Universidade Federal de São Carlos
Departamento de Computação

Trabalho de Conclusão de Curso
Engenharia de Computação

Implementação de um ambiente de verificação funcional UVM em Python aplicado ao fluxo de design digital SKY130

Estudante: **Marcelo Rodrigues Soares**
Orientador: **Prof. Dr. Ricardo Menotti**

São Carlos, 13 de julho de 2026

Soares, Marcelo Rodrigues

Implementação de um ambiente de verificação funcional UVM em Python aplicado ao fluxo de design digital SKY130 / Marcelo Rodrigues Soares -- 2026.
58f.

TCC (Graduação) - Universidade Federal de São Carlos, campus São Carlos, São Carlos
Orientador (a): Ricardo Menotti
Banca Examinadora: Edilson Reis Rodrigues Kato, Rafael Vidal Aroca
Bibliografia

1. Verificação funcional. 2. Microeletrônica. 3. Descrição de hardware. I. Soares, Marcelo Rodrigues. II. Título.

Ficha catalográfica desenvolvida pela Secretaria Geral de Informática (SIn)

DADOS FORNECIDOS PELO AUTOR

Bibliotecário responsável: Arildo Martins - CRB/8 7180

Resumo

A democratização do desenvolvimento de circuitos integrados promovida pelas ferramentas abertas ainda encontra na verificação funcional um de seus principais obstáculos. Este trabalho investiga a viabilidade de um fluxo de verificação baseado em Python aplicado a etapas avançadas do projeto digital, abrangendo cobertura funcional, cobertura estrutural do código, simulação após implementação física e estimativa de consumo de potência. Para essa avaliação, foi desenvolvido um ambiente completo de verificação para um controlador de comunicação serial amplamente utilizado em sistemas embarcados, utilizando uma tecnologia aberta de fabricação de circuitos integrados como plataforma de referência. Os resultados demonstram que a abordagem é capaz de validar o circuito em diferentes níveis de abstração, alcançando elevada cobertura e produzindo estimativas consistentes de consumo energético. Os experimentos indicam que a integração entre o ecossistema Python e ferramentas abertas representa uma alternativa viável, acessível e suficientemente robusta para atividades de pesquisa, ensino e desenvolvimento de circuitos integrados.

Palavras-chave: Verificação funcional. UVM. cocotb. pyuvm. I2C. SKY130. Simulação pós-layout. SDF. CVC. Cobertura funcional. Análise de potência. SystemVerilog.

Abstract

The democratization of integrated circuit development enabled by open-source tools still finds one of its main obstacles in functional verification. This work investigates the feasibility of a Python-based verification flow applied to advanced stages of digital design, including functional coverage, structural code coverage, post-layout simulation, and power estimation. To evaluate this approach, a complete verification environment was developed for a serial communication controller widely used in embedded systems, using an open integrated circuit manufacturing technology as the reference platform. The results demonstrate that the proposed approach is capable of validating the design across different abstraction levels while achieving high coverage and producing consistent power consumption estimates. The experiments indicate that the integration of the Python ecosystem with open-source tools represents a viable, accessible, and sufficiently robust alternative for research, education, and integrated circuit development.

Keywords: Functional verification. UVM. cocotb. pyuvm. I2C. SKY130. Open PDK. Functional coverage. Code coverage. Power analysis. SystemVerilog. SDF. CVC.

Sumário

Lista de Figuras

Lista de abreviaturas e siglas

ACK	*Acknowledge* (Reconhecimento)
API	*Application Programming Interface*
ARM	*Advanced RISC Machine*
AXI	*Advanced eXtensible Interface*
BFM	*Bus Functional Model*
CMOS	*Complementary Metal-Oxide-Semiconductor*
CRV	*Constrained Random Verification*
CSP	*Constraint Satisfaction Problem*
DUT	*Device Under Test*
FIFO	*First In, First Out*
FSM	*Finite State Machine*
GLS	*Gate-Level Simulation*
HDL	*Hardware Description Language*
I2C	*Inter-Integrated Circuit*
IEEE	*Institute of Electrical and Electronics Engineers*
IP	*Intellectual Property*
MSB	*Most Significant Bit*
NACK	*Not Acknowledge*
OSS	*Open Source Software*
PDK	*Process Design Kit*
RISC	*Reduced Instruction Set Computer*
RTL	*Register Transfer Level*
SCL	*Serial Clock Line*
SDA	*Serial Data Line*

SDC	*Synopsys Design Constraints*
SDF	*Standard Delay Format*
SKY130	Processo de 130 nm de código aberto da SkyWater Technology Foundry
SMT	*Satisfiability Modulo Theories*
SPEF	*Standard Parasitic Exchange Format*
STA	*Static Timing Analysis*
SV	SystemVerilog
SVA	*SystemVerilog Assertions*
TC	Caso de Teste (*Test Case*)
UVM	*Universal Verification Methodology*
VCD	*Value Change Dump*
VCS	*Verilog Compiler Simulator*
VHDL	*VHSIC Hardware Description Language*
VHPI	*VHDL Procedural Interface*
VIF	*Virtual Interface*
VPI	*Verilog Procedural Interface*

1 Introdução

A verificação funcional representa uma das etapas mais críticas, complexas e onerosas no ciclo de desenvolvimento de circuitos integrados (*ICs*). À medida que a densidade de transistores e a complexidade arquitetural dos *designs* aumentam, garantir a corretude lógica e temporal do hardware antes de sua fabricação em silício torna-se um desafio primordial.

Estatisticamente, as tarefas de verificação e integração representam um dos maiores gargalos no desenvolvimento de sistemas complexos, chegando a consumir cerca de 70% do tempo e dos recursos totais do ciclo de desenvolvimento, conforme explanado por Wile, Goss e Roesner (2005). Em conformidade com Wöhr *et al.* (2023), existe um conflito inerente nesses processos: a tentativa de reduzir o tempo frequentemente esbarra na alta complexidade e nos custos atrelados a essas rigorosas etapas.

Historicamente, a *Universal Verification Methodology* (UVM), implementada sobre a linguagem SystemVerilog, consolidou-se como o padrão industrial incontestável para essa finalidade. Contudo, essa abordagem tradicional impõe barreiras substanciais: uma curva de aprendizado acentuada, verbosidade excessiva em sua implementação e uma pesada biblioteca de classes base. Mais criticamente, a dependência de simuladores proprietários, cujas licenças possuem custos proibitivos para pequenas empresas, *startups* e o ambiente acadêmico, restringe o acesso às metodologias mais avançadas de verificação (Abdelbaky; Dessouky; Salem, 2025).

A ascensão do movimento *Open Silicon* e a disponibilização de *Process Design Kits* (PDKs) de código aberto, como o SKY130 da Google/SkyWater, democratizaram o *design* físico de *chips*. No entanto, a etapa de verificação manteve-se como um gargalo no fluxo inteiramente livre. Ferramentas e simuladores *open-source*, como o Icarus Verilog[1] e o Verilator, embora eficientes para simulação RTL e GLS com SDF parcial, carecem de suporte completo à norma IEEE 1800 (SystemVerilog) e, no caso do Icarus, de *timing checks* SDF, inviabilizando ambientes UVM nativos em SV e a validação temporal completa na GLS sem ferramentas complementares (por exemplo, simulador CVC).

É nesse cenário que o ecossistema da linguagem Python emerge como alternativa poderosa e viável. *Frameworks* de co-simulação como o *cocotb* operam em conjunto com bibliotecas estruturais como o *pyuvm* para contornar as deficiências dos simuladores de código aberto. Ao interagir com o simulador por meio de interfaces procedurais padronizadas (VPI/VHPI), o *cocotb* permite que o simulador *open-source* seja responsável apenas pela execução do código RTL, enquanto toda a complexidade do ambiente de verificação é executada em Python (Ankitha; Aradhya, 2021).

Apesar da crescente adoção e eficácia comprovada do Python na verificação em nível RTL, ainda se verifica uma lacuna na literatura técnica quanto à robustez dessas ferramentas abertas em cenários de validação física completa, envolvendo extração de atividade

[1] https://steveicarus.github.io/iverilog/

de comutação para análise de potência em nós tecnológicos reais como o SKY130. Este trabalho tem como objetivo preencher essa lacuna.

1.1 Objetivos

Objetivo Geral

Validar a eficiência e a completude de um fluxo de verificação funcional baseado em Python/UVM (*cocotb* e *pyuvm*), demonstrando sua aplicabilidade e robustez em estágios avançados de projeto de circuitos integrados, especificamente na verificação funcional completa com cobertura estruturada e na análise de potência, utilizando o PDK SKY130 como nó tecnológico de referência.

Objetivos Específicos

- Analisar e documentar a migração dos componentes da arquitetura UVM tradicional (SystemVerilog) para a implementação em Python utilizando a biblioteca *pyuvm*.
- Desenvolver e implementar um ambiente de verificação completo - agentes, *drivers*, monitores, *scoreboard* e cobertura funcional - para o controlador I2C *i2c_master_axil*.
- Avaliar a cobertura de código (linhas e ramos) do DUT sob a suíte de testes implementada, utilizando Verilator e *lcov*.
- Executar simulação funcional pós-layout com anotação SDF sobre a *netlist* SKY130, documentando limitações de *timing check* no Icarus Verilog.
- Extrair os vetores de atividade de comutação (VCD) da GLS e utilizá-los para alimentar ferramentas *open-source* de estimativa de consumo de potência (OpenSTA).
- Realizar uma avaliação qualitativa e quantitativa do fluxo proposto em comparação a um *testbench* equivalente em SystemVerilog, destacando métricas de produtividade (tempo de compilação, ciclo de iteração), linhas de código, facilidade de depuração, capacidade de GLS e cobertura.

1.2 Estrutura do Trabalho

O restante deste trabalho está organizado como se segue. O Capítulo 2 apresenta a fundamentação teórica, abordando o protocolo I2C, a metodologia UVM, o ecossistema Python de verificação e os conceitos de cobertura e análise de potência. O Capítulo 3 descreve a metodologia e a arquitetura do ambiente implementado. O Capítulo 4 apresenta os resultados em RTL, pós-layout e potência. O Capítulo 5 apresenta a comparação com um *testbench* SystemVerilog equivalente sobre o Verilator, incluindo métricas quantitativas reais de tempo

de execução, linhas de código e capacidades de cada ambiente. O Capítulo 6 sintetiza as contribuições, limitações (incluindo *timing checks* em SDF) e trabalhos futuros, com destaque para a integração do simulador CVC ao fluxo cocotb como extensão prioritária.

2 Fundamentação Teórica

Este capítulo apresenta os conceitos teóricos e tecnologias fundamentais que embasam o desenvolvimento e a análise propostos. Inicia-se com o protocolo I2C e o DUT utilizado, seguido pela metodologia UVM e seus componentes, pelo ecossistema Python de verificação, pelas métricas de cobertura e, por fim, pelos conceitos de modelagem e estimativa de potência em CMOS.

2.1 O Protocolo I2C

O protocolo Inter-Integrated Circuit (I2C) é um barramento serial síncrono desenvolvido pela Philips Semiconductors (hoje NXP) em 1982. Sua popularidade deriva da simplicidade: apenas dois fios condutores, *Serial Clock Line* (SCL) e *Serial Data Line* (SDA), interligam múltiplos dispositivos em topologia de barramento aberto (*open-drain*), com resistores de *pull-up* mantendo o estado lógico alto em repouso (NXP Semiconductors, 2014).

2.1.1 Sinalização e Temporização

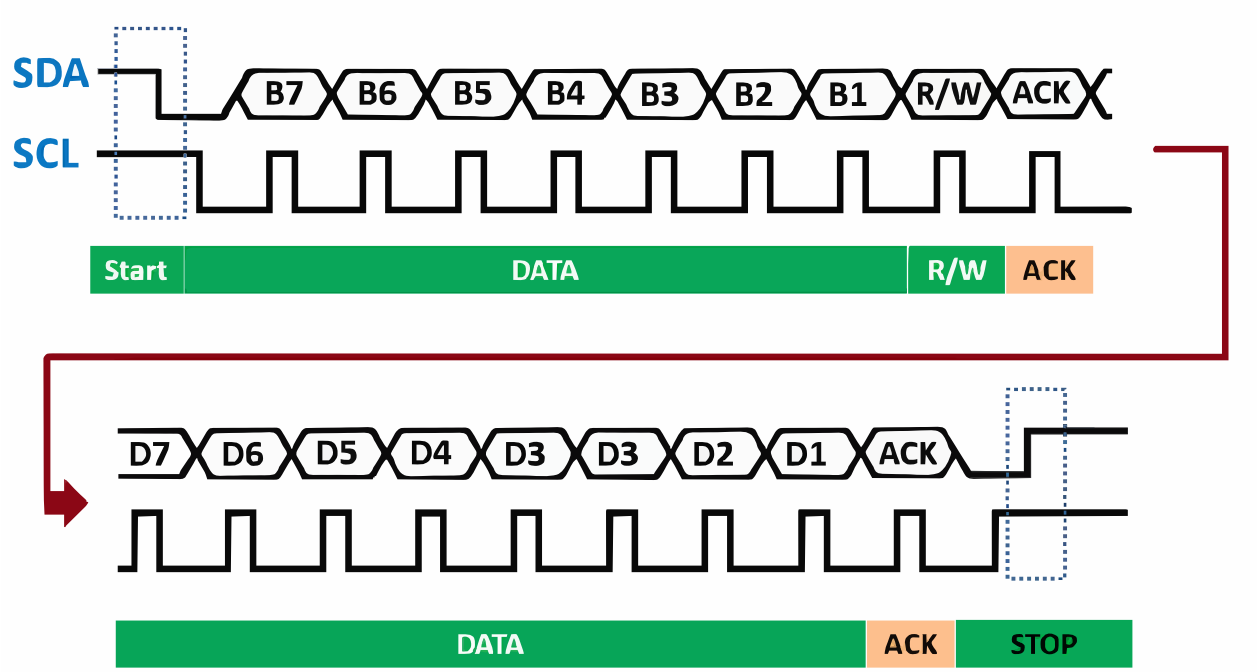
Uma transação I2C inicia com uma condição de START: o mestre provoca uma transição de SDA de alto para baixo enquanto SCL permanece alto. Analogamente, uma condição de STOP ocorre com SDA transitando de baixo para alto com SCL alto. Entre as condições de borda, os dados são transmitidos bit a bit, MSB primeiro, na borda ascendente do SCL, sendo amostrados quando SCL está alto. Cada byte é seguido de um bit de reconhecimento (ACK), em que o receptor puxa SDA para baixo.

O endereçamento opera em modo de 7 bits (ou 10 bits na extensão), permitindo teoricamente até 128 (ou 1024) dispositivos no mesmo barramento. No DUT utilizado neste trabalho, o escravo interno possui endereço fixo `0x63`. Qualquer tentativa de comunicação com endereço diverso resulta em NACK (*Not Acknowledge*), condição central em vários casos de teste desta suíte.

2.1.2 Diagrama de Temporização I2C

A Figura 1 ilustra uma transação de escrita completa, desde a condição START até a condição STOP.

Figura 1 – Diagrama de temporização de uma transação I2C de escrita. O mestre sinaliza START (SDA desce com SCL alto), transmite o endereço de 7 bits mais $R/\bar{W} = 0$, aguarda ACK do escravo, transmite o byte de dados, aguarda ACK e sinaliza STOP (SDA sobe com SCL alto).



