

Helping *Pythonistas* Become Microarchitects

Using Jupyter Notebooks and CIRCT/MLIR/LLVM

Zhiyang Ong

Department of Electrical and Computer Engineering
College of Engineering,
Texas A&M University
College Station, TX

September 30, 2023



- ❶ Problems in Computer Architecture
- ❷ New Golden Era of Computer Architecture, EDA, and Compiler Design
- ❸ Python-based IC Design

Table of Contents

- 1 Problems in Computer Architecture
- 2 New Golden Era of Computer Architecture, EDA, and Compiler Design
- 3 Python-based IC Design

Problems in Computer Architecture

Specifically with General-Purpose Processor Architectures

Golden Era of Computer Architecture (1980s till early 2000s):

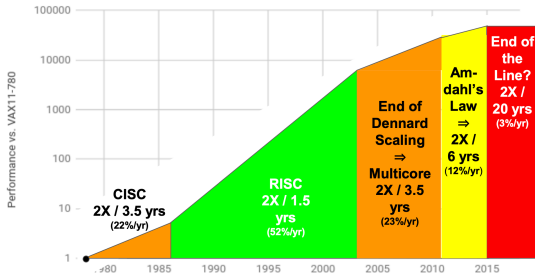
- **Memory Wall** [Wulf1995] [Hennessy1990] [Horowitz2023] [Solihin2002]
- **End of Dennard's scaling** [Dennard1974] [Haensch2006] [Chen2006] [Dennard2007] [Calhoun2008] [Iwai2009] and **Power Wall** [Keshavarzi2007]
- **Dark Silicon** [Esmaeilzadeh2011] [Esmaeilzadeh2012] [Rahmani2017] [Hurson2018]
- **ILP Wall** → **limitations of** [Hennessy2019, §1.11, pp. 39]
- **impending doom of Moore's law** [Duranton2019] [Kelleher2022]
- **decline of general-purpose processors** [Thompson2018]
- **Hardware Accelerator Wall** [Fuchs2019]

Problems in Computer Architecture

Specifically with General-Purpose Processor Architectures

End of Growth of Single Program Speed?

40 years of Processor Performance



Based on SPECintCPU. Source: John Hennessy and David Patterson, Computer Architecture: A Quantitative Approach, 6/e. 2018

Figure: Plot of the performance of general-purpose processors over time, from 1980 till the late 2010s [Hennessy2018]



Table of Contents

- 1 Problems in Computer Architecture
- 2 New Golden Era of Computer Architecture, EDA, and Compiler Design
- 3 Python-based IC Design

New Golden Era of Computer Architecture (1)

And, also for EDA and Compiler Design

Problems → Opportunities [Hennessy2019a]

Domain-Specific Computing [Hennessy2019] → **Heterogeneous System Architectures** [HSAFoundationAdministration2016] [Hwu2016] [Duranton2019]

Hardware Security [Gruss2017] [Szefer2018] [Duranton2021]

Open-Source ISA [Patterson2018b], and support ecosystem across the hardware/software stack

New Golden Era of Computer Architecture (2)

And, also for EDA and Compiler Design

Agile IC Design Methodologies [Gerstlauer2001] [Hennessy2018]
[Johnson2018a] + **Python-based IC Design**

Domain-Specific Compilers [Lattner2021a] + **Compilers for
Heterogeneous Systems**

System-Technology Co-Optimization [Wu2021]:

- **system** → computer systems → hardware/software co-design
- **semiconductor manufacturing technology** (including semiconductor device engineering)



New Golden Era of Computer Architecture (3)

And, also for EDA and Compiler Design

Recent **Non- von Neumann Computing Paradigms and Technology Trends:**

- **in-memory computing** [Zhu2013] [Paul2014] [Williams2017a] [Theis2017] [Imani2020] [Wu2021]
- **hyperdimensional computing** [Imani2020] [Wu2021]
- **photonic ICs** [Topaloglu2015]
- **wafer-scale computing**
- **3-D ICs and high-bandwidth memory interconnects**
- **chiplet-based System-in-Package design**

Table of Contents

- 1 Problems in Computer Architecture
- 2 New Golden Era of Computer Architecture, EDA, and Compiler Design
- 3 Python-based IC Design

Python-based IC Design: Options

Possible options:

