seq	count	timer	50%	.	R	.	R	R	.	.	.	R	.	R	R	.	.	R	R	.	R	R	.	30%	R	R	R	R	.	.	R	R	R	R	R	R	.	R	.	R	R	R	.	
8.9	comb	fsm	6s2i2o mo dia	85%	R	.	.	.	.	.	.	.	w	S	.	.	.	.	.	.	.	.	.	.	65%	t	.	.	.	t	.	R	R	.	.	t	.	.	R	t	.	.	.	.	.
9.8	comb	param	rotator	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	80%	.	n	.	.	.	.	R	.	.	a	.	.	.	.	.	.	n	.	.	.
14.2	seq	edge	8b pos detect	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	80%	R	.	.	.	.	.	.	.	R	.	.	.	R	.	.	.	.	.	.	R
19.7	seq	pipe	add4 2stage	60%	.	.	.	R	R	R	.	.	.	R	.	R	R	.	R	R	.	.	.	.	40%	R	R	R	.	v	R	.	.	R	R	.	v	.	R	v	.	R	.	R	.
4.6	comb	bool	truth tbl2	20%	.	R	.	R	R	R	R	R	R	R	R	R	R	R	R	R	.	.	R	.	5%	R	R	R	R	R	R	R	R	.	R	R	R	R	R	R	R	R	R	R	R
7.5	comb	arith	100b popcount	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	85%	.	.	.	t	.	.	T	.	.	.	.	.	.	.	.	T	.	.	.	
9.6	comb	param	dec	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	85%	.	n	n	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	n	
16.9	seq	fsm	6s2i2o mo dia	45%	.	e	e	r	w	R	.	.	r	r	.	.	.	r	e	w	.	R	.	.	30%	t	.	R	t	R	t	t	R	t	.	.	R	.	t	R	R	.	p	t	.
16.13	seq	fsm	ps2 protocol	40%	w	.	.	w	.	.	e	R	R	R	.	R	R	p	.	R	.	.	R	p	25%	.	R	R	R	.	R	p	.	.	R	R	R	R	R	R	.	R	R	R	R
19.4	seq	pipe	add2 1stage	75%	.	.	.	R	.	R	.	R	.	R	R	.	.	.	.	.	.	.	.	.	60%	R	.	R	.	.	.	R	.	.	.	R	.	R	.	.	R	.	.	R	R
16.8	seq	fsm	4s1i1o me dia	30%	e	.	.	.	e	e	R	R	R	.	e	R	.	e	R	.	e	e	w	r	15%	R	R	R	R	T	R	R	R	R	R	n	t	R	R	.	R	.	R	.	R
19.6	seq	pipe	add3 2stage	15%	R	R	R	.	R	R	R	R	R	R	w	R	R	R	R	.	R	R	R	.	0%	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
8.5	comb	fsm	4s2i1o mo tbl	60%	.	.	.	.	.	.	.	R	R	R	R	R	R	R	.	.	.	w	.	.	45%	R	.	.	.	R	R	R	.	R	R	R	R	.	.	.	.	t	.	R	
2.7	comb	wires	6x4b to 4x8b	10%	.	R	R	R	R	R	R	R	R	.	R	R	R	R	R	R	R	R	R	R	0%	n	n	t	n	n	R	R	n	R	n	n	n	n	n	v	R	n	n	n
15.5	seq	arb	4in weighted	10%	S	i	.	S	i	i	i	S	C	S	i	i	S	S	i	.	S	S	w	S	0%	R	R	R	R	R	v	R	R	R	v	R	R	R	R	T	R	R	R	t	R
3.4	comb	gates	oai22	85%	.	.	.	.	R	.	.	.	.	.	.	.	C	.	.	R	.	.	.	.	75%	.	.	R	.	R	.	.	.	R	.	.	.	.	R	.	.	.	.	.	R
3.12	comb	gates	nl0	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	90%	.	.	.	.	.	.	.	.	.	v	.	.	.	.	.	v	.	.	.	.	.
5.2	comb	mux	1b 8to1	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	90%	.	.	.	t	.	.	.	.	.	.	.	.	.	.	.	t	.	.	.	.
6.14	comb	codes	100b parity	60%	R	.	R	.	R	.	R	.	R	R	R	.	.	.	.	.	.	.	R	.	50%	T	T	.	.	.	.	.	.	a	R	R	R	T	.	.	a	n	.	.	R
7.7	comb	arith	8b sra	65%	R	.	.	R	.	.	R	R	R	.	.	.	R	.	.	.	R	.	.	.	55%	.	R	.	R	R	.	.	.	R	.	R	.	R	R	.	.	.	R	.	R
9.7	comb	param	penc	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	90%	.	.	.	.	.	.	.	.	R	.	.	.	.	.	.	.	.	.	.	n
10.9	seq	gates	nl2	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	90%	.	v	.	.	.	.	.	.	.	.	.	.	n	.	.	.	.	.	.	.
11.2	seq	bool	truth tff	85%	.	.	.	.	.	.	.	R	.	.	.	.	.	R	.	.	R	.	.	.	75%	R	.	.	.	.	.	.	R	.	R	R	.	.	.	.	.	.	.	R	
19.1	seq	pipe	delay 1stage	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	90%	.	.	.	.	.	.	.	R	.	.	.	.	R	.	.	.	.	.	.	.
2.3	comb	wires	8b bit rev	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	95%	.	.	.	.	.	.	.	a	.	.	.	.	.	.	.	.	.	.	.	.
3.9	comb	gates	bitwise	100%	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	95%	.	.	.	.	.	v	.	.	.	.	.	.	.	.	.	.	.	.	.	.