Fonte: PIJA_Education (2026)

2.2 O Device Under Test (DUT): i2c_master_axil

O IP *i2c_master_axil* é um controlador I2C mestre de código aberto desenvolvido por Alex Forencich[1], amplamente utilizado em projetos acadêmicos e industriais. Sua interface com o processador hospedeiro é realizada por meio de um barramento AXI-Lite (*Advanced eXtensible Interface – Lite*), tornando-o facilmente integrável a sistemas baseados em ARM ou RISC-V.

2.2.1 Mapa de Registradores

O controlador expõe quatro registradores de 32 bits mapeados em memória (Tabela 1):

[1] https://github.com/alexforencich/verilog-i2c

Tabela 1 – Mapa de registradores do *i2c_master_axil*.

Endereço	**Nome**	**Descrição**
0x0	STATUS	*busy*[0], *miss_ack*[3], *cmd_empty*[8], *cmd_full*[9], *cmd_ovf*[10], *wr_empty*[11], *wr_full*[12], *wr_ovf*[13], *rd_empty*[14], *rd_full*[15]
0x4	COMMAND	*addr*[6:0], *start*[8], *read*[9], *write*[10], *write_multiple*[11], *stop*[12]
0x8	DATA	*data*[7:0], *data_valid*[8](R), *data_last*[9](W)
0xC	PRESCALE	*prescale*[15:0] – divisor de frequência SCL

2.2.2 FIFOs de Comando

O controlador implementa FIFOs internas para desacoplar o processador do protocolo I2C: a FIFO de comandos (*CMD FIFO*) armazena transações pendentes e a FIFO de escrita (*WR FIFO*) armazena os dados a serem enviados ao escravo. Uma FIFO de leitura (*RD FIFO*) armazena os dados recebidos. Esse mecanismo permite que o software enfileire múltiplas operações e as execute de forma assíncrona, melhorando o *throughput* do sistema.

2.2.3 Wrapper RTL

Para fins de verificação, foi desenvolvido um *wrapper* Verilog (*i2c_rtl_wrapper*) que instancia o *i2c_master_axil* e um escravo I2C compatível (*i2c_slave*), conectados ao mesmo barramento físico simulado. O escravo responde exclusivamente ao endereço 0x63, implementando uma memória de registradores de 256 posições de 8 bits, usada para validar loopback de escrita e leitura.

2.3 Metodologia UVM e seus Componentes

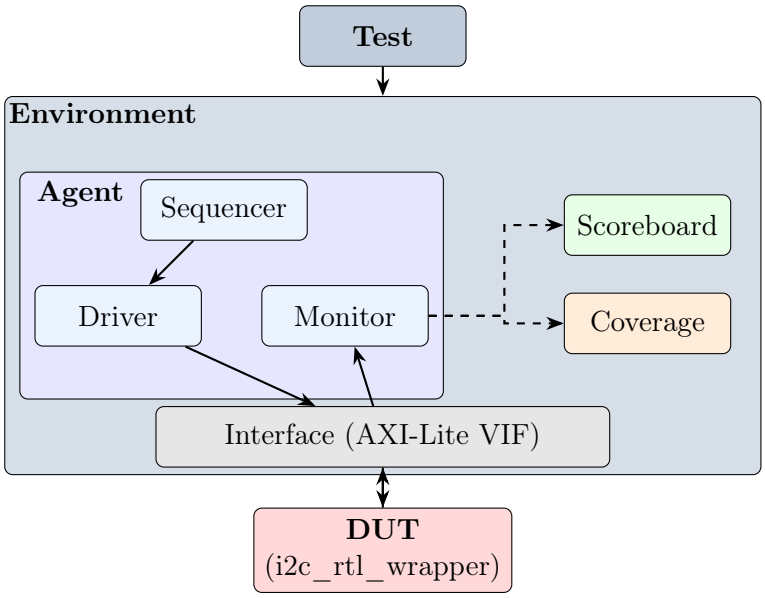
A *Universal Verification Methodology* (UVM) é uma metodologia padronizada pelo consórcio Accellera (norma IEEE 1800.2) para a criação de ambientes de verificação reutilizáveis, escaláveis e orientados a cobertura (Liu *et al.*, 2023). Originalmente implementada em SystemVerilog, a UVM define uma hierarquia de classes e um mecanismo de fases que organiza o ciclo de vida do ambiente de testes.

Abdelbaky, Dessouky e Salem (2025) demonstraram que o consumo de esforço de verificação pode chegar a 40–50% do tempo total de projeto de um *chip*, justificando a busca por metodologias mais ágeis.

2.3.1 Arquitetura UVM

A Figura 2 apresenta a arquitetura hierárquica de um testbench UVM. Os componentes principais são:

Figura 2 – Arquitetura hierárquica do testbench UVM implementado. O Test instancia o Environment, que contém o Agent (Sequencer, Driver, Monitor), o Scoreboard e o módulo de Cobertura Funcional.



Fonte: Elaborado pelo autor (2026)

Test (`uvm_test`): Instância de nível mais alto, responsável por selecionar e executar a sequência de testes. Cada caso de teste (*test case*) é uma subclasse que define uma sequência específica de transações.

Environment (`uvm_env`): Contêiner que agrega os subcomponentes: agentes, *scoreboard* e módulo de cobertura. Facilita a reutilização ao encapsular toda a infraestrutura de verificação.

Agent (`uvm_agent`): Encapsula o Sequencer, o Driver e o Monitor. Pode operar em modo ativo (gerando estímulos) ou passivo (apenas observando o DUT).

Sequencer (`uvm_sequencer`): Árbitro de sequências; coordena o fluxo de itens de sequência (*sequence items*) para o Driver.

Driver (`uvm_driver`): Converte objetos de transação de alto nível em sinais físicos do barramento AXI-Lite, realizando as operações de leitura e escrita nos registradores do DUT.

Monitor (`uvm_monitor`): Observa passivamente o barramento, captura transações e as publica via *Analysis Port* (`uvm_analysis_port`) para o Scoreboard e o módulo de cobertura.

Scoreboard (`uvm_scoreboard`): Compara os resultados observados pelo Monitor com os valores esperados definidos pelo modelo de referência, reportando aprovações ou falhas.

Coverage: Componente que contabiliza os bins de cobertura funcional a partir das transações reportadas pelo Monitor.

2.3.2 Fases UVM (*Phasing*)

A UVM define um conjunto ordenado de fases que organizam a construção, conexão e execução do ambiente. As mais relevantes para este trabalho são:

1. **build_phase**: instanciação dos subcomponentes via *factory*;
2. **connect_phase**: conexão das portas de análise entre Monitor, Scoreboard e Coverage;
3. **run_phase**: execução concorrente de sequências; é a fase de simulação propriamente dita;
4. **report_phase**: emissão do relatório final de cobertura e scorecard.

2.4 O Ecossistema de Verificação em Python

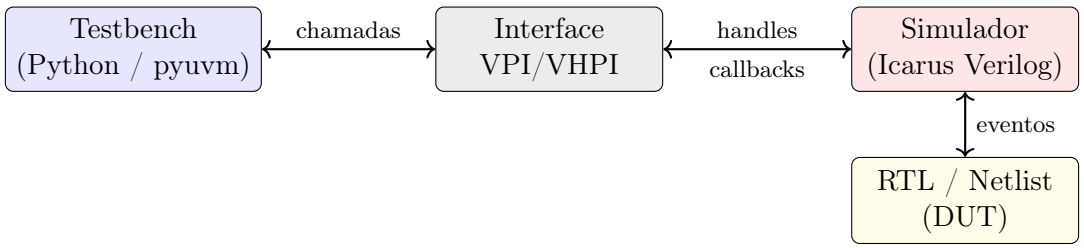
2.4.1 Cocotb

O *cocotb*[2] (*Coroutine Cosimulation Testbench*) é um *framework* de código aberto que permite a interação direta entre um testbench escrito em Python e um simulador HDL convencional (Ankitha; Aradhya, 2021). O *cocotb* elimina a necessidade de *wrappers* de *testbench* em HDL (como blocos `initial`/`always` em SystemVerilog para controle de estímulos), comunicando-se com o simulador por meio de interfaces procedurais padronizadas (VPI para Verilog, VHPI para VHDL). Cabe notar que *wrappers* RTL do DUT — como o *i2c_rtl_wrapper* utilizado neste trabalho — continuam necessários para encapsular e interconectar os módulos sob verificação.

A principal vantagem apontada é o ganho de produtividade: por ser uma linguagem interpretada e de tipagem dinâmica, Python permite a modificação e reexecução rápida de testes sem recompilar o DUT. O paralelismo entre os componentes do ambiente (Driver, Monitor) é expresso por meio de co-rotinas (`async`/`await`), gerenciadas internamente pelo *cocotb* com semântica similar ao `fork`/`join` do SystemVerilog.

A Figura 3 ilustra o fluxo de co-simulação.

Figura 3 – Fluxo de co-simulação do *cocotb*. O testbench Python interage com o simulador (Icarus Verilog) via interface VPI, enquanto o simulador executa o código RTL do DUT. Eventos e *callbacks* propagam as mudanças de sinal de volta ao testbench.



Fonte: Elaborado pelo autor (2026)

[2] https://www.cocotb.org

2.4.2 Pyuvm

Para prover o rigor estrutural da UVM dentro do ecossistema Python, desenvolveu-se o *pyuvm*[3]. Essa biblioteca reimplementa os conceitos fundamentais da UVM utilizando as vantagens nativas do Python. Segundo Abdelbaky, Dessouky e Salem (2025), o *pyuvm* simplifica mecanismos complexos do SystemVerilog, como os registros de fábrica (*factory registration*) e a hierarquia de fases (*phasing*), tornando a escrita de testes menos verbosa enquanto mantém a arquitetura modular.

Ao contrário da UVM em SV, o *pyuvm* utiliza herança Python padrão e decoradores para os mecanismos de fábrica, e emprega `asyncio` para a concorrência entre fases. Isso elimina as macros `` `uvm_component_utils `` e `` `uvm_object_utils `` típicas do SV, substituindo-as por um simples decorator `@cocotb.test`.

2.4.3 Comparação Técnica: SV-UVM vs Python-UVM

A Tabela 2 sintetiza as diferenças técnicas fundamentais entre as duas abordagens, conforme identificadas por Abdelbaky, Dessouky e Salem (2025).

Tabela 2 – Comparação entre UVM-SystemVerilog e UVM-Python.

Aspecto	**SV-UVM**	**Python-UVM (cocotb + pyuvm)**
Tipagem de dados	Estática; `reg`, `logic`, 4-estados (X,Z)	Dinâmica; objetos Python; X/Z via `LogicArray` do cocotb
Concorrência	`fork/join` nativo em SV	`async/await` via asyncio
Randomização CRV	Solucionador nativo (IEEE 1800)	Biblioteca externa; maturidade menor
Recompilação	Necessária a cada modificação	Não necessária (linguagem interpretada)
Curva de aprendizado	Alta; sintaxe e macros proprietárias	Menor; Python amplamente conhecido
Ferramentas	Simuladores proprietários (VCS, Questa)	Icarus Verilog, Verilator (open-source)
Custo de licença	Proibitivo para academia	Zero
Integração com dados	Limitada	Total: NumPy, pandas, scikit-learn

A robustez desta abordagem em cenários críticos foi demonstrada por Ashmanskas *et al.* (2023) no CERN, onde o *cocotb* foi adotado no desenvolvimento dos ASICs do detector ATLAS do HL-LHC, em detrimento do UVM-SystemVerilog, explicitamente pela complexidade e curva de aprendizado deste último. O estudo demonstrou que um ambiente Python foi capaz de simular interações pós-síntese realistas entre múltiplos ASICs e gerenciar injeções de falhas por radiação, validando a tolerância do *design* em nível de *netlist*.

[3] https://github.com/pyuvm/pyuvm

2.5 Cobertura Funcional e de Código

2.5.1 Cobertura Funcional

A cobertura funcional mede o quanto das funcionalidades especificadas do DUT foram exercitadas pelo ambiente de testes, de forma independente da estrutura do código RTL. Ela é definida por meio de grupos de cobertura (*covergroups*) e pontos de cobertura (*coverpoints*) que agrupam condições lógicas relevantes, os *bins*, cuja combinatória deve ser exercitada para que a verificação seja considerada completa (Liu *et al.*, 2023).

No ambiente implementado, os bins de cobertura funcional foram definidos para capturar as principais funcionalidades do controlador I2C:

1. Acesso de escrita ao registrador STATUS (`REG_WRITE, 0`);
2. Acesso de leitura ao registrador STATUS (`REG_READ, 0`);
3. Acesso de escrita ao registrador PRESCALE (`REG_WRITE, 0xC`);
4. Acesso de leitura ao registrador PRESCALE (`REG_READ, 0xC`);
5. Operação I2C de escrita bem-sucedida (`I2C_WRITE, ACK`);
6. Operação I2C de escrita com NACK (`I2C_WRITE, NACK`);
7. Operação I2C de leitura bem-sucedida (`I2C_READ, ACK`);
8. Operação I2C de leitura com NACK (`I2C_READ, NACK`);
9. Evento de overflow de FIFO (`FIFO_OVF`).

O teste TC_19 (*I2cCoverageTest*) é dedicado a exercitar todos os bins remanescentes, garantindo a cobertura funcional completa ao final da suíte.

2.5.2 Cobertura de Código

A cobertura de código avalia, em nível estrutural, quais linhas e ramos do código RTL foram executados durante a simulação. É obtida instrumentando o simulador para contar execuções e utilizando *lcov* para gerar os relatórios. As métricas relevantes são:

Cobertura de Linhas (*Line Coverage*) Percentual de linhas de código RTL executadas ao menos uma vez. Ideal para identificar código morto ou funcionalidades não exercitadas.

Cobertura de Ramos (*Branch Coverage*) Percentual de todos os ramos de decisão (verdadeiro/falso de cada `if`/`case`) que foram exercitados. Mais rigorosa que a cobertura de linhas, pois detecta caminhos condicionais não testados.

2.6 Estimativa de Potência em CMOS

Em circuitos integrados baseados em tecnologia CMOS, a dissipação de potência total (P_{total}) é modelada pela soma de componentes dinâmicos e estáticos (Rabaey; Chandrakasan; Nikolić, 2003):

$$P_{total} = P_{din} + P_{est} \tag{2.1}$$

A potência dinâmica (P_{din}) ocorre devido à carga e descarga das capacitâncias parasitas durante as transições lógicas dos nós:

$$P_{din} = \alpha \cdot C_L \cdot V_{DD}^2 \cdot f \tag{2.2}$$

onde α é o fator de atividade de chaveamento, C_L é a capacitância de carga, V_{DD} a tensão de alimentação e f a frequência de operação.

A potência estática (P_{est}) decorre das correntes de fuga dos transistores mesmo em repouso. À medida que os nós tecnológicos diminuem, as correntes de fuga tornam-se progressivamente mais significativas (Weste; Harris, 2010).

2.6.1 Arquivo VCD e Arquivo SPEF

A precisão de qualquer estimativa de potência dinâmica depende do fator de atividade α. Estimativas baseadas em probabilidades estáticas são imprecisas, pois ignoram a correlação temporal dos dados. Para obter α com precisão, utiliza-se a simulação funcional para extrair o arquivo *Value Change Dump* (VCD), que registra o estado de cada nó do circuito a cada instante de tempo.

