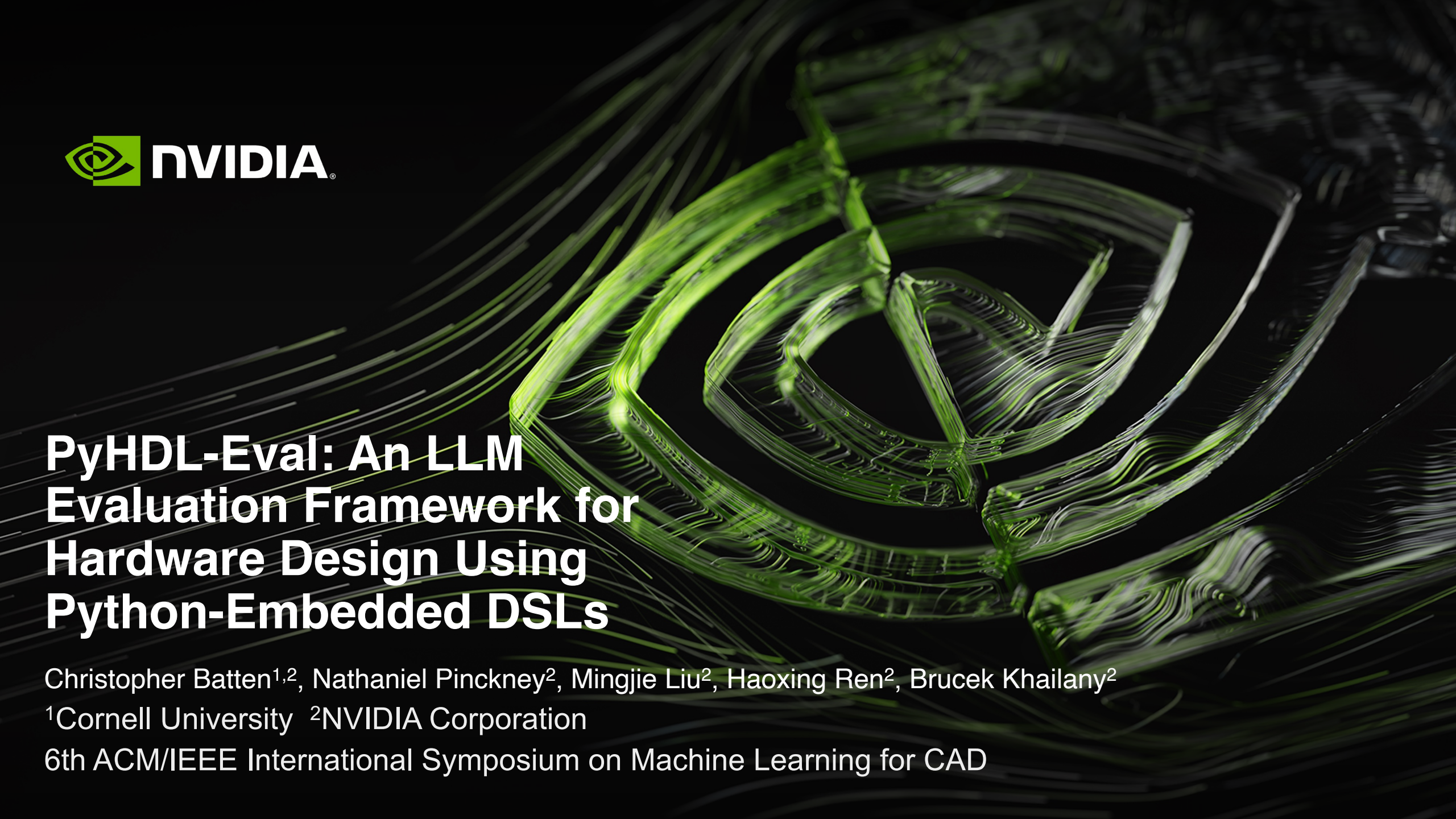
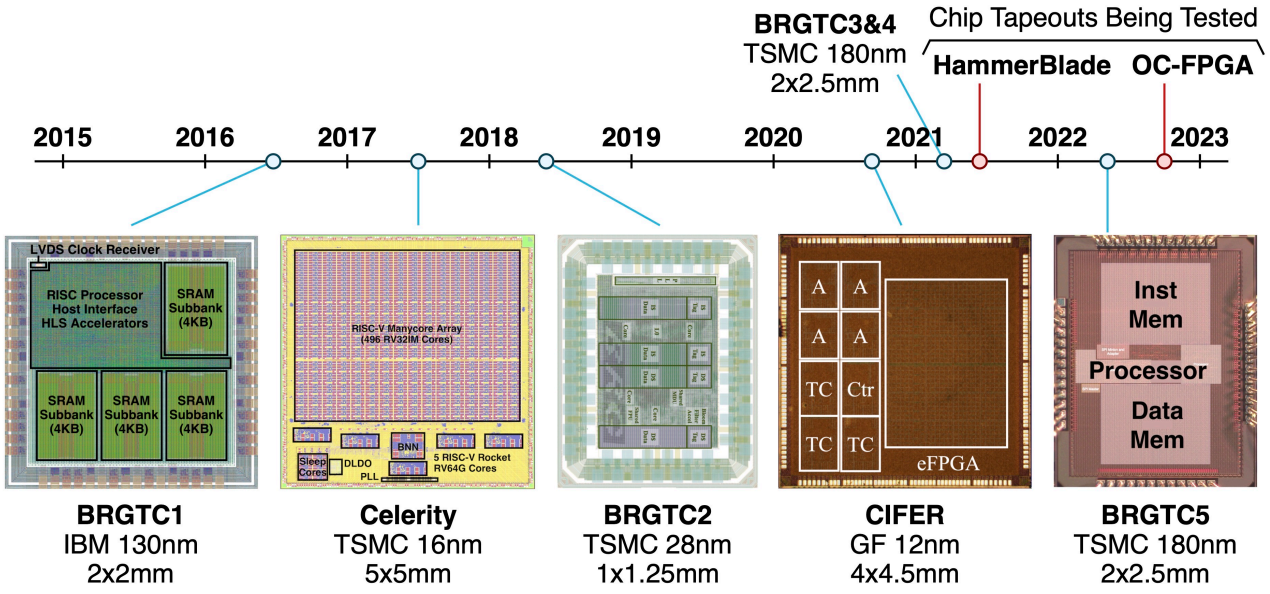