Results of 20 samples per problem for GPT4 Turbo targeting Verilog and MyHDL with three ICL examples. PR = average pass rate across 20 samples for corresponding problem. Problems are first sorted by the difference in pass rate between Verilog and MyHDL; the 62 problems where MyHDL does the worst compared to Verilog are shown. Outcome for each of the 20 samples is shown with a single character. Key for Verilog outcomes: . = sample passes all functional tests; e = need explicit cast when assigning an enum using a ternary statement; i = incorrect use of an initial block; w = incorrect use of reg/wire; r = incorrect use of asynchronous reset; p = unknown port name; C = unclassified non-syntax compile-time error; S = unclassified syntax error; R = failing functional test case. Key for MyHDL outcomes: . = sample passes all functional tests; t = type error; p = MyHDL not imported correctly; a = attribute error; v = value error; n = name error; S = unclassified syntax error; R = failing functional test case; T = timeout during functional testing.

11 Artifact Appendix

11.1 Abstract

This appendix describes how reproduce all of the results in the paper. Our approach is to use a Docker image and a README hosted on Zenodo:

- https://zenodo.org/records/13117553

The README provides detailed step-by-step instructions, while the Docker image includes:

- pre-installed binaries for all tools (GCC 13.2.0, Make 4.3, Icarus Verilog simulator 12.0, Verilator Verilog simulator 5.020, Python 3.12.3);
- pre-installed Python packages for all five Python-embedded DSLs (PyMTL3, PyRTL 0.11.1, MyHDL 0.11.45, Migen 0.9.2, Amaranth 0.4.5);
- source code for the PyHDL-Eval framework (Verilog reference solutions, Verilog test benches, Python test scripts, workflow orchestration scripts);
- RTL modules pre-generated using all five LLMs (CodeGemma 7B, Llama3 8B/70B, GPT4, GPT4 Turbo); and
- scripts for running simulations and generating all of the tables in the paper.

The PyHDL-Eval framework includes scripts to query all five LLMs to regenerate all Verilog, PyMTL3, MyHDL, Migen, and Amaranth RTL modules. However, regenerating these modules is not included as part of the artifact evaluation for three reasons.

- Regenerating the modules is not deterministic. The exact same query can produce different responses especially since the experiments in the paper use a temperature > 0 to better characterize each LLMs fundamental reasoning. Even for LLMs that support setting a seed, in-flight batching mean responses are still not deterministic.
- Regenerating the modules is time consuming. The results in the paper require over 200K queries (168 problems, 20 samples/problem, 5 models, 12 configurations) using a total of almost 200M tokens, so depending on API rate throttling it could take multiple days to regenerate all of the modules.
- Regenerating the modules is expensive. Using the closed models would require an evaluator to pay for API access, and using the open models would either require dedicated GPUs or paying for API access to hosted models. With 200K queries and almost 200M tokens, the cost to regenerate all of modules could be significant.