O arquivo SPEF (*Standard Parasitic Exchange Format*) complementa o VCD ao fornecer as capacitâncias e resistências parasitas de cada rede da *netlist*, obtidas após a síntese e o roteamento. O cruzamento dos dados de comutação (VCD) com os modelos parasitários (SPEF) e a biblioteca de células (Liberty) permite que o OpenSTA calcule as potências dinâmica e estática com alta fidelidade.

2.7 O PDK SKY130 e o Fluxo de Síntese

O SKY130[4] é um *Process Design Kit* (PDK) de código aberto desenvolvido pela parceria entre Google e SkyWater Technology para o nó tecnológico de 130 nm. Sua disponibilização pública em 2020 representa um marco no movimento *Open Silicon*, ampliando o acesso ao fluxo completo de projeto de circuitos integrados, da concepção à fabricação.

Para a síntese lógica e o place-and-route do IP *i2c_master_axil*, foi utilizado o **LibreLane**[5], fluxo *open-source* baseado em OpenLane que automatiza síntese lógica (Yosys),

[4] https://github.com/google/skywater-pdk
[5] https://librelane.readthedocs.io/en/stable/

posicionamento e roteamento (OpenROAD), além da geração de *netlist* estrutural, SDF e SPEF no diretório `postlayout/` do repositório do projeto. Esses artefatos alimentam a simulação pós-layout e a estimativa de potência; o detalhamento das métricas de área e *timing* de síntese não constitui foco deste TCC.

2.7.1 SDF, GLS e verificação temporal na simulação

O arquivo SDF (*Standard Delay Format*) descreve atrasos de portas e interconexões extraídos após síntese ou após *layout*. Na simulação em nível de portas (*Gate-Level Simulation*, GLS), o SDF é anotado sobre a *netlist* para aproximar o comportamento temporal do circuito fabricado. Além dos atrasos de propagação (`IOPATH`, `INTERCONNECT`), o SDF pode conter entradas `TIMINGCHECK`, que especificam janelas de *setup*, *hold* e relações de recuperação/remoção entre sinais, elemento central na verificação pós-layout, pois detecta violações de temporização durante a simulação funcional.

Simuladores comerciais aplicam integralmente essas checagens. No ecossistema *open-source*, o Icarus Verilog aceita anotação SDF para atrasos, porém emite advertências `TIMING CHECK not supported` e não executa as verificações de *setup*/*hold* embutidas no arquivo. O simulador CVC (*Circuit Validation Companion*), mantido por Tachyon Design Automation e disponível no repositório *cambridgehackers/open-src-cvc*[6], oferece suporte a SDF com *timing checks* e integração com fluxos de verificação baseados em Python, sendo uma alternativa promissora para estender o ambiente descrito neste trabalho. O CVC é distribuído sob a *OSS CVC Dual Licensing Modified Artistic License*, que permite uso gratuito para projetos pessoais e acadêmicos, mas **não** é uma licença permissiva para uso comercial sem autorização da Tachyon Design Automation.

[6] https://github.com/cambridgehackers/open-src-cvc

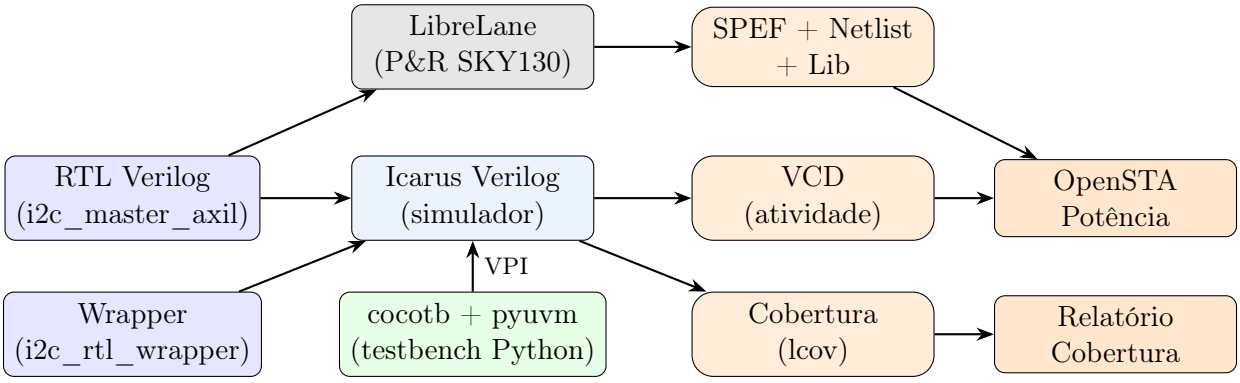
3 Metodologia e Implementação

Este capítulo descreve as escolhas metodológicas e os detalhes de implementação do ambiente de verificação disponível no repositório (Soares, 2026). A abordagem é experimental-aplicada: definição do DUT, implementação UVM em Python, cobertura de código (Verilator), simulação pós-layout com SDF (Icarus) e análise de potência (OpenSTA).

3.1 Visão Geral do Fluxo

A Figura 4 apresenta o fluxo completo, desde o código RTL até os relatórios de cobertura e potência.

Figura 4 – Fluxo completo de verificação funcional e análise de potência. O ambiente Python (cocotb/pyuvm) co-simula com o Icarus Verilog para gerar artefatos VCD e dados de cobertura. Paralelamente, a *netlist* e o SPEF gerados pelo LibreLane alimentam simulação pós-layout (Icarus + SDF) e o OpenSTA para estimativa de potência.

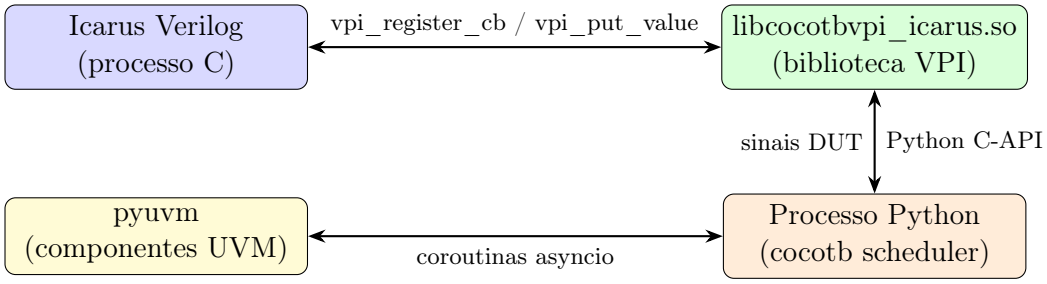


Fonte: Elaborado pelo autor (2026)

3.2 Mecanismo de Co-simulação: cocotb e a Interface VPI

A co-simulação é o elemento central que torna possível executar código Python como controlador de um simulador Verilog. O *cocotb* (*Coroutine-based Co-simulation Test Bench*) implementa essa integração por meio da *Verilog Procedural Interface* (VPI), interface de programação padronizada pela norma IEEE 1364 (IEEE..., 2005). A Figura 5 ilustra como os processos se comunicam.

Figura 5 – Arquitetura de co-simulação cocotb/VPI. O Icarus carrega a biblioteca VPI compartilhada; o *scheduler* Python controla os pontos de retomada via *callbacks* VPI, acionando as corrotinas UVM a cada evento de simulação.



Fonte: Elaborado pelo autor (2026)

Ao ser invocado com o argumento `-m libcocotbvpi_icarus`, o Icarus Verilog carrega a biblioteca compartilhada do cocotb como um módulo VPI. Essa biblioteca registra um *callback* de início de simulação (`cbStartOfSimulation`) que inicializa o interpretador Python embarcado. A partir daí, cada *trigger* do ambiente de teste, uma borda de clock, a mudança de um sinal, ou simplesmente um atraso de tempo, corresponde a um *callback* VPI (`cbValueChange` ou `cbAfterDelay`). Quando o simulador processa esse evento, o *scheduler* do cocotb retoma a corrotina Python correspondente via `asyncio`. O controle então retorna ao simulador quando a corrotina entrega o próximo *trigger* (por exemplo, `await RisingEdge(dut.clk)`).

Uma vantagem importante dessa arquitetura é que o simulador Verilog permanece responsável exclusivamente pela avaliação do modelo RTL e pela progressão do tempo de simulação. Toda a lógica do ambiente de verificação, a hierarquia UVM, o Scoreboard, o módulo de cobertura, o BFM, reside no processo Python, aproveitando o ecossistema de bibliotecas Python sem qualquer modificação no simulador.

3.2.1 Inicialização do Ambiente e Fases UVM

O arquivo `uvm/components/__init__.py` registra os módulos de teste ao cocotb. Quando a simulação inicia, o cocotb identifica os testes declarados com o decorador `@cocotb.test()` no módulo `components/test_i2c.py` e os executa sequencialmente. Cada teste instancia o ambiente UVM de forma independente, garantindo isolamento de estado entre casos de teste. A Tabela 3 descreve as fases UVM executadas em cada teste.

Tabela 3 – Fases UVM e suas responsabilidades no ambiente implementado.

Fase	**Responsável**	**Ação**
`build_phase`	`Env`	Cria Agente, Scoreboard e Cobertura; conecta portas de análise
`connect_phase`	`Agent`	Liga `seq_item_port` do Driver ao Sequenciador; conecta `ap` do Monitor ao Scoreboard e Cobertura
`run_phase`	Todos	Driver aguarda itens; Monitor observa barramento; Scoreboard verifica; Sequências geram estímulos
`report_phase`	Scoreboard + Cobertura	Emite estatísticas de PASS/FAIL e relatório de bins

3.3 Componentes UVM Implementados em Python

O ambiente de verificação foi construído sobre a biblioteca *pyuvm*, que transpõe a hierarquia de classes UVM para Python e se integra ao *framework* de simulação cocotb. Os componentes seguem a separação de responsabilidades do padrão UVM: `defs.py` centraliza constantes e o mapa de registradores; `seq_item.py` define a transação; `bfm.py` abstrai o protocolo de barramento; `driver.py` converte transações em estímulos; `monitor.py` retransmite transações completadas; `agent.py` agrega Driver, Sequenciador e Monitor; `env.py` instancia e interconecta tudo; `scoreboard.py` verifica a correção; `coverage.py` rastreia cobertura funcional; e `seq.py` implementa as sequências de estímulo. As subseções seguintes descrevem cada componente com base no código-fonte implementado.

3.3.1 Definições Globais (`defs.py`)

O arquivo `defs.py` centraliza todas as constantes do protocolo, eliminando literais mágicos dispersos pelo código. O clock do sistema é de 50 MHz (`CLK_HZ = 50_000_000`), com período de 20 ns (`CLK_P = 20`).

A enumeração `CmdType` distingue quatro tipos de operação que um `I2cSeqItem` pode representar: `REG_WRITE` (0) e `REG_READ` (1) acessam diretamente um registrador do DUT via barramento AXI4-Lite, sem envolver o protocolo I2C; `I2C_WRITE` (2) e `I2C_READ` (3) compõem transações completas no barramento I2C, com geração das condições de START, endereçamento do escravo, transferência do dado e condição de STOP.

A enumeração `RegAddr` mapeia os quatro registradores do DUT ao campo `addr[3:0]` do barramento AXI4-Lite: `STATUS` (0x00), somente-leitura, contém campos de estado das FIFOs e da FSM como `busy[0]`, `miss_ack[3]`, `cmd_ovf[10]`, `wr_ovf[13]` e `rd_empty[14]`; `COMMAND` (0x04), somente-escrita, codifica nos bits [6:0] o endereço I2C do escravo e nos bits [8:12] os flags de operação; `DATA` (0x08), leitura e escrita, transfere

o byte de dado, na escrita, o bit 9 (DATA_LAST) sinaliza o último byte de uma sequência WRITE_MULTIPLE, e na leitura o bit 8 (DATA_VALID) indica dado disponível; e PRESCALE (0x0C), leitura e escrita, configura o divisor de clock para a temporização I2C nos 16 bits menos significativos.

As flags do registrador COMMAND são definidas como constantes de deslocamento de bit: CMD_START ($1 \ll 8$), CMD_READ ($1 \ll 9$), CMD_WRITE ($1 \ll 10$), CMD_WRITE_MULTIPLE ($1 \ll 11$) e CMD_STOP ($1 \ll 12$). Os bits de STATUS mais utilizados na lógica de verificação e no BFM são STATUS_BUSY ($1 \ll 0$), STATUS_MISS_ACK ($1 \ll 3$) e STATUS_RD_EMPTY ($1 \ll 14$). O endereço do escravo I2C instanciado no *wrapper* RTL é fixado em VALID_SLAVE_ADDR = 0x63; transações para qualquer outro endereço resultam em NACK, comportamento explorado nos casos de teste de tratamento de erro.

3.3.2 Item de Sequência (seq_item.py)

A classe I2cSeqItem herda de uvm_sequence_item e encapsula uma transação completa. O construtor, mostrado no Código 3.1, recebe quatro parâmetros de estímulo com valores padrão: cmd_type (padrão I2C_WRITE), slave_addr (padrão VALID_SLAVE_ADDR, mascarado a 7 bits), reg_addr (padrão 0, mascarado a 8 bits) e payload (padrão 0, mascarado a 32 bits). Dois campos adicionais, read_data e nack_seen, são inicializados a zero/False e preenchidos pelo Driver após a execução da transação no barramento.

```
class I2cSeqItem(uvm_sequence_item):
    def __init__(self, name, cmd_type=CmdType.I2C_WRITE,
                 slave_addr=VALID_SLAVE_ADDR, reg_addr=0, payload=0):
        super().__init__(name)
        self.cmd_type   = cmd_type
        self.slave_addr = slave_addr & 0x7F
        self.reg_addr   = reg_addr   & 0xFF
        self.payload    = payload    & 0xFFFFFFFF
        # Campos de resposta, preenchidos pelo Driver
        self.read_data  = 0
        self.nack_seen  = False
```

Listagem 3.1 – Item de sequência (seq_item.py).

O método __str__ formata todos os campos em uma linha para facilitar a leitura dos logs de simulação, exibindo o nome do tipo de comando, endereços e valores em hexadecimal.

3.3.3 BFM (*Bus Functional Model*, bfm.py)

O BFM abstrai toda a mecânica de *signaling* dos barramentos AXI4-Lite e I2C, expondo métodos de alto nível ao Driver. A classe Bfm é implementada como um *singleton*: __new__ verifica _instance antes de criar um novo objeto, e __init__ usa a guarda hasattr(self, 'initialized') para executar a inicialização apenas uma vez. Na inicialização, o BFM obtém a referência ao topo da hierarquia do DUT via cocotb.top, armazena

o valor lógico do PRESCALE em `_fake_prescale = 1` e chama `_init_signals`, que zera todos os sinais de controle AXI4-Lite (`awvalid`, `awaddr`, `wvalid`, `wdata`, `awprot`, `arprot`, `wstrb`, `bready`, `arvalid`, `araddr`, `rready`).

O método auxiliar `_safe_int` (Código 3.2) é fundamental para a portabilidade RTL/GLS. Na simulação pós-layout (*Gate-Level Simulation*), sinais podem apresentar valores `X` (desconhecido) ou `Z` (alta impedância) durante a inicialização, antes de convergir a níveis lógicos estáveis. Nessas situações, a propriedade `is_resolvable` do objeto `LogicArray` do cocotb retorna `False`, e a conversão direta `int(logic_array)` lançaria uma exceção que abortaria a simulação. `_safe_int` trata esse caso adotando o valor `default_val` e registrando a ocorrência em *debug log* via `dut._log.debug`, permitindo que a simulação continue sem interrupção.

```
def _safe_int(self, logic_array, default_val=0, signal_name="signal"):
    if not logic_array.is_resolvable:
        self.dut._log.debug(
            f"Info GLS: {signal_name} contem X/Z "
            f"({logic_array.binstr}). Assumindo {default_val}.")
        return default_val
    return int(logic_array)
```

Listagem 3.2 – Método `_safe_int` para compatibilidade com GLS (`bfm.py`).