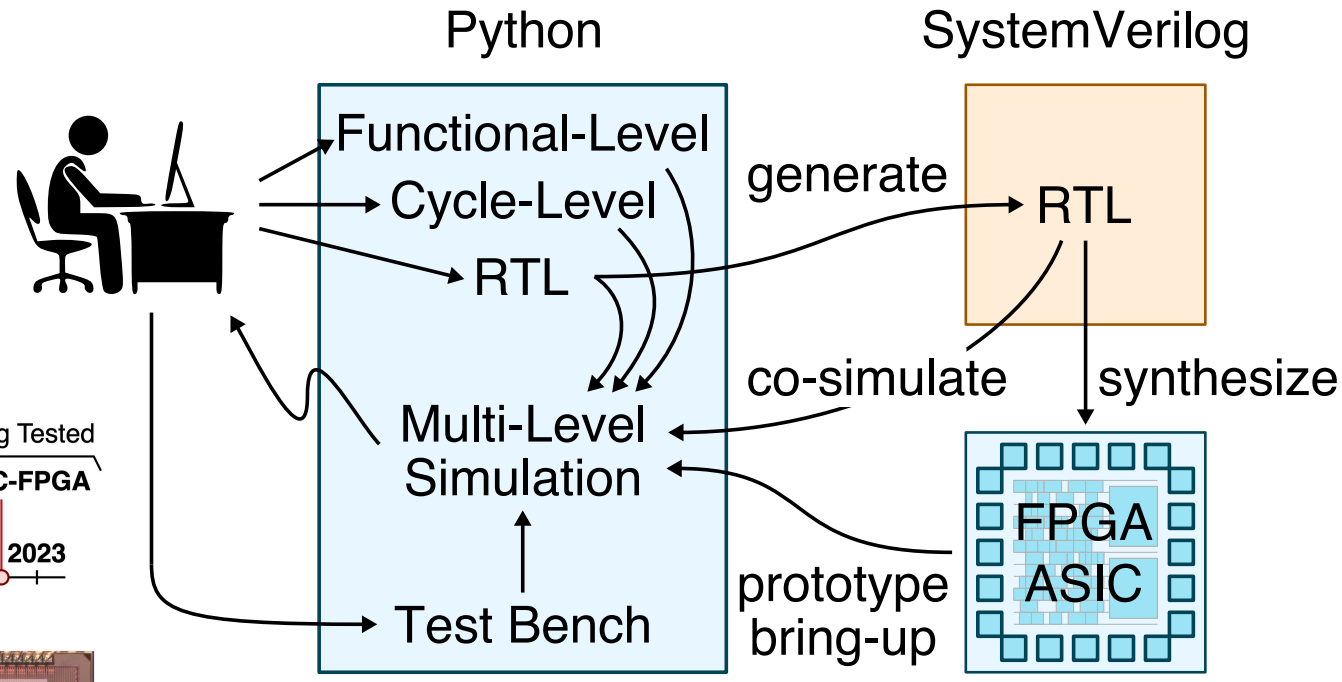
PyHDL-Eval: An LLM Evaluation Framework for Hardware Design Using Python-Embedded DSLs

Christopher Batten[1,2], Nathaniel Pinckney[2], Mingjie Liu[2], Haoxing Ren[2], Brucek Khailany[2]
[1]Cornell University [2]NVIDIA Corporation
6th ACM/IEEE International Symposium on Machine Learning for CAD

PyMTL: A Python Framework for Hardware Modeling, Generation, Simulation, & Verification

- PyMTL3 enables:
  - implementing multi-level hardware models
  - analyzing, instrumenting, transforming models
  - simulating models w/ fast JIT-compiled simulator
  - applying state-of-the-art SW testing to HW



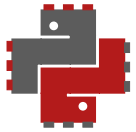



```python
from pymtl3 import *

class RegIncrRTL( Component ):

  def construct( s, nbits ):
    s.in_ = InPort ( nbits )
    s.out = OutPort( nbits )
    s.tmp = Wire   ( nbits )

  @update_ff
  def seq_logic():
    s.tmp <<= s.in_

  @update
  def comb_logic():
    s.out @= s.tmp + 1
```

Python-Embedded DSLs

Modules as function vs classes
Different bit datatypes

Cycle-level vs event-based
Implicit vs explicit clk/reset



PyMTL

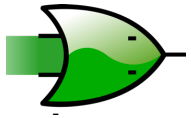
```python
from pymtl3 import *

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
```