Given these issues, the Docker image includes the 200K+ LLM-generated Verilog, PyMTL3, MyHDL, Migen, and Amaranth RTL modules used in the paper. The artifact evaluation involves simulating these 200K+ RTL modules and analyzing the results to reproduce the tables in the paper.

11.2 Artifact Check-List

- **Program:** The Docker image includes the PyHDL-Eval framework along with all other programs required to reproduce the results in the paper.
- **Compilation:** The Docker image includes the required C++ compiler and Python interpreter.
- **Binary:** The Docker image includes pre-installed binaries for all tools.
- **Model:** As described above, the artifact evaluation does not include querying the LLMs studied in this paper due to determinism, time, and cost. However the specific model versions are listed in the paper, and these LLMs are widely accessible.
- **Data Set:** The Docker image includes all necessary datasets (i.e., Verilog reference solutions, Verilog test benches, Python test scripts, LLM-generated RTL modules).
- **Run-Time Environment:** The Docker image uses an Ubuntu 24.02 base image and pre-installed binaries for all tools.
- **Hardware:** The Docker image can run on a variety of hardware, but we recommend using an x86_64 machine with 16+ cores.
- **Output:** The README explains how to generate log files for all 200K+ simulations, summary reports, and ultimately the tables and figures used in the paper.
- **Experiments:** The Docker image includes scripts to automate the process of running all simulations and analyzing the results.
- **Required Disk Space:** The Docker image takes about 350 MB of disk space and 850 MB when loaded. The Docker container will require approximately 10 GB of space when running.
- **Workflow Preparation Time:** The Docker image provides scripts to automate the workflow, so the artifact evaluation should require very little workflow preparation time beyond installing Docker and downloading the Docker image.
- **Experiment Time:** Reproducing the results in the paper requires running over 200K+ simulations, although each simulation is quite short. Running these simulations should take approximately 3–4 hours if using a machine with 16+ cores.
- **Publicly Available:** A README and Docker image are publicly available at Zenodo: https://zenodo.org/records/13117553.
- **Code and Data Licenses:** MIT License
- **Workflow Framework:** The Docker image includes Make-based workflow orchestration scripts.
- **Archive DOI:** 10.5281/zenodo.13117553

11.3 Description

- **How to Access:** A README and Docker image are publicly available at Zenodo: https://zenodo.org/records/13117553.
- **Hardware Dependencies:** The Docker image can run on a variety of hardware, but we recommend using an x86_64 machine with 16+ cores.
- **Software Dependencies:** The Docker image contains all software dependencies required for the artifact evaluation including pre-installed binaries for all tools; pre-installed Python packages for all five Python-embedded DSLs; and source code for the PyHDL-Eval framework.

- **Data Sets:** The Docker image includes all necessary datasets (i.e., Verilog reference solutions, Verilog test benches, Python test scripts, LLM-generated RTL modules).
- **Models:** As described above, the artifact evaluation does not include querying the LLMs studied in this paper due to determinism, time, and cost. However the specific model versions are listed in the paper, and these LLMs are widely accessible.

11.4 Installation

Download and install Docker on the evaluation machine. The README provides detailed step-by-step instructions for how to download, load, and run the Docker image for reproducing the results in this paper. The Docker image includes all software dependencies required for the artifact evaluation including pre-installed binaries for all tools; pre-installed Python packages for all five Python-embedded DSLs; and source code for the PyHDL-Eval framework. The Docker image also includes RTL modules pre-generated using all five LLMs and scripts for running simulations and generating all of the tables in the paper.

11.5 Experiment Workflow

The README provides detailed step-by-step instructions for using the included workflow to regenerate the results in the paper.

11.6 Evaluation and Expected Results

The README and Docker image will enable reproducing:

- Table 1: PyHDL-Eval Problem Categories
- Table 2: Average Pass Rate
- Table 3: Average Pass Rate with Three ICL Examples Organized by Problem Category
- Appendix A: PyHDL-Eval Problem List
- Appendix D: Results for ICL Organized by Problem Category
- Appendix E: Results per Problem for GPT4 Turbo Targeting Verilog vs. MyHDL

Note that the data in Table 2 is all that is needed to regenerate the plots in Figure 6 and Figure 7. "Golden" reference reports are included in the Docker image to simplify comparing newly generated results to the results used for the paper.