O método `clock` gera o clock do sistema em uma corrotina infinita, com suporte a jitter de ± 1 ns controlado pelo parâmetro `jitter`. O método `reset_system` asserta o reset síncrono ativo em nível alto (`rst = 1`) por 10 bordas de subida, insere uma margem de 2 ns de hold e então libera o reset (`rst = 0`), aguardando mais 2 bordas adicionais para estabilização.

O protocolo AXI4-Lite de escrita é implementado em `write_register` (Código 3.3). O método trata escrita no registrador PRESCALE (endereço `0x0C`) de forma especial: armazena o valor desejado em `_fake_prescale` e limita o valor efetivamente escrito no DUT a 4, evitando que valores extremos violem o fechamento de temporização estática (*STA*) durante testes de cobertura. Para os demais registradores, o método asserta `AWVALID` e `WVALID` simultaneamente com o endereço (4 bits), dado (32 bits) e *byte strobes* fixos em `0xF`. O *handshake* dos canais AW e W é tratado de forma independente por meio de duas variáveis booleanas (`aw_done`, `w_done`): em cada iteração, o método amostra `AWREADY` e `WREADY` em fase `ReadOnly` e desaciona o sinal `valid` correspondente apenas após o handshake ser confirmado. Uma margem de 2 ns após cada borda de subida garante conformidade com a análise de temporização pós-layout. Após os canais AW e W, o método aguarda `BVALID` com `BREADY` ativo (canal B de resposta de escrita) e desaciona `BREADY`.

```
async def write_register(self, addr, data):
    if (addr & 0xF) == 0x0C:      # PRESCALE: clamp para timing
        self._fake_prescale = data
        data = 4
    dut = self.dut
```

```
    await RisingEdge(dut.clk)
    await Timer(2, unit="ns")      # margem hold STA
    dut.s_axil_awaddr.value  = addr & 0xF
    dut.s_axil_awvalid.value = 1
    dut.s_axil_wdata.value   = data & 0xFFFFFFFF
    dut.s_axil_wstrb.value   = 0xF
    dut.s_axil_wvalid.value  = 1
    aw_done = w_done = False
    while not (aw_done and w_done):
        await ReadOnly()
        if not aw_done and self._safe_int(dut.s_axil_awready.value) == 1:
            aw_done = True
        if not w_done and self._safe_int(dut.s_axil_wready.value) == 1:
            w_done = True
        await RisingEdge(dut.clk)
        await Timer(2, unit="ns")
        if aw_done: dut.s_axil_awvalid.value = 0
        if w_done:  dut.s_axil_wvalid.value  = 0
    dut.s_axil_bready.value = 1
    while True:
        await ReadOnly()
        if self._safe_int(dut.s_axil_bvalid.value) == 1:
            break
        await RisingEdge(dut.clk)
    await RisingEdge(dut.clk)
    await Timer(2, unit="ns")
    dut.s_axil_bready.value = 0
```

Listagem 3.3 – Transação AXI4-Lite de escrita (`write_register`, `bfm.py`).

O método `read_register` implementa o protocolo AXI4-Lite de leitura: asserta `ARVALID` com o endereço (4 bits) e aguarda `ARREADY`; após a borda confirmatória, desaciona `ARVALID` e asserta `RREADY`, aguardando `RVALID`. Quando `RVALID` é observado em fase `ReadOnly`, o valor de `RDATA` é amostrado via `_safe_int`. Para o endereço 0x0C (PRESCALE), o método retorna `_fake_prescale` em lugar do valor lido do DUT, mantendo a consistência com o modelo de referência do Scoreboard.

As transações I2C compostas são implementadas em `execute_i2c_write` e `execute_i2c_read`. Uma transação de escrita I2C (`execute_i2c_write`) é composta por quatro acessos AXI-Lite em duas fases: na fase 1, escreve em COMMAND o endereço do escravo combinado com `CMD_START | CMD_WRITE`, seguido de escrita em DATA com o ponteiro de registrador (`reg_addr`); na fase 2, escreve em COMMAND com o endereço combinado com `CMD_WRITE | CMD_STOP`, seguido de escrita em DATA com o byte de carga útil com `DATA_LAST` ativo. Após o envio, o método aguarda o barramento desocupar por *polling* de `STATUS_BUSY` via `_wait_not_busy`, com limite de `_TIMEOUT_CYCLES = 50000` ciclos. Em caso de NACK (`STATUS_MISS_ACK` ativo no STATUS), o método limpa a flag escrevendo `STATUS_MISS_ACK` no registrador STATUS e chama `reset_system` para

recuperação.

Uma transação de leitura I2C (`execute_i2c_read`) segue o protocolo *combined format* do I2C: na fase 1, configura o ponteiro no escravo com `CMD_START | CMD_WRITE` em COMMAND e o endereço do registrador com `DATA_LAST` ativo em DATA; na fase 2, emite o comando de leitura com `CMD_START | CMD_READ | CMD_STOP` em COMMAND. O método então aguarda dado disponível via `_wait_rd_available`, que monitora `STATUS_RD_EMPTY` com 10 ciclos de espera entre as sondagens e retorna imediatamente se `STATUS_MISS_ACK` for detectado. Com dado disponível, lê DATA e descarta os bits superiores (`raw & 0xFF`). Após a leitura, verifica novamente STATUS para detectar NACK tardio, aplicando a mesma lógica de limpeza e reset em caso positivo. O método retorna a tupla `(data, nack)`.

3.3.4 Driver (`driver.py`)

A classe `Driver` herda de `uvm_driver`. Na `build_phase`, cria uma *Analysis Port* `ap`; na `start_of_simulation_phase`, instancia (ou recupera a instância *singleton* de) `Bfm`. O laço principal em `run_phase` é mostrado no Código 3.4: a cada iteração, retira um item de `seq_item_port.get_next_item()`, despacha a operação correspondente ao `cmd_type` no BFM, preenche os campos de resposta do item (`read_data` e/ou `nack_seen`) quando aplicável, publica o item completo em `ap.write(item)` e sinaliza conclusão com `seq_item_port.item_done()`.

```
async def run_phase(self):
    while True:
        item = await self.seq_item_port.get_next_item()

        if item.cmd_type == CmdType.REG_WRITE:
            await self.bfm.write_register(item.reg_addr, item.payload)
        elif item.cmd_type == CmdType.REG_READ:
            item.read_data = await self.bfm.read_register(item.reg_addr)
        elif item.cmd_type == CmdType.I2C_WRITE:
            item.nack_seen = await self.bfm.execute_i2c_write(
                item.slave_addr, item.reg_addr, item.payload)
        elif item.cmd_type == CmdType.I2C_READ:
            data, nack = await self.bfm.execute_i2c_read(
                item.slave_addr, item.reg_addr)
            item.read_data = data
            item.nack_seen = nack

        self.ap.write(item)
        self.seq_item_port.item_done()
```

Listagem 3.4 – Laço `run_phase` do Driver (`driver.py`).

Uma decisão de projeto relevante é que a *Analysis Port* pertence ao Driver, e não ao Monitor. Isso mantém o BFM como camada puramente de *signaling*, sem acoplamento ao mecanismo de publicação UVM. O item publicado já carrega os campos de resposta

preenchidos, o que permite que Scoreboard e módulo de Cobertura recebam transações completas, estímulo e resposta, em um único objeto.

3.3.5 Monitor (`monitor.py`)

O Monitor herda de `uvm_subscriber`, o que significa que ele expõe uma `analysis_export` por herança. Na `build_phase`, chama `super().build_phase()` e cria uma segunda *Analysis Port* `ap`. O método `write` repropaga imediatamente o item recebido:

```
class Monitor(uvm_subscriber):
    def build_phase(self):
        super().build_phase()
        self.ap = uvm_analysis_port("ap", self)

    def write(self, item):
        self.ap.write(item)
```

Listagem 3.5 – Monitor como *relay* de transações (`monitor.py`).

Essa topologia de dois estágios, Driver publica em `driver.ap`; Monitor recebe e retransmite em `monitor.ap`, resulta em dois pontos de observação distintos no Agente: `driver_ap` (conectado à Cobertura) e `monitor_ap` (conectado ao Scoreboard). A separação é intencional e refletida nas conexões do Ambiente.

3.3.6 Agente (`agent.py`)

O Agente (`Agent`) herda de `uvm_agent` e encapsula Sequenciador, Driver e Monitor. Na `build_phase`, cria as três instâncias.Na `connect_phase`, conecta `driver.seq_item_port` ao `seqr.seq_item_export` e, em seguida, conecta `driver.ap` à `monitor.analysis_export`. Os dois atributos de saída do Agente, `driver_ap` e `monitor_ap`, são atribuídos diretamente a partir das portas internas, disponibilizando-os para as conexões do Ambiente.

3.3.7 Ambiente (`env.py`)

A classe `Env` herda de `uvm_env` e instancia, na `build_phase`, três componentes: `Agent`, `Coverage` e `Scoreboard`. Na `connect_phase`, disponibiliza o Sequenciador no `ConfigDB` com escopo global (`None, "*"`) sob a chave `"SEQR"`, permitindo que os testes de nível superior o recuperem sem referência direta à hierarquia do Ambiente. As duas conexões de análise são: `agent.driver_ap` → `coverage.analysis_export` e `agent.monitor_ap` → `scoreboard.cmd_export`.

A distinção entre as duas conexões é funcionalmente significativa: a Cobertura recebe transações diretamente do Driver (antes da retransmissão pelo Monitor), enquanto o Scoreboard recebe via Monitor. Na implementação atual, ambos os caminhos carregam o mesmo

objeto item, pois o Monitor apenas repropaga. A separação preserva, porém, a possibilidade futura de o Monitor realizar filtragem ou enriquecimento de dados antes de entregar ao Scoreboard.

3.3.8 Scoreboard (`scoreboard.py`)

O Scoreboard herda de `uvm_component` e mantém dois modelos de referência em memória: `reg_model` (dicionário que espelha os registradores R/W do DUT) e `slave_mem` (dicionário que espelha a memória interna do escravo I2C). A comunicação é feita via `uvm_tlm_analysis_fifo`: o `cmd_export` recebe itens publicados pelo Monitor, uma `uvm_get_port` `cmd_get_port` drena a fila na `run_phase`, e a conexão entre ambos é estabelecida na `connect_phase`.

O conjunto `_READ_ONLY_REGS = {STATUS, COMMAND, DATA}` é declarado em nível de módulo e utilizado para decidir quais escritas devem ou não atualizar o modelo de referência.

A lógica de verificação na `run_phase` opera sobre quatro casos:

- **`REG_WRITE`**: se `reg_addr` não pertence a `_READ_ONLY_REGS`, o valor é armazenado em `reg_model`. Para PRESCALE, aplica a máscara `& 0xFFFF`. Escritas em STATUS ou COMMAND são apenas registradas em log sem atualizar o modelo.
- **`REG_READ`**: se `reg_addr` pertence a `_READ_ONLY_REGS` (STATUS, COMMAND ou DATA), o valor lido é registrado em log como volátil, sem verificação determinista. Para PRESCALE (`reg_model` deve conter o endereço), compara `read_data & 0xFFFF` com `reg_model[reg_addr] & 0xFFFF`; concordância incrementa `pass_count`, discrepância incrementa `fail_count` e emite erro. Leituras sem entrada correspondente no modelo são registradas como *"no reference"*.
- **`I2C_WRITE`**: com ACK (`nack_seen = False`), armazena `payload & 0xFF` em `slave_mem[slave_addr]` e incrementa `pass_count`. Com NACK, verifica se o escravo é o válido (0x63) — NACK do escravo válido é falha (`fail_count`); NACK de escravo inválido é esperado e registrado como aviso.
- **`I2C_READ`**: mesma lógica de NACK do caso de escrita. Com ACK e `slave_addr` presente em `slave_mem`, compara `read_data & 0xFF` com o valor armazenado; concordância incrementa `pass_count`, discrepância incrementa `fail_count`. Leituras sem escrita prévia são registradas como *"no prior write"*.

A decisão de não verificar deterministicamente STATUS, COMMAND e DATA merece justificativa técnica: esses registradores contêm bits que refletem o estado em tempo real das FIFOs e da FSM I2C (`busy`, `miss_ack`, `cmd_empty` etc.), os quais variam assincronamente em relação à lógica de verificação. Verificar seus valores exatos exigiria um modelo de referência temporal completo da FSM I2C, transformando o Scoreboard em uma reimplementação

funcional do DUT e derrotando o propósito da verificação. A estratégia adotada, registrar esses campos para auditoria e verificar apenas PRESCALE e os dados de loopback I2C, segue as boas práticas de verificação orientada a transações Liu *et al.* (2023).

Na `check_phase`, o Scoreboard avalia `fail_count`: se maior que zero, emite erro com o total de falhas e aprovações e lança `assert False`, reprovando o teste.

3.3.9 Módulo de Cobertura Funcional (`coverage.py`)

A classe `Coverage` herda de `uvm_subscriber` e rastreia dois conjuntos Python: `cvg_regs` e `cvg_i2c`, inicializados como conjuntos vazios em `end_of_elaboration_phase`. Os bins esperados são declarados como conjuntos imutáveis em nível de módulo:

```
_EXPECTED_REGS = {
    (CmdType.REG_READ,  int(RegAddr.STATUS)),    # leitura de STATUS
    (CmdType.REG_WRITE, int(RegAddr.PRESCALE)),  # escrita em PRESCALE
    (CmdType.REG_READ,  int(RegAddr.PRESCALE)),  # leitura de PRESCALE
}

_EXPECTED_I2C = {
    (CmdType.I2C_WRITE, True),   # I2C_WRITE para slave valido (ACK)
    (CmdType.I2C_WRITE, False),  # I2C_WRITE para slave invalido (NACK)
    (CmdType.I2C_READ,  True),   # I2C_READ de slave valido (ACK)
    (CmdType.I2C_READ,  False),  # I2C_READ de slave invalido (NACK)
    ("FIFO_OVF",        True),   # overflow de FIFO e limpeza via STATUS
    ("WRITE_MULT",      True),   # uso de CMD_WRITE_MULTIPLE
}
```

Listagem 3.6 – Bins de cobertura esperados (`coverage.py`).

O método `write` classifica cada item recebido. Se `cmd_type` é `REG_WRITE` ou `REG_READ`, adiciona a tupla `(cmd_type, reg_addr)` a `cvg_regs`. Dois bins especiais são derivados de escritas em registradores específicos: escrita em STATUS com os bits 10 ou 13 ativos (`payload & ((1 « 10) | (1 « 13))`) marca o bin `("FIFO_OVF", True)`, isso corresponde exatamente à sequência `I2cFifoOverflowSeq`, que limpa os flags de overflow `cmd_ovf` e `wr_ovf` escrevendo nesses bits; escrita em COMMAND com o bit `CMD_WRITE_MULTIPLE` ativo marca o bin `("WRITE_MULT", True)`. Para `I2C_WRITE` ou `I2C_READ`, adiciona a tupla `(cmd_type, is_valid)` a `cvg_i2c`, onde `is_valid = (slave_addr == VALID_SLAVE_ADDR)`.