PyRTL

```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
```

MyHDL

```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
```



Migen

```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 )
```

Amaranth

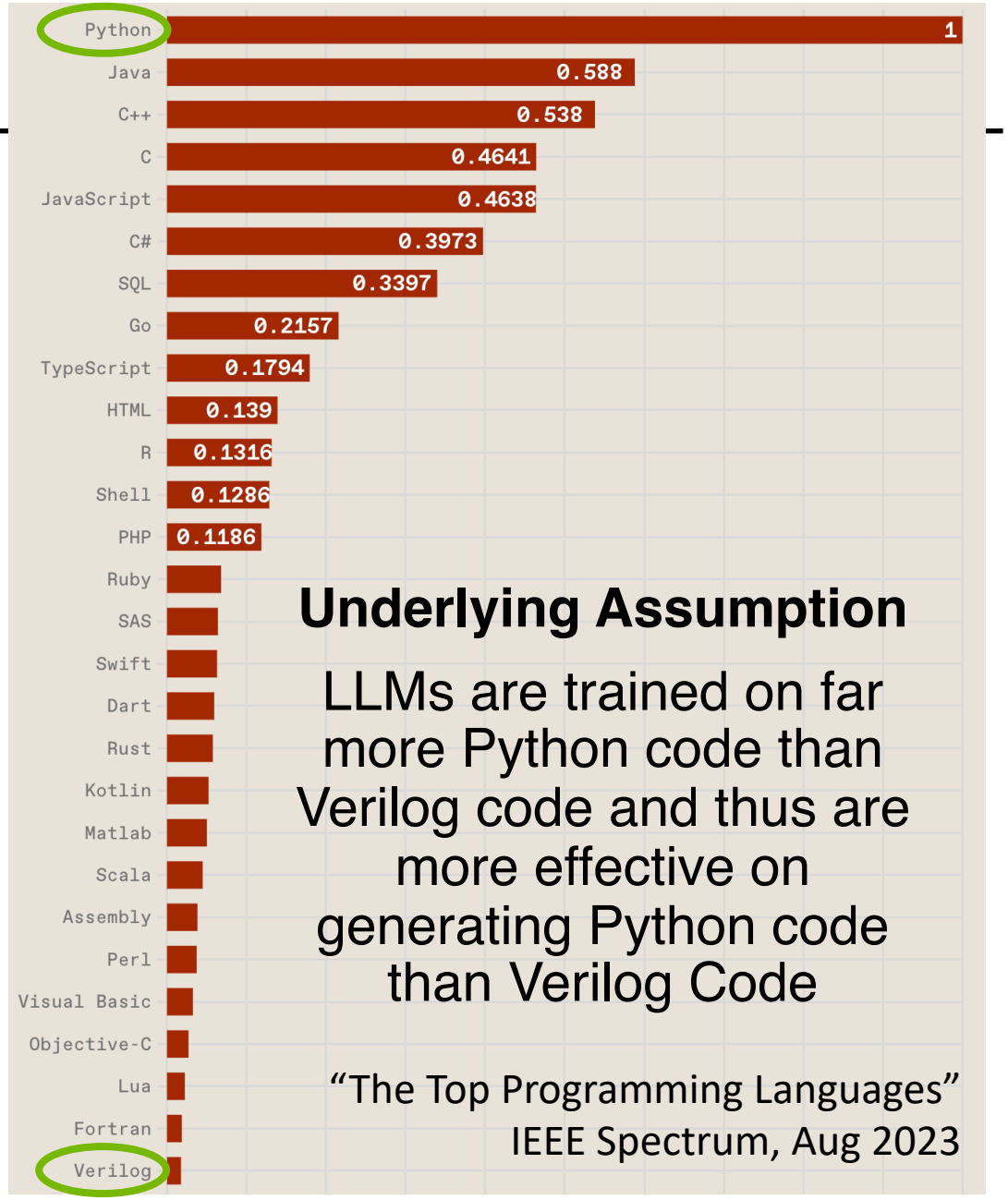
```python
from amaranth import *

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
```

Embedded DSLs & LLMs

A hardware design framework embedded in the most popular general-purpose programming language can potentially make ~~**design engineers**~~ more effective

Hypothesis

A hardware design framework embedded in the most popular general-purpose programming language can potentially make **LLMs** more effective