- **MyHDL** (old)
- **PyMTL** (Cornell University)
- **PyRTL** (University of California, Santa Barbara)
- **Jupyter Notebook + Python** -based IC design flow (supported by Google Colab)
- **CIRCT: Circuit IR Compilers and Tools** [Lattner2021]
 - LLVM (initially, Low Level Virtual Machine) [Lopes2014] [Pandey2015] [Sarda2015]
 - Multi-Level Intermediate Representation, MLIR (extension of LLVM ecosystem for domain-specific computing)

Python-based IC Design: Options

Possible options:

- **MyHDL** (old)
- **PyMTL** (Cornell University)
- **PyRTL** (University of California, Santa Barbara)
- **Jupyter Notebook + Python** -based IC design flow (supported by Google Colab)
- **CIRCT: Circuit IR Compilers and Tools** [Lattner2021]
 - LLVM (initially, Low Level Virtual Machine) [Lopes2014] [Pandey2015] [Sarda2015]
 - Multi-Level Intermediate Representation, MLIR (extension of LLVM ecosystem for domain-specific computing)

Jupyter Notebook + Python -based IC Design Flow

Supported by Google Colab

= DSLX

1. Update the `mul4` function below to use the [DSLX standard library](#) functions to implement a 4-bit multiplier (don't forget the `std::` prefix).
2. Generate the verilog for the design.
3. Run the OpenLane flow up until synthesis.
4. Observe the change in the complexity of the graph.
5. Compare to the results w/ the previous adder design.

```
[ ] %%bash -c 'cat > user_module.x; interpreter_main user_module.x'
import std

fn mul4(a: u4, b: u4) -> u8 {
  u8:0 // TODO(YOU) implement mul4
}

fn user_module(io_in: u8) -> u8 {
  mul4(io_in[0:4], io_in[4:8]) as u8
}

#[test]
fn test() {
  let _ = assert_eq(mul4(u4:8, u4:8), u8:64);
  let _ = assert_eq(user_module(u8:0b1000_1000), u8:0b0100_0000);
  -
}
```

Figure: DSLX -based Design: Python-like hardware construction language (HCL), or HDL

Jupyter Notebook + Python -based IC Design Flow

Supported by Google Colab

The screenshot shows a Jupyter Notebook titled 'xls-workshop-openlane.ipynb'. The left sidebar contains a 'Table of contents' with the following items: DSLX, Convert to Hardware IR, Generate RTL, Run the OpenLane flow, Configuration, Synthesis, Preview, Floorplan, Preview, Placement, Preview, Routing, Preview, Sign off, and Preview. The main area displays a code cell with the following Python code:

```
[ ]: %%bash -c "cat > test.py; make"
import cocotb
from cocotb.triggers import ClockCycles, RisingEdge
from cocotb.clock import Clock

@cocotb.test()
async def test_test(dut):
    c = Clock(dut.clk, 1, 'ns')
    cocotb.start_soon(c.start())

    # TODO(YOU): update with the 8-input to user_module
    dut.user_module_tb.io_in.value = NaN
    # pipeline stage 0: fetch
    await ClockCycles(dut.clk, 2)
    # TODO(YOU): verify that the input value propagate to stage #0
    assert dut.user_module_tb.p0_io_in == NaN

    # pipeline stage 1: slice
    await ClockCycles(dut.clk, 2)
    # TODO(YOU): update "pl_bit_slice_*" names to match the wire in the gene
    # TODO(YOU): verify that the input value is sliced into 4-bit umul opera
    assert dut.user_module_tb.pl_bit_slice_TODO == NaN
    assert dut.user_module_tb.pl_bit_slice_TODO == NaN

    # pipeline stage 2: multiply
    await ClockCycles(dut.clk, 2)
    # TODO(YOU): verify that the output value match the multiplication result
    assert dut.user_module_tb.out == NaN
```

Figure: Python-based VLSI Functional Verification: Using cocotb for HDL-based Simulation. . . Can use *PyUVM*

New Golden Era of Computer Architecture, EDA, and Compiler Design

Slide Subtitle 3

Problems → Opportunities

Domain-Specific Computing
→ Heterogeneous System
Architectures.

Statement 3.

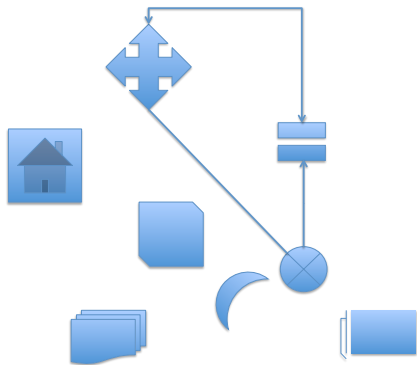


Figure: My caption [?, ?]



References