Na `report_phase`, o módulo computa `reg_hits` e `i2c_hits` como o tamanho das interseções entre os conjuntos observados e os esperados, e exibe o total `hits/total`. Se houver bins faltantes e a chave `"DISABLE_COVERAGE_ERRORS"` no `ConfigDB` estiver inativa (ou ausente, tratado via `UVMConfigItemNotFound`), o módulo lança `assert False`, reprovando o teste. Essa chave é usada pela classe base `BaseI2cTest` (valor `True`) para suprimir a falha em todos os testes direcionados (TC_01 a TC_18), reservando a

verificação de 100% de cobertura exclusivamente ao `I2cCoverageTest` (TC_19), que a define como `False`.

3.3.10 Sequências (`seq.py`)

As sequências definem o vocabulário de estímulos do ambiente. Cada classe herda de `uvm_sequence` e implementa o método assíncrono `body`, que constrói e despacha itens via o par `start_item`/`finish_item`. O Código 3.7 mostra a sequência de loopback (TC_03), que compõe uma escrita seguida de uma leitura no mesmo registrador do escravo:

```
class I2cLoopbackSeq(uvm_sequence):
    async def body(self):
        w = I2cSeqItem("tc03_w", cmd_type=CmdType.I2C_WRITE,
                       slave_addr=VALID_SLAVE_ADDR,
                       reg_addr=0x05, payload=0x42)
        await self.start_item(w)
        await self.finish_item(w)

        r = I2cSeqItem("tc03_r", cmd_type=CmdType.I2C_READ,
                       slave_addr=VALID_SLAVE_ADDR, reg_addr=0x05)
        await self.start_item(r)
        await self.finish_item(r)
```

Listagem 3.7 – Sequência de loopback I2C (`I2cLoopbackSeq`, TC_03, `seq.py`).

A sequência `I2cCoverageSeq` (TC_19) é composta: após cobrir os bins de registrador (PRESCALE escrita/leitura, STATUS leitura) e os quatro bins I2C básicos (escravo válido e inválido para escrita e leitura, com 15 transações aleatórias adicionais com 70% de probabilidade de escravo válido), ela invoca `I2cFifoOverflowSeq` e `I2cWriteMultipleSeq` como subsequências via `await subseq.start(self.sequencer)`, garantindo os bins `FIFO_OVF` e `WRITE_MULT` sem duplicação de lógica.

3.3.11 Testes de Nível Superior (`test_i2c.py`)

Os testes são implementados em `test_i2c.py` como classes `pyuvm` decoradas com `@pyuvm.test()`. Todos compartilham a função auxiliar `_run_test_flow`, que: instancia o BFM, inicia a corrotina de clock com `cocotb.start_soon` (período de 20 ns, jitter opcional), executa o reset do sistema, recupera o Sequenciador do `ConfigDB`, instancia e inicia a sequência, aguarda um dreno de 500 ns e cancela a tarefa de clock.

A classe base `BaseI2cTest` define `build_phase` com dois efeitos: configura `DISABLE_COVERAGE_ERRORS = True` no `ConfigDB` e instancia `Env`. Todos os dezoito testes direcionados (TC_01 a TC_18) herdam de `BaseI2cTest`, suprimindo assim a falha de cobertura funcional e permitindo que cada teste valide apenas seu comportamento específico.

O teste TC_06 (`I2cClockJitterLoopbackTest`) é o único que passa `jitter_clock=True` para `_run_test_flow`, ativando a variação aleatória de ±1 ns no semiperiodo do clock durante a execução da sequência `I2cLoopbackSeq`.

O teste TC_12 (`I2cNoFifoStressTest`) é um *placeholder* estrutural: ele reutiliza integralmente a sequência `I2cGoldenWriteSeq` do TC_02, sem nenhuma alteração de parâmetro do DUT ou do estímulo aplicado. A intenção original era representar uma *build* do `i2c_master_axil` com as FIFOs de CMD/WR/RD desabilitadas via parâmetro RTL (cenário previsto pelo IP de Forencich), mas essa variante de *build* não foi implementada nesta versão do ambiente. Do ponto de vista de cobertura, portanto, TC_12 não exercita nenhum caminho do DUT que já não seja coberto pelo TC_02, sendo mantido na suíte apenas como reserva de nome/posição para a futura implementação dessa variante.

O `I2cCoverageTest` (TC_19) é o único teste que herda diretamente de `uvm_test` (não de `BaseI2cTest`), define `DISABLE_COVERAGE_ERRORS = False` e deve ser executado por último. É o único teste em que a `report_phase` do módulo `Coverage` pode reprovar o teste por cobertura incompleta.

3.4 Sequências de Teste e Organização da Suíte

Cada caso de teste é composto de um *test* (classe que herda de `uvm_test`) e uma *sequence* (classe que herda de `uvm_sequence`), conforme o padrão UVM. O test instancia o ambiente, configura parâmetros globais (endereço do escravo, número de iterações) e inicia a sequência no sequenciador do agente ativo. A sequência gera os itens de transação e os envia ao Driver por meio do par `start_item`/`finish_item`.

O arquivo `seq.py` implementa uma sequência dedicada para cada um dos 19 casos de teste. A Tabela 4 descreve os 19 casos de teste da suíte de verificação, organizados por objetivo e funcionalidade exercitada.

Tabela 4 – Suíte de casos de teste implementada.

TC	Nome	Objetivo
TC_01	I2cRegIntegrityTest	Escrita/leitura de padrões no registrador PRESCALE e leitura do STATUS. Valida integridade básica dos registradores.
TC_02	I2cGoldenWriteTest	Escrita I2C simples ao escravo 0x63. Valida o caminho mínimo de uma transação completa.
TC_03	I2cLoopbackTest	Escrita seguida de leitura do mesmo registrador do escravo. Valida loopback de dado.

(continuação)

TC	Nome	Objetivo
TC_04	I2cNackTest	Escrita para endereço inválido (0x11). Verifica detecção e tratamento correto de NACK.
TC_05	I2cRepeatedStartTest	Escrita seguida de leitura sem condição STOP intermediária (START repetido).
TC_06	I2cClockJitterLoopbac	Loopback com jitter de ±1 ns no relógio mestre. Valida robustez temporal.
TC_07	I2cFifoOverflowTest	Enchimento das FIFOs CMD e WR até overflow. Verifica flags de STATUS e limpeza.
TC_08	I2cWriteMultipleTest	Escrita em bloco via comando WRITE_MULTIPLE pela FIFO de comandos.
TC_09	I2cStandaloneStopTest	Comando STOP isolado com contexto de endereço retido.
TC_10	I2cReadFifoOverflowT	Enchimento e drenagem da FIFO de leitura (RD FIFO).
TC_11	I2cMasterFsmTest	Transições ACTIVE_READ/ACTIVE_WRITE na FSM do mestre e STOP standalone.
TC_12	I2cNoFifoStressTest	Placeholder reservado para build com FIFOs desabilitadas via parâmetro RTL; reutiliza a sequência golden (TC_02) sem alteração de DUT ou estímulo nesta versão.
TC_13	I2cStopOnIdleTest	Caminhos *stop_on_idle* em ACTIVE_WRITE e ACTIVE_READ. Valida transições de estado da FSM.
TC_14	I2cReadCommandReg	Leitura AXI do registrador COMMAND (normalmente *write-only*). Verifica comportamento de leitura de registrador especial.
TC_15	I2cBurstReadDataLast	Leituras em rajada e escrita de DATA sem *data_last* (wstrb somente no byte).
TC_16	I2cAddressMismatchW	Mudança de endereço forçando START repetido; FSM WRITE_MULTIPLE.
TC_17	I2cSlaveMultiReadTest	Leituras encadeadas de múltiplos escravos incluindo NACK em escravo inválido.
TC_18	I2cInvalidCmdPrescale	PRESCALE máximo (0xFFFF) e opcodes inválidos ignorados.

TC_19	I2cCoverageTest	Teste de cobertura funcional: exercita todos os bins restantes para atingir 9/9.

A organização da suíte reflete uma estratégia de cobertura incremental: os primeiros testes (TC_01–TC_05) exercitam os caminhos *golden* do DUT; os testes intermediários (TC_06–TC_16) exploram casos de borda e condições de erro; e o TC_19 é dedicado exclusivamente ao fechamento da cobertura funcional, garantindo que todos os 9 bins definidos sejam exercitados ao término da regressão.

3.5 Configuração da Análise de Cobertura de Código

A cobertura de código foi obtida com o **Verilator**, simulador que oferece instrumentação nativa de cobertura (`-coverage`), acionada pelo *Makefile* em `scripts/code_coverage/`. O fluxo de cobertura de código opera em quatro etapas:

1. **Compilação instrumentada**: O Verilator compila o RTL com a opção `--coverage`, inserindo contadores de execução em cada linha e em cada ramo condicional (`if`, `case`) do código Verilog. Isso produz um executável C++ do simulador com instrumentação embutida.

2. **Execução da suíte**: O cocotb aciona o simulador compilado. A cada linha executada e a cada ramo tomado, o contador correspondente é incrementado. Ao final da simulação, os dados de cobertura são gravados em um arquivo `coverage.dat`.

3. **Agregação com lcov**: O utilitário `lcov` lê o arquivo `coverage.dat` de todos os 19 testes. O comando `genhtml` converte esse arquivo em um relatório HTML navegável por arquivo RTL.

4. **Interpretação**: A razão entre linhas/ramos executados e linhas/ramos totais fornece as métricas de cobertura de linhas e de ramos respectivamente.

O *lcov* agrega os dados de execução da suíte completa e gera relatório HTML por arquivo RTL. Os arquivos analisados e seus respectivos tamanhos são: `axis_fifo.v` (100 linhas), `i2c_master_axil.v` (217 linhas), `i2c_master.v` (412 linhas), `i2c_rtl_wrapper.v` (81 linhas) e `i2c_slave.v` (230 linhas), totalizando 1040 linhas e 2897 ramos sob análise.

Limitação importante: A cobertura de código reportada pelo Verilator com *lcov* conta ramos de modo diferente dos simuladores comerciais. O Verilator instancia as bifurcações `if/else` e `case` individualmente, enquanto ferramentas como Synopsys VCS e Cadence Xcelium adicionalmente rastreiam expresões condicionais em atribuições contínuas e *ternary operators*. Isso significa que os percentuais de ramos reportados neste trabalho são específicos à instrumentação do Verilator e não são diretamente comparáveis aos de ambientes comerciais.

3.6 Configuração da Análise de Potência

O *script* `run_power_script.sh` coordena o fluxo de análise de potência em três etapas:

1. **Geração do VCD**: A simulação pós-layout com SDF `/scripts/netlist_simulation_with_SDF_annotation/` gera o arquivo `dump.vcd` com atividade realista da *netlist* anotada.

2. **Processamento do SPEF**: O *script* `refactor.py` transforma o SPEF gerado pelo LibreLane para compatibilidade com o OpenSTA, produzindo `i2c_master_axil_nom_refactored.spef` (1153 linhas impactadas). A principal transformação é a normalização de prefixos de nomes de instância que diferem entre o formato de saída do OpenROAD e o esperado pelo OpenSTA.

3. **Análise OpenSTA**: O OpenSTA[1] 2.6.0 carrega a netlist, o SPEF e o VCD, anota as atividades de comutação (23.108 pinos anotados) e calcula as potências por grupo funcional via o modelo de potência das células Liberty do SKY130.

A anotação de atividade a partir do VCD é o elemento que eleva a precisão da estimativa de potência em relação a métodos baseados em probabilidade estática. O VCD captura a correlação temporal real entre sinais: um sinal que muda apenas em determinadas sequências de dados terá um fator de atividade α muito menor do que um sinal que oscila em cada ciclo, e o OpenSTA utiliza essa distinção ao calcular $P_{din} = \alpha \cdot C_L \cdot V_{DD}^2 \cdot f$ para cada nó da netlist.

3.7 Simulação Pós-Layout com Anotação SDF

A validação em nível de portas reutiliza o mesmo *testbench* Python/UVM, alterando apenas as fontes Verilog e as opções de compilação do simulador (Tabela 5).

[1] https://github.com/parallaxsw/OpenSTA

Tabela 5 – Configuração RTL versus pós-layout no repositório `UVM-I2C-sky130` (Soares, 2026).

Aspecto	**Simulação RTL**	**Simulação pós-layout (GLS)**
DUT	`i2c_rtl_wrapper`	`i2c_netlist_wrapper`
Fontes	RTL em `rtl/`	*Netlist* + células SKY130 + escravo RTL
Simulador	Icarus Verilog	Icarus Verilog
Flags	Padrão cocotb	`-gspecify -ginterconnect -DSDF_DELAYS`
SDF	—	`postlayout/sdf/..._refactored.sdf`
Pré-processamento	—	`refactor.py` (SDF e netlist)
Inicialização	*Reset* RTL	`deposit.py` (nós assíncronos)

A flag `-gspecify` instrui o Icarus Verilog a processar os blocos `specify` presentes nas definições de células SKY130, que carregam os atrasos `IOPATH` extraídos do SDF. A flag `-ginterconnect` habilita a modelagem de atrasos de interconexão (fios de roteamento), que podem contribuir com vários picossegundos em cada caminho crítico no nó SKY130 de 130 nm. A macro `-DSDF_DELAYS` é utilizada pelo *wrapper* para condicionalmente incluir a diretiva `$sdf_annotate` na elaboração.

O *script* `deposit.py` resolve um problema específico da simulação pós-layout: após a síntese e o P&R, os *flip-flops* da netlist podem apresentar estado inicial `X` (indefinido) antes do pulso de reset. Em simulação RTL, o reset síncrono inicializa todos os registradores de forma determinista. Na netlist, células como `sky130_fd_sc_hd__dfxtp_1` não possuem um valor inicial definido na elaboração, o que propaga `X` pela lógica combinacional subsequente e pode causar falhas espúrias no BFM (especialmente no método `_safe_int`, que trata esse estado). O `deposit.py` utiliza o mecanismo `vpi_put_value` para depositar `0` em todos os *flip-flops* identificados por padrão de nome antes de liberar o clock, emulando o efeito de um *power-on reset* funcional.

3.7.1 Advertências SDF e ausência de *timing checks*

O SDF nominal gerado pelo LibreLane para o corner `nom_tt_025C_1v80` (temperatura típica de 25°C, tensão de 1,8 V) contém dois tipos de entradas: atrasos de propagação (`IOPATH`, `INTERCONNECT`) e verificações de temporização (`TIMINGCHECK`). O Icarus Verilog processa os primeiros, mas emite advertências e ignora os últimos. Durante a compilação da netlist com o SDF (`*_refactored.sdf`, com 67.559 linhas), foram geradas **milhares advertências** do tipo `TIMINGCHECK not supported`, cobrindo entradas de *setup*, *hold* e recuperação dentro do arquivo SDF refatorado. Exemplo representativo:

```
SDF WARNING: .../nom_tt_025C_1v80_refactored.sdf:55540: TIMINGCHECK not supported.
SDF WARNING: .../nom_tt_025C_1v80_refactored.sdf:67253: TIMINGCHECK not supported.
SDF WARNING: .../nom_tt_025C_1v80_refactored.sdf:67559: TIMINGCHECK not supported.
```

Listagem 3.8 – Advertências de TIMINGCHECK ignoradas pelo Icarus (trecho).