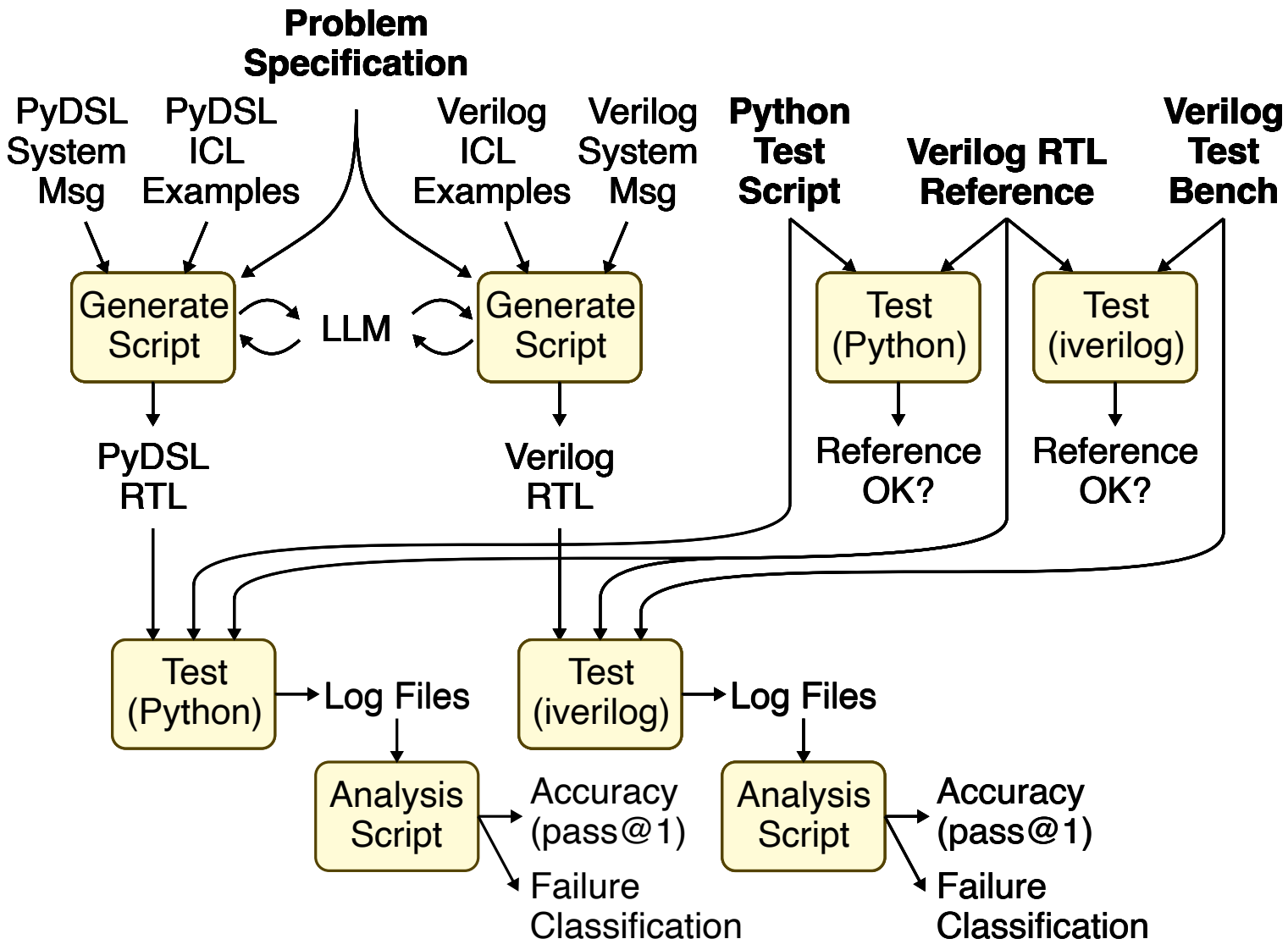


Underlying Assumption

LLMs are trained on far more Python code than Verilog code and thus are more effective on generating Python code than Verilog Code

"The Top Programming Languages"
IEEE Spectrum, Aug 2023

PyHDL-Eval Framework



- Each problem includes prompt, Verilog reference, Verilog test bench, and Python test script
- **Generate** script supports different ICL examples, models, model hosting services, and languages
- **Verilog reference** can be tested against itself using both Icarus Verilog and Python via PyMTL3/Verilator
- **LLM-generated Verilog RTL** is tested against Verilog reference using Icarus Verilog
- **Analysis scripts** enable automatic failure classification
- **LLM-generated Python-embedded DSL RTL** is tested against Verilog reference using Python test scripts
- Includes **workflow management scripts**

PyHDL-Eval Problems

	Problem Category		Examples of Problems in Category	Num Probs
1	comb	const	constant zero, one, lohi, 32-bit value	4
2	comb	wires	pass through, split, bit/byte reverse, sign extend	8
3	comb	gates	single logic gate, multiple logic gates, natural language	13
4	comb	bool	logic equations, truth table, K-map, waveform, natural language	15
5	comb	mux	various bitwidths and number of inputs, mux/demux	9
6	comb	codes	encoder, decoder, priority encoder, graycode, BCD, ASCII, parity	14
7	comb	arith	add/sub, popcount, multiplication, shift/rotate, comparison, ALU, sorter	17
8	comb	fsm	FSM tables and diagrams for next state and output logic	9
9	comb	param	parameterized: gates, mux, encoder, decoder, priority encoder, rotator	8
10	seq	gates	single gate, multiple gates, natural language	9
11	seq	bool	truth table, waveform	5
12	seq	sreg	shift registers: SISO, PISO, SIPO, universal, bidir, LFSR	7
13	seq	count	counters: binary, up/down, saturating, variable, decade, timer, clock	9
14	seq	edge	edge detect, edge capture, edge counters, max switching	5
15	seq	arb	arbiters: priority, rotational oblivious, round robin, grant-hold, weighted	5
16	seq	fsm	FSM tables and diagrams, natural language	13
17	seq	mem	1r1w, 2r1w register file (forwarding, reg zero, byte enables), 1s1w CAM	6
18	seq	arith	bit serial incrementer, bit serial adder, byte-serial adder, accumulator	4
19	seq	pipe	1/2/3-stage delay, 1/2-stage 2/3/4-input adder, 2-stage minmax	8

- New evaluation benchmark with **168 specification-to-RTL problems developed from scratch**
- Careful **ontological approach** with problems categorized into classes and subclasses
  - 97 combinational problems
  - 71 sequential problems
  - 19 subclasses
- Covers ~75% of the concepts from the VerilogEval benchmark (~117/156 problems)
  - *Excludes* problems not supported by all Python-embedded DSLs (latch-based design, neg edge, asynchronous reset)
  - *Excludes* complex lemmings and cellular automata game problems
  - *Excludes* bug fixing and hierarchical design
  - *Excludes* under-specified, ambiguous problems
  - *Includes* parameterized design problems
  - *Includes* new kind of problems