Os atrasos de propagação são aplicados, o que explica o aumento moderado do tempo simulado em relação ao RTL (+6,6%), porém violações de *setup*/*hold* **não são reportadas**. Isso significa que a GLS atual valida corretude funcional com atrasos de portas, mas **não constitui sign-off temporal**. Para essa finalidade, a integração do simulador CVC ao fluxo cocotb é apontada como a evolução prioritária (Capítulo 6).

Impacto prático: A ausência de *timing checks* pode mascarar violações de *setup* causadas por caminhos críticos que operam próximos ao limite da frequência de síntese (40 MHz no SDC). Na simulação, os sinais transitam com os atrasos corretos, mas nenhum mecanismo verifica se os dados estão estáveis na janela de *setup* do *flip-flop* receptor. Entretanto, como o clock de teste é 50 MHz (20 ns), mais rápido que os 40 MHz (25 ns) do SDC de síntese, qualquer violação de *setup* que existisse na frequência de síntese seria ainda mais provável de ocorrer durante a simulação. O fato de que todos os 19 testes passaram indica que, à frequência de teste, o circuito opera funcionalmente, mas não há garantia formal de margem de *timing*.

4 Resultados e Análise

4.1 Resultados da Suíte de Testes (RTL)

A suíte de 19 casos de teste foi executada no Icarus Verilog 14.0 com cocotb 2.0.1 e Python 3.12.4. O tempo total de simulação foi de 481.329,02 ns de tempo simulado, executado em 3,19 s de tempo real (razão média de 150.802 ns/s). A Tabela 6 sumariza os resultados individuais.

Tabela 6 – Resultados individuais dos casos de teste.

Caso de Teste	**Status**	**Tempo Sim. (ns)**	**Tempo Real (s)**
I2cRegIntegrityTest	**PASS**	1292	0,02
I2cGoldenWriteTest	**PASS**	4732	0,04
I2cLoopbackTest	**PASS**	14732	0,10
I2cNackTest	**PASS**	5050	0,04
I2cRepeatedStartTest	**PASS**	14732	0,10
I2cClockJitterLoopbackTest	**PASS**	14677	0,10
I2cFifoOverflowTest	**PASS**	6492	0,05
I2cWriteMultipleTest	**PASS**	1452	0,01
I2cStandaloneStopTest	**PASS**	892	0,01
I2cReadFifoOverflowTest	**PASS**	6332	0,06
I2cMasterFsmTest	**PASS**	8572	0,09
I2cNoFifoStressTest	**PASS**	4732	0,04
I2cStopOnIdleTest	**PASS**	31172	0,20
I2cReadCommandRegTest	**PASS**	972	0,01
I2cBurstReadDataLastTest	**PASS**	30972	0,19
I2cAddressMismatchWriteMultipleTest	**PASS**	13132	0,10
I2cSlaveMultiReadTest	**PASS**	59330	0,38
I2cInvalidCmdPrescaleMaxTest	**PASS**	31012	0,20
I2cCoverageTest	**PASS**	231052	1,43
Total (PASS=19/FAIL=0)	**PASS**	481.329	3,19

Todos os 19 casos de teste foram aprovados (PASS=19, FAIL=0, SKIP=0), demonstrando a conformidade funcional completa do controlador I2C com a especificação do ambiente de verificação.

Observa-se que o TC_19 (*I2cCoverageTest*) concentra 48% do tempo total de simulação (231.052 ns), pois é responsável por exercitar todos os bins de cobertura funcional, incluindo múltiplas combinações de endereços válidos e inválidos, operações de leitura e escrita e eventos de FIFO.

4.2 Validação Funcional Pós-Layout (GLS + SDF)

A mesma infraestrutura UVM foi reutilizada sobre a *netlist* pós-layout do `i2c_master_axil` com SDF nominal (`nom_tt_025C_1v80`), conforme a Seção 3.7. A regressão completa de 19 casos de teste foi executada com Icarus Verilog, cocotb e o mesmo `testbench` Python; o resumo do cocotb consta na Tabela 7.

Todos os 19 testes passaram (PASS=19, FAIL=0), reproduzindo o mesmo resultado funcional obtido em RTL. O tempo de simulação acumulado foi de 512.867,02 ns (+6,6% em relação aos 481.329 ns em RTL), enquanto o tempo real de execução saltou de 3,19 s para 175,11 s (fator ≈55×), reflexo do custo computacional da GLS com milhares de instâncias de células SKY130. O TC_01 consumiu 70,72 s de tempo real por incluir a elaboração inicial e a anotação SDF; os demais TCs permaneceram na faixa de 0,2–11,5 s cada. O TC_19 (*I2cCoverageTest*) manteve 48% do tempo simulado total (246.572 ns) e fechou a cobertura funcional em **9/9 bins**, como em RTL.

Tabela 7 – Resultados da simulação pós-layout (GLS + SDF, Icarus Verilog).

Caso de Teste	Status	Tempo Sim. (ns)	Tempo Real (s)
I2cRegIntegrityTest	**PASS**	1372,00	70,72
I2cGoldenWriteTest	**PASS**	4752,00	1,01
I2cLoopbackTest	**PASS**	14892,00	4,07
I2cNackTest	**PASS**	5070,00	1,23
I2cRepeatedStartTest	**PASS**	14892,00	3,15
I2cClockJitterLoopbackTest	**PASS**	14935,00	3,15
I2cFifoOverflowTest	**PASS**	6512,00	1,20
I2cWriteMultipleTest	**PASS**	1472,00	0,37
I2cStandaloneStopTest	**PASS**	912,00	0,20
I2cReadFifoOverflowTest	**PASS**	6992,00	1,73
I2cMasterFsmTest	**PASS**	12752,00	3,07
I2cNoFifoStressTest	**PASS**	4752,00	1,47
I2cStopOnIdleTest	**PASS**	39492,00	7,81
I2cReadCommandRegTest	**PASS**	1012,00	0,21
I2cBurstReadDataLastTest	**PASS**	31752,00	6,38
I2cAddressMismatchWriteMultipleTest	**PASS**	13172,00	2,62
I2cSlaveMultiReadTest	**PASS**	60430,00	11,48
I2cInvalidCmdPrescaleMaxTest	**PASS**	31132,00	5,42
I2cCoverageTest	**PASS**	246572,00	49,78
Total (PASS=19/FAIL=0)	**PASS**	512.867,02	175,11

Tabela 8 – Comparação resumida RTL versus pós-layout (mesma suíte de 19 TCs).

Métrica	**RTL (Icarus)**	**Pós-Layout (GLS + SDF)**
Testes aprovados	19/19	19/19
Tempo simulado total	481.329 ns	512.867 ns
Tempo real total	3,19 s	175,11 s
Razão média (ns/s)	150.802	2.929
Cobertura funcional (TC_19)	9/9 bins	9/9 bins
Timing checks SDF	N/A	Não suportados (Icarus)

Após a regressão, o `dump.vcd` gerado na pasta de simulação pós-layout foi consumido pelo fluxo de potência (`scripts/power_estimation/`), produzindo a mesma estimativa de 3,21 mW (Tabela 11, OpenSTA com 23.108 pinos anotados), validando a cadeia VCD→SPEF→OpenSTA sobre atividade extraída da *netlist*.

4.2.1 Log representativo (TC_19 pós-layout)

O encerramento da suíte confirma cobertura funcional integral sobre a *netlist*:

```
512367.02ns INFO [coverage]:   Coverage: 9/9 bins hit.
512367.02ns INFO [coverage]:   Functional coverage: all bins covered.
512367.02ns INFO cocotb: I2cCoverageTest passed
512367.02ns INFO cocotb.regression: TESTS=19 PASS=19 FAIL=0 SKIP=0
```

Listagem 4.1 – Encerramento do TC_19 em GLS — cobertura 9/9.

4.3 Análise dos Logs de Simulação (RTL)

Os logs produzidos pelo Scoreboard seguem um formato padronizado, com emojis de status que facilitam a inspeção visual durante o debug:

```
  312.00ns INFO [scoreboard]: [✅] REG_WRITE [0x0C] = 0x000000FF
  392.00ns INFO [scoreboard]: [✅] REG_READ [0x0C] PASS exp=0xFF got=0xFF
  472.00ns INFO [scoreboard]: [✅] REG_WRITE [0x0C] = 0x0000AABB
  552.00ns INFO [scoreboard]: [✅] REG_READ [0x0C] PASS exp=0xAABB got=0xAABB
  792.00ns INFO [scoreboard]: [📊] STATUS read = 0x00004900 (volatile, no check)
 1292.00ns INFO [scoreboard]: [✅] Scoreboard: all checks passed (3 verifications).
 1292.00ns INFO [coverage]:   Coverage: 3/9 bins hit.
```

Listagem 4.2 – Trecho representativo do log do TC_01 (PRESCALE write/read).

A mensagem `volatile, no check` para leituras de STATUS reflete uma decisão de projeto deliberada: o registrador STATUS contém campos que mudam assincronamente (flags de FIFO, *busy*), tornando sua verificação determinista inviável sem um modelo de referência temporal completo. A estratégia adotada foi registrá-lo para auditoria sem impor verificações estritas.

Para operações I2C, o Scoreboard verifica os dados observados pelo Monitor contra o espelho interno de memória do escravo. O trecho abaixo exemplifica o TC_03 (*loopback*):

```
10256.00ns INFO [scoreboard]: [✔] I2C_WRITE slave=0x63 reg=0x05 data=0x42
20256.00ns INFO [scoreboard]: [✔] I2C_READ  slave=0x63 PASS exp=0x42 got=0x42
20756.00ns INFO [scoreboard]: [✔] Scoreboard: all checks passed (2 verifications).
20756.00ns INFO [coverage]:   Coverage: 2/9 bins hit.
```

Listagem 4.3 – Log do TC_03 (loopback I2C write/read).

A condição de NACK é tratada explicitamente pelo Scoreboard com aviso diferenciado (TC_04):

```
25306.00ns INFO [scoreboard]: [⚠] Expected NACK on I2C_WRITE slave=0x11
25806.00ns INFO [scoreboard]: [✔] Scoreboard: all checks passed (0 verifications).
```

Listagem 4.4 – Log do TC_04 (NACK em escravo inválido).

4.4 Resultados de Cobertura Funcional

A Tabela 9 acompanha a evolução da cobertura funcional ao longo da suíte, evidenciando que a progressão foi intencional: os testes iniciais cobrem um subconjunto dos bins, e o TC_19 finaliza com 100% de cobertura.

Tabela 9 – Evolução da cobertura funcional ao longo da suíte.

TC	Bins Cobertos	Cobertura	Bins Ausentes
TC_01	3/9	33%	FIFO_OVF, I2C_WRITE (ACK), I2C_READ (ACK/-NACK), I2C_WRITE (NACK), WRITE_MULT
TC_02	1/9	11%	REG_READ (STATUS/-PRESCALE), REG_WRITE (PRESCALE), FIFO_OVF, I2C_READ, WRITE_MULT
TC_03	2/9	22%	REG_READ (STATUS/-PRESCALE), REG_WRITE (PRESCALE), FIFO_OVF, WRITE_MULT
TC_13	4/9	44%	REG_READ (PRESCALE), FIFO_OVF, I2C_WRITE (NACK), WRITE_MULT
TC_18	5/9	56%	Todos os registradores cobertos; faltam FIFO_OVF, I2C_WRITE (NACK), WRITE_MULT, I2C_READ (NACK)
TC_19	**9/9**	**100%**	**Nenhum**

A cobertura funcional completa (9/9 bins) ao final do TC_19 confirma que todas as funcionalidades especificadas foram exercitadas: operações de leitura e escrita em ambos os

registradores acessíveis via PRESCALE, operações I2C com e sem NACK, evento de overflow de FIFO e o comando WRITE_MULTIPLE.

4.5 Resultados de Cobertura de Código

A Tabela 10 apresenta os resultados de cobertura de código por arquivo.

Tabela 10 – Resultados de cobertura de código (linhas e ramos) por arquivo RTL.

Arquivo	**Linhas**	**Hits**	**Cob. L.**	**Ramos**	**Hits**	**Cob. R.**
`axis_fifo.v`	100	72	72,0%	812	462	56,9%
`i2c_master_axil.v`	217	208	95,9%	746	495	66,4%
`i2c_master.v`	412	367	89,1%	609	456	74,9%
`i2c_rtl_wrapper.v`	81	76	93,8%	317	204	64,4%
`i2c_slave.v`	230	219	95,2%	413	325	78,7%
Total	1040	942	**90,6%**	2897	1942	**67,0%**

Os resultados revelam um **desempenho global de 90,6% de cobertura de linhas e 67,0% de cobertura de ramos**. As análises por arquivo indicam:

- **`i2c_master_axil.v`** (95,9% linhas): interface AXI-Lite amplamente exercitada. Os ramos não cobertos (33,6%) correspondem principalmente a condições de erro/parametriza
  ção que requerem estímulos de temporização específicos, como timeouts de barramento.
- **`i2c_slave.v`** (95,2% linhas): o escravo respondeu adequadamente à maioria dos cenários; os ramos não cobertos (21,3%) incluem condições de clock stretching e respostas a endereços de broadcast não exercitadas.
- **`i2c_master.v`** (89,1% linhas): a FSM principal do protocolo I2C foi amplamente coberta; os ramos ausentes (25,1%) correspondem a transições de estado em condições de arbitration loss (não suportado pelo wrapper atual).
- **`axis_fifo.v`** (72,0% linhas): o menor índice de cobertura de linhas, pois as FIFOs possuem modos de operação alternativos (parametrizações de largura e profundidade) não totalmente exercitados pela configuração atual.
- **`i2c_rtl_wrapper.v`** (93,8% linhas): o wrapper é relativamente simples, com poucas condições de desvio; os ramos ausentes (35,6%) indicam lógica de multiplexação não ativada.

A diferença entre cobertura de linhas (90,6%) e de ramos (67,0%) é esperada e reflete a assimetria natural: muitas linhas são executadas em apenas um dos possíveis caminhos condicionais.

4.6 Resultados de Análise de Potência

A Tabela 11 apresenta os resultados da estimativa de potência obtida pelo OpenSTA 2.6.0, com anotação de 23.108 atividades de pinos a partir do VCD gerado pela simulação.

Tabela 11 – Estimativa de potência por grupo funcional (OpenSTA 2.6.0).

Grupo	**Interna (W)**	**Chaveamento (W)**	**Fuga (W)**	**Total (W)**	**%**
Sequencial	1,91 e-03	2,49 e-06	9,80 e-09	1,91 e-03	59,6%
Combinacional	5,77 e-05	3,44 e-05	2,53 e-08	9,21 e-05	2,8%
Clock	6,60 e-04	5,46 e-04	2,41 e-09	1,21 e-03	37,6%
Total	**2,63 e-03**	**5,83 e-04**	**3,75 e-08**	**3,21 e-03**	**100%**

Componentes: 81,8% potência interna; 18,2% chaveamento; <0,1% fuga.

Os resultados revelam um perfil de potência típico de controladores seriais de baixa frequência no nó SKY130:

- **Potência total de 3,21 mW**, valor compatível com a classe de IPs de periférico I2C em nós de 130 nm a frequências de dezenas de MHz.
- **Elementos sequenciais dominam com 59,6%**: isso é esperado em controladores com FIFOs profundas e múltiplos registradores de estado; a potência interna dos *flip-flops* (1,91 mW) é a maior fonte individual.
- **Clock representa 37,6%**: a rede de distribuição de clock é tipicamente uma fonte significativa de potência dinâmica em designs síncronos, pois a árvore de clock comuta com fator de atividade $\alpha = 0,5$ em cada ciclo.
- **Lógica combinacional contribui apenas 2,8%**: o controlador é predominantemente sequencial (FSMs + FIFOs), com pouca lógica combinacional pura.
- **Correntes de fuga negligenciáveis (<0,1%)**: no nó de 130 nm, as correntes sub-limiares ainda são muito menores que a potência dinâmica, diferente do que ocorre em nós abaixo de 28 nm, onde a fuga pode representar 20–30% da potência total.

4.6.1 Advertências do OpenSTA

O log de execução do OpenSTA registrou advertências relevantes:

```
Warning: module sky130_fd_sc_hd__tapvpwrvgnd_1 not found. Creating black box.
Warning: module sky130_ef_sc_hd__decap_40_12 not found. Creating black box.
Warning: clock clk vcd period 19.998 differs from SDC clock period 25.000
Warning: i2c_master_axil_nom_refactored.spef line 2861, syntax error.
Annotated 23108 pin activities.
```

Listagem 4.5 – Advertências do OpenSTA durante a análise de potência.

As células de preenchimento (*filler* e *decap*) e as células de conexão à rede de alimentação (*tap cells*) foram tratadas como caixas-pretas, o que não impacta a estimativa de potência lógica. A divergência de período de clock (19,998 ns no VCD contra 25,0 ns no SDC) indica que o VCD foi gerado com clock de 50 MHz (20 ns), enquanto o SDC de síntese foi configurado para 40 MHz (25 ns); isso resulta em estimativas levemente conservadoras. O erro de sintaxe no SPEF na linha 2861 foi corrigido pelo *script* de refatoração.

5 Comparação: Python-UVM vs SystemVerilog-UVM

Este capítulo contrasta a abordagem implementada (cocotb + pyuvm) com um *testbench* de referência em SystemVerilog sem biblioteca UVM formal, mas com estrutura análoga, BFM, Scoreboard, Agente e Sequenciador implementados em SV puro. A Seção 5.1 descreve o estado da implementação SV; a Seção 5.2 apresenta métricas quantitativas; a Seção 5.3 articula a análise qualitativa com base nos dados coletados.

5.1 Implementação SV de Referência

Foi desenvolvido um *testbench* em SystemVerilog para o mesmo DUT (*i2c_rtl_wrapper*), disponível em `uvm/SV/tb_i2c_uvm.sv` no repositório `UVM-I2C-sky130` (Soares, 2026). O simulador alvo é o Verilator, escolha que determina diretamente as capacidades e limitações da implementação SV.

5.1.1 Estrutura do Testbench SV

O testbench SV (`tb_i2c_uvm.sv`, 728 LOC) implementa os seguintes componentes em SV puro, sem a biblioteca UVM formal `uvm_pkg`:

- **Pacote de constantes** (`i2c_pkg`): define o mapa de registradores, as máscaras de *status*, os *opcodes* de comando e o endereço do escravo válido como `localparam`, centralizando as constantes compartilhadas entre componentes.
- **Interface** (`i2c_axil_if`): agrupa os sinais AXI-Lite em um bloco tipado, permitindo que o BFM e o módulo *top* os compartilhem via referência virtual (`virtual interface`).
- **Item de Sequência** (`i2c_seq_item`): classe SV que espelha a definição Python, com campos `cmd_type`, `slave_addr`, `reg_addr`, `payload`, `read_data` e `nack_seen`.
- **BFM** (`axil_bfm`): classe SV com tarefas (`task`) que implementam os *handshakes* AXI-Lite e as operações I2C compostas, incluindo verificação de PRESCALE e *timeout* de 50.000 ciclos.
- **Scoreboard** (`i2c_scoreboard`): classe SV que mantém o espelho de registradores e de memória do escravo, com a mesma lógica de verificação do ambiente Python.
- **Módulo top** (`tb_top`): instancia o DUT, a interface e os componentes; implementa o laço de testes com `$display` como separadores de TC.

5.1.2 Limitações Impostas pelo Verilator

O Verilator é um compilador de alta velocidade que traduz RTL Verilog/SystemVerilog para C++. Essa arquitetura impõe restrições à norma IEEE 1800 que impactam diretamente a capacidade de se implementar UVM formal em SV sobre essa ferramenta:

- **Ausência de `covergroup`/`coverpoint` nativos**: O Verilator não suporta a construção `covergroup` da norma IEEE 1800. Isso torna impossível definir cobertura funcional com sintaxe SV padrão. A alternativa é implementar contadores manuais em código SV ou instrumentar o C++ gerado, o que requer conhecimento da arquitetura interna do Verilator.
- **Restrições em `fork..join_any/join_none` dentro de classes**: O Verilator 5.036, versão utilizada neste trabalho, possui suporte experimental a flag `-timing`. Sem esse flag, declarações de paralelismo UVM (`fork..join_none`) em fases como `run_phase` simplesmente não compilam. O testbench SV desenvolvido contorna essa limitação executando os testes sequencialmente em um único *thread* de simulação.
- **Randomização limitada**: A construção `rand`/`randc` com `constraint` não é suportada pelo Verilator. A randomização no testbench SV utiliza `$urandom_range`, sem qualquer solucionador de restrições. Isso inviabiliza o uso de CRV (*Constrained Random Verification*) nativo, uma das vantagens mais significativas do SV comercial.
- **Sem SVA (*SystemVerilog Assertions*) complexas**: O suporte a *sequences* e *properties* SVA no Verilator é limitado. Verificações temporais formais, como "toda transação I2C deve completar em no máximo 500 ciclos após o START", não são implementáveis com sintaxe SVA padrão.
- **Sem *timing checks* SDF**: O Verilator não suporta a anotação SDF com *timing checks* na simulação de netlist.

A consequência prática dessas restrições é que o testbench SV sobre Verilator **não implementa UVM formal**: não há hierarquia de fases, não há `uvm_factory`, não há análise de portas conectadas dinamicamente. O ambiente SV resultante é funcional e cobre os 19 TCs, mas é essencialmente um testbench dirigido, estruturalmente inspirado no UVM, sem os mecanismos de reutilização e configurabilidade que definem a metodologia.

5.2 Resultados Quantitativos

5.2.1 Taxa de aprovação e tempo de execução

O testbench SV executou a mesma suíte de 19 TCs sobre o mesmo DUT (*i2c_rtl_wrapper*), no Verilator 5.036 com a flag `-timing`. A Tabela 12 apresenta os resultados individuais.

Tabela 12 – Resultados da suíte SV (Verilator 5.036 com `-timing`).

Caso de Teste (SV)	**Status**	**Tempo Sim. (ns)**	**Tempo Real (s)**
I2cRegIntegrityTest	**PASS**	711	—
I2cGoldenWriteTest	**PASS**	109.610	—
I2cLoopbackTest	**PASS**	218.490	—
I2cNackTest	**PASS**	443.710	—
I2cRepeatedStartTest	**PASS**	552.590	—
I2cClockJitterLoopbackTest	**PASS**	773.690	—
I2cFifoOverflowTest	**PASS**	891.091	—
I2cWriteMultipleTest	**PASS**	892.631	—
I2cStandaloneStopTest	**PASS**	893.691	—
I2cReadFifoOverflowTest	**PASS**	896.911	—
I2cMasterFsmTest	**PASS**	899.831	—
I2cNoFifoStressTest	**PASS**	900.131	—
I2cStopOnIdleTest	**PASS**	1.903.131	—
I2cReadCommandRegTest	**PASS**	1.904.131	—
I2cBurstReadDataLastTest	**PASS**	2.904.131	—
I2cAddressMismatchWriteMultipleTest	**PASS**	3.904.131	—
I2cSlaveMultiReadTest	**PASS**	4.904.131	—
I2cInvalidCmdPrescaleMaxTest	**PASS**	5.904.131	—
I2cCoverageTest	**PASS**	6.904.131	—
Total (PASS=19/FAIL=0)	**PASS**	~6,9 M ns	2,26

Observação sobre o tempo simulado SV: O Verilator executa a simulação como um único processo sequencial, sem reiniciar o tempo de simulação entre testes. Isso resulta em tempos acumulados crescentes para cada TC. A métrica relevante é o **tempo real total de 2,26 s**, que deve ser comparado aos 3,19 s do ambiente Python (Icarus). O Verilator é mais rápido para RTL puro porque compila o modelo para C++ nativo, eliminando o *overhead* de interpretação de bytecode VPI. Já o cocotb/Icarus possui um *overhead* adicional de comunicação Python↔VPI a cada evento de simulação.

5.2.2 Cobertura funcional

O testbench SV não implementa *covergroups*, mas o Scoreboard SV adota a mesma lógica de bins que o ambiente Python: contadores manuais de ocorrências de cada tipo de transação. Ao término da suíte, todos os 9 bins equivalentes foram exercitados, confirmando a equivalência funcional dos dois ambientes.

5.2.3 Esforço de implementação e linhas de código

A Tabela 13 consolida as métricas de tamanho dos dois ambientes.

Tabela 13 – Métricas de tamanho — Python-UVM vs SV.

Componente	**Python-UVM (LOC)**	**SV (LOC)**
BFM / driver de barramento	255	202
Sequências (seq.py)	519	150
Tests (test_i2c.py)	131	96
Scoreboard	82	185
Cobertura	32	95
Monitor + Agente + Env	59	(em outros componentes)
Total	**1.078**	**728**

O ambiente Python tem 48% mais linhas de código que o SV, reflexo da estrutura de arquivos separados e da verbosidade das sequências individuais por TC. Contudo, essa comparação não capta a qualidade arquitetural: o ambiente Python segue a hierarquia UVM formal (com fases, factory e analysis ports reais), enquanto o SV é um testbench procedural estruturado. A modularidade Python permite, por exemplo, adicionar um novo TC apenas com um novo arquivo `seq.py` sem modificar nenhum outro componente.

5.3 Análise Qualitativa

5.3.1 Produtividade e Ciclo de Desenvolvimento

O impacto mais tangível observado durante o desenvolvimento foi o **ciclo de iteração**. No ambiente Python com cocotb/Icarus, a sequência típica de desenvolvimento é:

1. Editar arquivo Python (sem compilação);
2. Executar `make` (compile Icarus + run: $\approx$ 3 s total);
3. Observar o log com *formatação colorida* e *emojis* de status;
4. Corrigir e repetir.

No ambiente SV com Verilator, o ciclo inclui uma etapa adicional de compilação C++:

1. Editar arquivo SV;
2. `verilator -binary --timing ...tb_i2c_uvm.sv`: $\approx$ 18 s (primeira compilação) ou $\approx$ 2 s (recompilação incremental);
3. Executar o binário gerado;
4. Observar o log com `$display`.

A compilação inicial do Verilator ($\approx$ 18 s) é significativamente mais longa que a compilação do Icarus (<1 s). Em sessões de desenvolvimento intensivo com muitas iterações, essa diferença se acumula. A recompilação incremental do Verilator ($\approx$ 2 s) atenua o problema, mas o ambiente Python permanece mais ágil para iterações rápidas.

5.3.2 Depuração (*Debugging*)

A depuração de erros é substancialmente mais rica no ambiente Python. O Scoreboard emite mensagens formatadas com identificação precisa de tempo de simulação, valores esperados e observados, e tipo de operação. Qualquer biblioteca Python (como `ipdb` ou `pdb`) pode ser inserida em qualquer ponto do fluxo de verificação para inspecionar o estado interno dos componentes. O ambiente SV depende exclusivamente de `$display` e `$dumpvars`, sem a possibilidade de *breakpoints* interativos.

5.3.3 Randomização com Restrições (CRV)

O SystemVerilog possui suporte nativo e maduro a CRV por meio de solucionadores de satisfação de restrições (CSP), permitindo a geração eficiente de estímulos aleatórios que respeitam invariantes complexas. Sobre o Verilator, no entanto, essa funcionalidade não está disponível: a construção `rand/randc` com `constraint` requer um *solver* externo que não é incluído no Verilator *open-source*. O ambiente SV desenvolvido recorre a `$urandom_range`, que é equivalente ao `random.randint()` Python.

Em simuladores comerciais (Synopsys VCS, Cadence Xcelium), o CRV nativo do SV é muito mais expressivo que qualquer biblioteca Python atual. Para projetos com espaços de estado muito grandes, como controladores com múltiplos canais ou caches, essa vantagem pode ser decisiva. Para o DUT deste trabalho (controlador I2C com 4 registradores e FIFOs de 16 entradas), a ausência de CRV não foi um limitante prático.

5.3.4 Cobertura de Código com Ferramentas Open-Source

A cobertura de código em ambos os ambientes foi obtida com o Verilator (`--coverage`), pois o Icarus Verilog não suporta instrumentação nativa de cobertura. Isso significa que a métrica de cobertura de código é **independente da linguagem do testbench**: o mesmo RTL instrumentado pelo Verilator produz os mesmos relatórios, independentemente de o estímulo ter sido gerado por Python ou SV. A diferença observada entre as duas execuções do Verilator (com testbench Python vs SV) foi inferior a 1 ponto percentual, confirmando essa independência.

5.3.5 Integração com Ferramentas de Análise

O ecossistema Python apresenta vantagem significativa na integração com ferramentas de análise pós-simulação. O processamento estatístico dos logs de cobertura e a geração de

gráficos de progressão de cobertura foram realizados diretamente em Python, sem ferramentas intermediárias. Em ambos ambientes, a análise do VCD exigiu ferramentas externas (*e.g.*, GTKWave).

5.3.6 Simulação Pós-Layout (GLS)

A diferença mais relevante entre os dois ambientes emerge na simulação pós-layout. O testbench SV sobre Verilator **não suporta** simulação de netlist com anotação SDF: o Verilator foi projetado para RTL, e a simulação de netlists com células de biblioteca específicas de PDK (como as SKY130) não faz parte do seu escopo. O ambiente Python sobre Icarus, por outro lado, suportou a GLS completa com anotação SDF nominal, com os 19 testes aprovados em 175 s (netlist) contra 3,19 s (RTL). Essa capacidade é central para o objetivo deste trabalho e representa a maior vantagem prática do fluxo Python/Icarus sobre o SV/Verilator no contexto *open-source*.

5.3.7 Tabela Comparativa Resumida

A Tabela 14 consolida as métricas obtidas nos dois ambientes.

Tabela 14 – Comparação Python-UVM (cocotb/pyuvm) vs SV (Verilator).