PyHDL-Eval Problem 9.8

	Problem Category		Examples of Problems in Category	Num Probs
1	comb	const	constant zero, one, lohi, 32-bit value	4
2	comb	wires	pass through, split, bit/byte reverse, sign extend	8
3	comb	gates	single logic gate, multiple logic gates, natural language	13
4	comb	bool	logic equations, truth table, K-map, waveform, natural language	15
5	comb	mux	various bitwidths and number of inputs, mux/demux	9
6	comb	codes	encoder, decoder, priority encoder, graycode, BCD, ASCII, parity	14
7	comb	arith	add/sub, popcount, multiplication, shift/rotate, comparison, ALU, sorter	17
8	comb	fsm	FSM tables and diagrams for next state and output logic	9
9	comb	param	parameterized: gates, mux, encoder, decoder, priority encoder, rotator	8
10	seq	gates	single gate, multiple gates, natural language	9
11	seq	bool	truth table, waveform	5
12	seq	sreg	shift registers: SISO, PISO, SIPO, universal, bidir, LFSR	7
13	seq	count	counters: binary, up/down, saturating, variable, decade, timer, clock	9
14	seq	edge	edge detect, edge capture, edge counters, max switching	5
15	seq	arb	arbiters: priority, rotational oblivious, round robin, grant-hold, weighted	5
16	seq	fsm	FSM tables and diagrams, natural language	13
17	seq	mem	1r1w, 2r1w register file (forwarding, reg zero, byte enables), 1s1w CAM	6
18	seq	arith	bit serial incrementer, bit serial adder, byte-serial adder, accumulator	4
19	seq	pipe	1/2/3-stage delay, 1/2-stage 2/3/4-input adder, 2-stage minmax	8

```
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
```

```
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
```

PyHDL-Eval Problem 19.6

	Problem Category		Examples of Problems in Category	Num Probs
1	comb	const	constant zero, one, lohi, 32-bit value	4
2	comb	wires	pass through, split, bit/byte reverse, sign extend	8
3	comb	gates	single logic gate, multiple logic gates, natural language	13
4	comb	bool	logic equations, truth table, K-map, waveform, natural language	15
5	comb	mux	various bitwidths and number of inputs, mux/demux	9
6	comb	codes	encoder, decoder, priority encoder, graycode, BCD, ASCII, parity	14
7	comb	arith	add/sub, popcount, multiplication, shift/rotate, comparison, ALU, sorter	17
8	comb	fsm	FSM tables and diagrams for next state and output logic	9
9	comb	param	parameterized: gates, mux, encoder, decoder, priority encoder, rotator	8
10	seq	gates	single gate, multiple gates, natural language	9
11	seq	bool	truth table, waveform	5
12	seq	sreg	shift registers: SISO, PISO, SIPO, universal, bidir, LFSR	7
13	seq	count	counters: binary, up/down, saturating, variable, decade, timer, clock	9
14	seq	edge	edge detect, edge capture, edge counters, max switching	5
15	seq	arb	arbiters: priority, rotational oblivious, round robin, grant-hold, weighted	5
16	seq	fsm	FSM tables and diagrams, natural language	13
17	seq	mem	1r1w, 2r1w register file (forwarding, reg zero, byte enables), 1s1w CAM	6
18	seq	arith	bit serial incrementer, bit serial adder, byte-serial adder, accumulator	4
19	seq	pipe	1/2/3-stage delay, 1/2-stage 2/3/4-input adder, 2-stage minmax	8

```
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.
```

Verilog TBs

- Includes simple Verilog testing framework
  - Instantiate reference and DUT modules
  - Define comparison task
  - Define test case tasks
  - Define main initial block
- Features to make it easier to debug tests
  - Easily run test cases in isolation
  - Common trace output showing DUT inputs/outputs every cycle
  - Common error reporting

```
//========================================================================
// Prob02p01_comb_wires_8b_passthru_test
//========================================================================

`include "test_utils.v"

module Top();

  //----------------------------------------------------------------------
  // Setup
  //----------------------------------------------------------------------

  logic clk;
  logic rst;

  TestUtils t( .* );

  //----------------------------------------------------------------------
  // Instantiate reference and top modules
  //----------------------------------------------------------------------

  logic [7:0] ref_module_in_;
  logic [7:0] ref_module_out;

  RefModule ref_module
  (
    .in_ (ref_module_in_),
    .out (ref_module_out)
  );

  logic [7:0] top_module_in_;
  logic [7:0] top_module_out;

  TopModule top_module
  (
    .in_ (top_module_in_),
    .out (top_module_out)
  );

  //----------------------------------------------------------------------
  // compare
  //----------------------------------------------------------------------
  // All tasks start at #1 after the rising edge of the clock. So we
  // write the inputs #1 after the rising edge, and check the outputs #1
  // before the next rising edge.

  task compare
  (
    input logic [7:0] in_
  );

    ref_module_in_ = in_;
    top_module_in_ = in_;

    #8;

    if ( t.n != 0 )
      $display( "%3d: %x %x", t.cycles, top_module_in_, top_module_out );

    `TEST_UTILS_CHECK_EQ( top_module_out, ref_module_out );

    #2;

  endtask
```

```
  //----------------------------------------------------------------------
  // test_case_1_directed
  //----------------------------------------------------------------------

  task test_case_1_directed();
    $display( "\ntest_case_1_directed" );

    t.reset();
    compare( 8'b0000_0000 );
    compare( 8'b0000_0001 );
    compare( 8'b0000_0010 );
    compare( 8'b0000_0100 );
    compare( 8'b0000_0100 );
    compare( 8'b0001_0001 );
    compare( 8'b0010_0010 );
    compare( 8'b0100_0100 );
    compare( 8'b1000_1000 );

  endtask

  //----------------------------------------------------------------------
  // test_case_2_random
  //----------------------------------------------------------------------
  // svt.seed is set to a known value in the reset() task, so when use
  // $urandom(t.seed) we will get reproducible random numbers no matter
  // the order that test cases are executed.

  logic [7:0] test_case_2_random_data;
  task test_case_2_random();
    $display( "\ntest_case_2_random" );

    t.reset();
    for ( int i = 0; i < 20; i = i+1 )
      compare( $urandom(t.seed) );

  endtask

  //----------------------------------------------------------------------
  // main
  //----------------------------------------------------------------------
  // We start with a #1 delay so that all tasks will essentially start at
  // #1 after the rising edge of the clock.

  initial begin
    #1;

    if ((t.n <= 0) || (t.n == 1)) test_case_1_directed();
    if ((t.n <= 0) || (t.n == 2)) test_case_2_random();

    $write("\n");
    $finish;
  end

endmodule
```

Python Test Scripts

- Includes API to enable a single Python test script to be applied across Verilog and five Python-embedded DSLs
  - constructor (create module from source file)
  - reset module
  - write port
  - read port
  - tick_phase0, tick_phase1
  - cleanup
- Each Python-embedded DSL requires ~100 lines of Python code

```python
#=========================================================================
# Prob02p01_comb_wires_8b_passthru_test
#=========================================================================

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

#-------------------------------------------------------------------------
# Configuration
#-------------------------------------------------------------------------

config = Config(
  ports = [
    ( "in_", InputPort (8) ),
    ( "out", OutputPort(8) ),
  ],
)

#-------------------------------------------------------------------------
# test_case_directed
#-------------------------------------------------------------------------

def test_case_directed( pytestconfig ):
  run_sim( pytestconfig, __file__, config,
  [
    0b0000_0000,
    0b0000_0001,
    0b0000_0010,
    0b0000_0100,
    0b0000_0100,
    0b0001_0001,
    0b0010_0010,
    0b0100_0100,
    0b1000_1000,
  ])

#-------------------------------------------------------------------------
# test_case_random
#-------------------------------------------------------------------------

@settings(deadline=1000,max_examples=20)
@given( st.lists(pst.bits(8)) )
def test_case_random( pytestconfig, test_vectors ):
  run_sim( pytestconfig, __file__, config, test_vectors )
```

Support for specifying parameters and startup cycles

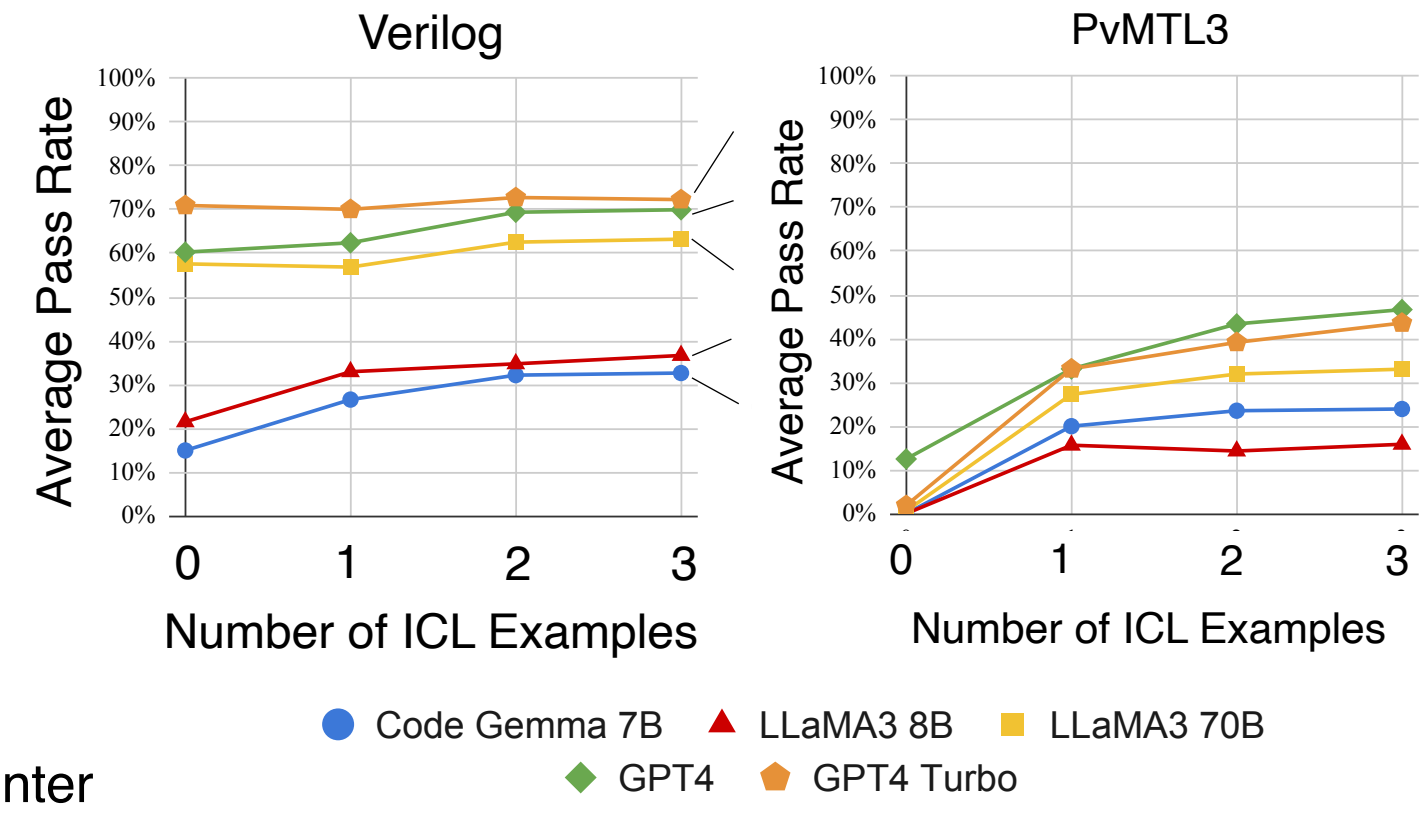
run_sim compares Python-embedded DSL to Verilog reference via PyMTL3/Verilator