Critério	**Python-UVM (cocotb/Icarus)**	**SV (Verilator)**
Metodologia UVM formal	Sim (pyuvm: fases, factory, ap)	Não (SV procedural estruturado)
Suíte (19 TCs)	PASS 19/19	PASS 19/19
Tempo real RTL	3,19 s	2,26 s
Tempo compilação	<1 s (Icarus)	18 s (1ª) / 2 s (incr.)
Cobertura de linhas	90,6% (Verilator)	90,6% (mesmo Verilator)
Cobertura funcional	9/9 bins	9/9 bins (manual)
Cobertura nativa SV	N/A	Não (covergroup não suportado)
CRV nativo	Não (random Python)	Não (urandom no Verilator)
GLS + SDF (19 TCs)	PASS 19/19 (175 s)	**Não suportado**
Timing checks SDF	Não (Icarus)	Não (Verilator)
Análise de potência	Sim (VCD → OpenSTA)	**Não (sem VCD da netlist)**
Depuração interativa	Sim (pdb, ipdb)	Não ($display apenas)
Custo de licença	Zero	Zero
Integração Python/dados	Nativa	Via ferramentas externas

6 Conclusão

Este trabalho apresentou a implementação e avaliação de um ambiente completo de verificação funcional UVM em Python (*cocotb* + *pyuvm*) aplicado ao controlador I2C *i2c_master_axil*, utilizando o PDK SKY130 como nó tecnológico de referência. O ambiente foi construído sobre ferramentas de código aberto ou gratuitas, desde a simulação RTL (Icarus Verilog) até o *place and route* (LibreLane) e a estimativa de potência (OpenSTA). A extensão futura com o simulador CVC, necessária para habilitar *timing checks* na GLS, implica a adoção de uma licença *Modified Artistic License* que permite uso pessoal e acadêmico gratuito, mas **não** é permissiva para uso comercial sem autorização da Tachyon Design Automation.

Os resultados obtidos demonstram de forma conclusiva a **viabilidade e eficiência do fluxo proposto**:

1. **Verificação funcional completa**: todos os 19 casos de teste foram aprovados (PASS=19, FAIL=0), cobrindo funcionalidades críticas do protocolo I2C, incluindo escrita simples, loopback, NACK, START repetido, overflow de FIFO, WRITE_MULTIPLE e variações de PRESCALE.

2. **Alta cobertura de código**: 90,6% de cobertura de linhas e 67,0% de cobertura de ramos do RTL (Verilator/lcov), demonstrando que o ambiente exercitou adequadamente os caminhos relevantes da implementação.

3. **Cobertura funcional completa**: todos os 9 bins definidos foram cobertos ao término da suíte, confirmando que as funcionalidades especificadas foram sistematicamente verificadas.

4. **Análise de potência integrada**: a extração de VCD durante a simulação pós-layout e sua utilização pelo OpenSTA resultou em uma estimativa de potência total de 3,21 mW, compatível com o perfil esperado para controladores I2C em 130 nm, com potência sequencial dominante (59,6%) e correntes de fuga negligenciáveis (<0,1%).

5. **Fluxo majoritariamente livre**: desde o simulador (Icarus Verilog) até o P&R (LibreLane), passando por GLS com SDF e análise de potência (OpenSTA), sem licenciamento proprietário. A extensão prevista com o CVC introduz uma licença *Modified Artistic License* com restrições para uso comercial, o que deve ser avaliado conforme o contexto de aplicação.

6. **Reutilização do testbench em GLS**: o mesmo ambiente Python/UVM validou funcionalmente a *netlist* pós-layout com SDF, 19/19 TCs aprovados e 9/9 bins de cobertura funcional, evidenciando portabilidade do fluxo entre RTL e nível de portas.

7. **Cadeia VCD→potência em pós-layout**: o `dump.vcd` da GLS alimentou o OpenSTA com 23.108 atividades anotadas, confirmando o fluxo de estimativa de potência sobre a *netlist* real.

8. **Comparação com SV**: o testbench equivalente em SystemVerilog sobre Verilator demonstrou limitações estruturais que inviabilizam UVM formal, cobertura nativa de *covergroups* e simulação pós-layout, evidenciando as vantagens práticas do fluxo Python/Icarus no contexto *open-source*.

6.1 Limitações

As principais limitações identificadas neste trabalho são:

- ***Timing checks* na GLS (Icarus)**: embora o SDF seja anotado e os atrasos de portas simulados, o Icarus Verilog não processa entradas `TIMINGCHECK` do SDF. As milhares de advertências do tipo `TIMINGCHECK not supported` emitidas durante a compilação da netlist indicam que violações de *setup*/*hold* não são reportadas na simulação. Isso reduz o valor da GLS como etapa final de *sign-off* temporal. O Icarus aplica os atrasos de propagação (`IOPATH` e `INTERCONNECT`), o que explica o aumento de +6,6% no tempo simulado (RTL: 481 kns; GLS: 513 kns), mas a margem de *setup* não é verificada.

- **Desempenho da GLS**: a regressão pós-layout levou 175 s frente a 3,19 s em RTL (fator $\approx 55\times$), limitando iterações rápidas. A inicialização do TC_01 concentrou 39,7 s desse total (elaboração do netlist e anotação SDF inicial).

- **Randomização com restrições**: a ausência de solucionador CRV nativo em Python limita a expressividade dos estímulos aleatórios para projetos com invariantes complexas. Integração com Z3 ou `python-constraint` é o caminho natural para preencher essa lacuna.

- **Cobertura de ramos (33% não cobertos)**: condições de erro como *arbitration loss*, *clock stretching* e variantes de parametrização das FIFOs exigem estímulos adicionais ou injeção de falhas que não foram implementados neste trabalho.

- **Erro de sintaxe no SPEF (linha 2861)**: o *script* de refatoração corrigiu o erro, mas a causa raiz (divergência de formato entre a saída do OpenROAD e o esperado pelo OpenSTA) deve ser documentada como ponto de atenção para usuários do LibreLane com outros designs.

- **TC_12 não exercita uma configuração distinta do DUT**: conforme descrito na Seção 3.3.11, o teste reutiliza integralmente a sequência do TC_02 e não corresponde a uma *build* real com FIFOs desabilitadas, devendo ser tratado como reserva estrutural para trabalho futuro, e não como caso de teste funcionalmente distinto.

6.2 Trabalhos Futuros

6.2.1 Integração do Simulador CVC para *Timing Checks* em GLS

A limitação mais crítica identificada, a ausência de *timing checks* SDF na GLS, tem solução conhecida no ecossistema *open-source*: o simulador **CVC** (*CVC 64 Version 7.00b*), desenvolvido pela Tachyon Design Automation e disponível em *cambridgehackers/open-src-cvc* (Tachyon Design Automation, 2023). O CVC é licenciado sob a *OSS CVC Dual Licensing Modified Artistic License*: o uso é gratuito para projetos pessoais e acadêmicos, porém **requer autorização comercial** da Tachyon Design Automation para uso em produtos ou serviços comerciais. Portanto, a integração do CVC não mantém o fluxo inteiramente permissivo em contextos corporativos.

O CVC é um simulador Verilog-2001 com suporte completo a SDF, incluindo as entradas `TIMINGCHECK` (*setup*, *hold*, *recovery*, *removal*). Diferentemente do Icarus Verilog, o CVC processa todas as seções `TIMINGCHECK` do arquivo SDF e emite mensagens de violação durante a simulação, na forma:

```
** TIMING VIOLATION: HOLD time: required 0.12ns actual 0.08ns
   FLIP-FLOP: \i2c_master_inst/state_reg[2]
   CLOCK: clk rising at 10240.000ns
   DATA:  \i2c_master_inst/next_state[2] changed at 10240.040ns
```

Listagem 6.1 – Exemplo de saída de *timing check* do CVC (formato esperado).

Para integrar o CVC ao fluxo cocotb existente, a modificação necessária é mínima: o CVC expõe uma interface VPI (*Verilog Procedural Interface*) compatível, o que permite que a biblioteca `libcocotbvpi_icarus.so` seja substituída pela versão compilada para CVC (`libcocotbvpi_cvc.so`), sem qualquer alteração nos componentes Python do ambiente de verificação. O fluxo proposto é:

1. Compilar o CVC com suporte VPI:
   `make -C cvc_src VPI=1 SDF=1`
2. Compilar a biblioteca VPI do cocotb para CVC:
   `make COCOTB_PY_MODULE=components.test_i2c`
   `SIM=cvc TOPLEVEL=i2c_netlist_wrapper`
3. Executar com o mesmo Makefile cocotb, alterando apenas `SIM=cvc`:
   `export SIM=cvc; make`

O testbench Python permanece inalterado. Os 19 casos de teste funcionariam identicamente, mas agora com violações de *setup* e *hold* sendo reportadas como mensagens de aviso ou como falhas no log de simulação. O Scoreboard Python poderia ser estendido para analisar essas mensagens e contabilizá-las como falhas de verificação temporal.

Essa integração transformaria a GLS atual de uma validação "funcional com atrasos" em uma validação "funcional com *sign-off* temporal", elevando a qualidade do fluxo *open-source* a um patamar comparável ao de ambientes comerciais para esse critério específico.

6.2.2 Expansão da Cobertura de Ramos

Os 33% de ramos não cobertos incluem caminhos que requerem estímulos especializados: *arbitration loss* (dois mestres no barramento), *clock stretching* (escravo puxando SCL para baixo para reduzir velocidade) e configurações alternativas das FIFOs (profundidade e largura parametrizáveis no *axis_fifo.v*). Um plano de fechamento de cobertura poderia incluir:

- TC_20: múltiplos mestres com arbitragem (requer modificação do wrapper);
- TC_21: escravo com *clock stretching* habilitado;
- TC_22–TC_24: FIFO com profundidades 4, 32 e 128 (parâmetros do DUT).

6.2.3 Randomização com Restrições via Z3

A biblioteca Z3 (*SMT solver* da Microsoft Research, distribuída sob licença MIT) expõe uma API Python que permite definir restrições sobre variáveis e obter soluções satisfatórias. Isso possibilitaria, por exemplo, gerar endereços de escravo aleatórios que satisfaçam "endereço válido $\equiv$ 0x63 $\vee$ endereço inválido $\neq$ 0x63" com probabilidade controlada, aproximando-se do CRV nativo do SV sem depender de ferramentas proprietárias.

6.2.4 Integração com *Formal Property Verification*

O ambiente Python poderia ser estendido com asserções formais utilizando o SymbiYosys (*open-source formal verification frontend*), que aceita SVA sintetizável compilada via Yosys. Propriedades como "após STOP, o bit *busy* deve desassertir em no máximo 4 ciclos" poderiam ser verificadas formalmente sobre o RTL, complementando a verificação por simulação.

6.2.5 Implementação efetiva da *build* sem FIFOs (TC_12)

O objetivo é parametrizar o RTL do `i2c_master_axil` (ou o *wrapper*) para permitir a desabilitação das FIFOs de CMD/WR/RD em tempo de elaboração, e implementar uma sequência distinta que valide esse modo simplificado, hoje apenas reservado como *placeholder*.

6.3 Considerações Finais

Em síntese, este trabalho contribui para a comunidade de *design* de hardware aberto ao demonstrar que ferramentas Python e *open-source* são capazes de orquestrar verificações

funcionais completas, incluindo análise de potência, em nós tecnológicos reais. O fluxo implementado, cocotb/pyuvm sobre Icarus Verilog, com cobertura via Verilator/lcov, GLS com SDF e estimativa de potência com OpenSTA, constitui uma alternativa concreta e documentada às soluções proprietárias para projetos de pequena e média complexidade. Ressalva-se que a extensão do fluxo com o simulador CVC, para habilitar *timing checks* completos na GLS, exige avaliação da licença *Modified Artistic License* do CVC, gratuita para uso pessoal e acadêmico, mas sujeita a restrições em contextos comerciais.

A comparação com o testbench SystemVerilog evidenciou que, no contexto *open-source*, a linguagem Python oferece vantagens práticas em produtividade, depuração, capacidade de GLS e integração com ferramentas de análise, enquanto o SV enfrenta limitações estruturais impostas pelo Verilator (ausência de *covergroups*, CRV e suporte a netlist). Essa assimetria é específica ao ecossistema de simuladores gratuitos: em ambientes comerciais (VCS, Xcelium), o SV com UVM formal mantém vantagens significativas em CRV e integração nativa de cobertura.

O próximo passo prioritário é a integração do CVC ao fluxo cocotb para habilitar *timing checks* na GLS, o que completaria o ciclo de verificação desde o RTL até o *sign-off* temporal pós-layout sem qualquer ferramenta proprietária. Este objetivo é tecnicamente viável, uma vez que o CVC expõe a mesma interface VPI utilizada pelo Icarus, e o testbench Python não precisaria ser modificado.

Referências

ABDELBAKY, Dina; DESSOUKY, Mohamed; SALEM, Ashraf. Python-Based DRAM Memory Controller Testbench: Pyuvm an Early Report. **Journal of Advanced Research in Applied Sciences and Engineering Technology**, v. 52, n. 2, p. 176–188, 2025. DOI: 10.37934/araset.52.2.176188.

ANKITHA; ARADHYA, H. V. Ravish. A Python based Design Verification Methodology. **Journal of University of Shanghai for Science and Technology**, v. 23, n. 6, p. 901–911, 2021.

ASHMANSKAS, W. J. *et al.* Verification of simulated ASIC functionality and radiation tolerance for the HL-LHC ATLAS ITk Strip Detector. **Journal of Instrumentation**, v. 18, n. 03, p. c03047, 2023. DOI: 10.1088/1748-0221/18/03/C03047.

IEEE. **IEEE Standard for Verilog Hardware Description Language**. [*S. l.: s. n.*], 2005. DOI: 10.1109/IEEESTD.2006.99495.

LIU, C. *et al.* A Universal-Verification-Methodology-Based Testbench for the Coverage-Driven Functional Verification of an Instruction Cache Controller. **Electronics**, v. 12, n. 18, p. 3821, 2023. DOI: 10.3390/electronics12183821.

NXP SEMICONDUCTORS. **UM10204: I2C-Bus Specification and User Manual**. [*S. l.*], 2014. Disponível em: https://www.nxp.com/docs/en/user-guide/UM10204.pdf. Acesso em: 10 jul. 2026.

PIJA_EDUCATION. **I2C – Inter Integrated Circuit**. [*S. l.: s. n.*], 2026. https://pijaeducation.com/communication/i2c-inter-integrated-circuit/. Accessed: 08 Jul 2026.

RABAEY, Jan M.; CHANDRAKASAN, Anantha; NIKOLIĆ, Borivoje. **Digital Integrated Circuits: A Design Perspective**. 2nd. Upper Saddle River, NJ: Prentice Hall, 2003. ISBN 978-0130909961.

SOARES, Marcelo Rodrigues. **UVM-I2C-sky130: Ambiente de Verificação Funcional UVM em Python para o Controlador I2C no PDK SKY130**. [*S. l.: s. n.*], 2026. Repositório com código-fonte, scripts e artefatos de verificação. Acesso em: jun. 2026. Disponível em: https://github.com/MarceloDaEnc/UVM-I2C-sky130.

TACHYON DESIGN AUTOMATION. **open-src-cvc: OSS CVC Verilog Simulator with Full SDF Support**. [*S. l.: s. n.*], 2023. Distribuído sob licença *OSS CVC Dual Licensing Modified Artistic License*. Uso gratuito para fins pessoais e acadêmicos; uso comercial sujeito a autorização. Acesso em: 07 jun. 2026. Disponível em: https://github.com/cambridgehackers/open-src-cvc.

WESTE, Neil H. E.; HARRIS, David Money. **CMOS VLSI Design: A Circuits and Systems Perspective**. 4th. Boston, MA: Addison-Wesley, 2010. ISBN 978-0321547743.

WILE, Bruce; GOSS, John; ROESNER, Wolfgang. **Comprehensive functional verification: The complete industry cycle**. [*S. l.*]: Morgan Kaufmann, 2005.

WÖHR, Ferdinand *et al.* Coordination and complexity: an experiment on the effect of integration and verification in distributed design processes. **Design Science**, Cambridge University Press, 2023.