Pytest library for testing & Hypothesis library used for advanced random testing

```python
#=========================================================================
# pyhdl_eval.myhdl
#=========================================================================
# PyHDL-Eval support for MyHDL.

import importlib
import types
import myhdl

from pyhdl_eval.bits import Bits

class TopModuleMyHDL():

  def __init__( s, file_name, config ):

    loader = importlib.machinery.SourceFileLoader("TopModule",file_name)
    mod = types.ModuleType(loader.name)
    loader.exec_module(mod)

    # ... ~30 lines of code to dynamically add ports ...

    # Create a top level wrapper for hardware module

    @myhdl.block
    def test_top():
      top = mod.TopModule( *signal_list, **config.parameters )

      @myhdl.always(myhdl.delay(5))
      def clkgen():
        s.signals.clk.next = not s.signals.clk

      return top, clkgen

    s.sim = test_top()

  def reset( s ):
    if s.has_reset:
      s.signals.reset.next = 1
      s.sim.run_sim( 30, quiet=True )
      s.signals.reset.next = 0

  def write( s, port_name, value ):
    if isinstance( value, Bits ):
      value = value.uint()
    port = getattr( s.signals, port_name )
    port.next = value

  def read( s, port_name ):
    myhdl_modbv = getattr( s.signals, port_name )
    nbits = len(myhdl_modbv)
    return Bits(nbits,int(myhdl_modbv))

  def tick_phase0( s ):
    s.sim.run_sim( 8, quiet=True )

  def tick_phase1( s ):
    s.sim.run_sim( 2, quiet=True )

  def cleanup( s ):
    s.sim.quit_sim()
```

PyHDL-Eval Results: ICL for Verilog & PyMTL3

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%



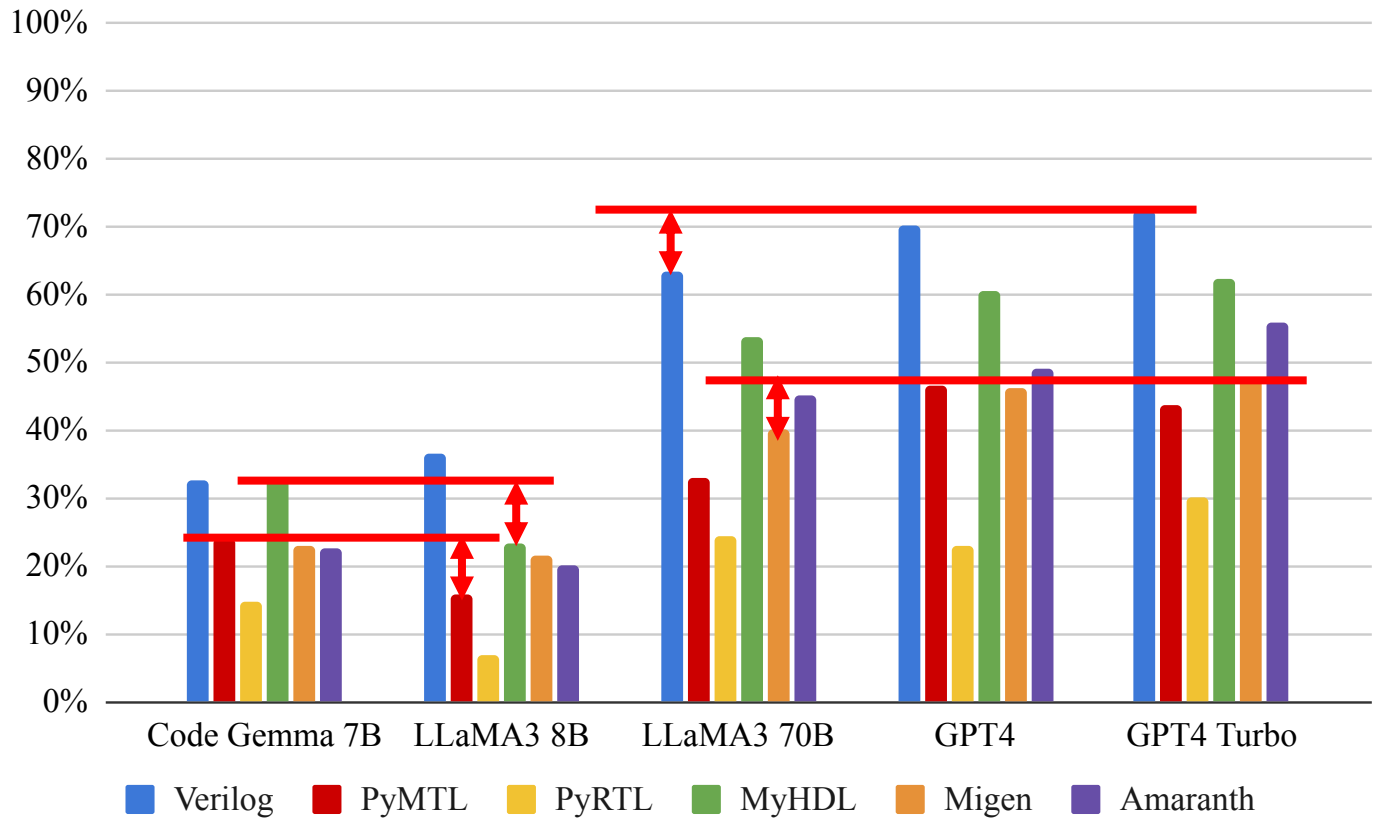
- Up to three ICL examples
  - 8-bit combinational incrementer
  - 8-bit registered incrementer
  - 8-bit parameterized registered incrementer
- ICL for Verilog
  - ICL examples help smaller models more than larger models
  - ICL examples also help LLaMA3 70B and GPT4
  - ICL examples do not significantly help GPT4 Turbo

ICL is critical for Python-embedded DSLs (even for GPT4 Turbo)!

PyHDL-Eval Results: Comparing Embedded DSLs

DSL	Num ICL Ex	Code Gemma 7B	Llama3 8B	Llama3 70B	GPT4	GPT4 Turbo
Verilog	3	32.7%	36.6%	63.2%	69.9%	72.2%
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%

- State-of-the-art GPT4 Turbo does reasonably well on Verilog specification-to-RTL tasks
- Larger open models (LLaMA3 70B) nearing larger closed models (GPT4, GPT4 Turbo)
- Smaller code foundation model (Code Gemma 7B) outperforms smaller general foundation model (LLaMA3 8B) on Python-embedded DSL
- LLMs have more success on specification-to-RTL tasks targeting MyHDL vs other Python-embedded DSLs



Ultimately, our findings do not support the original hypothesis that LLMs would be naturally more effective targeting Python-embedded DSLs

PyHDL-Eval Results: Pass Rate vs. Subclasses

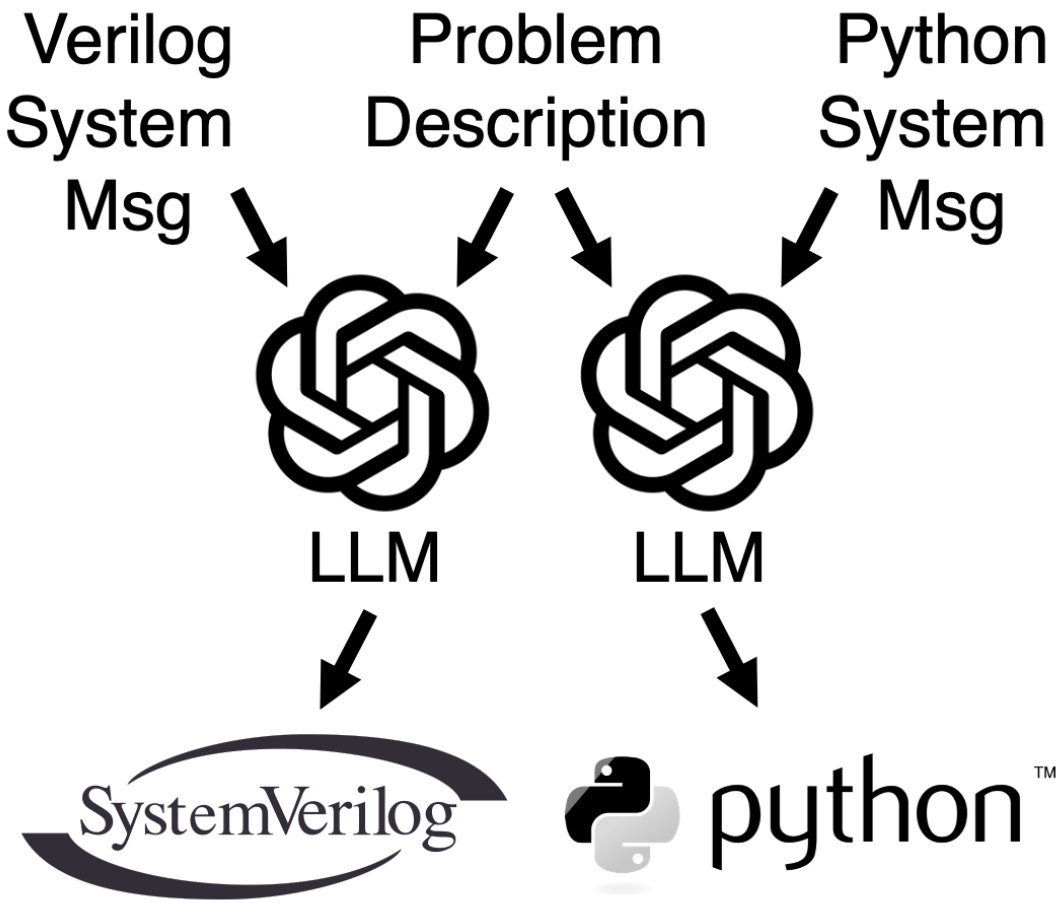
	Problem Category		Num Probs	Code Gemma 7B						LLaMA3 8B						LLaMA3 70B						GPT4						GPT4 Turbo					
				Ver	PyM	PyR	MyH	Mig	Ama	Ver	PyM	PyR	MyH	Mig	Ama	Ver	PyM	PyR	MyH	Mig	Ama	Ver	PyM	PyR	MyH	Mig	Ama	Ver	PyM	PyR	MyH	Mig	Ama
1	comb	const	4	86	100	69	0	100	98	85	95	25	0	100	100	100	100	76	0	100	100	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	5	45	54	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	29	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	57	17	12	11	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	16	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

Uses empty sensitivity list

Insists on using explicit clock

Key Contributions

- This is the first LLM evaluation framework for hardware design targeting multiple Python-embedded DSLs
- In-context learning is critical for improving the LLM pass rate for Python-embedded DSLs
- Rigorous analysis of LLMs targeting both Verilog and five Python-embedded DSLs suggest LLMs achieve higher pass rates when targeting Verilog



https://github.com/cornell-brg/pyhdl-eval