Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Departamento de Engenharia Elétrica

Trabalho de Conclusão de Curso

DESIGN E VERIFICAÇÃO FUNCIONAL DE UM BLOCO *I²C CONTROLLER*

Heriberto Gomes da Fonseca Junior

Campina Grande - PB

Junho de 2023

Heriberto Gomes da Fonseca Junior

DESIGN E VERIFICAÇÃO FUNCIONAL DE UM BLOCO *I²C CONTROLLER*

Trabalho de Conclusão de Curso submetido à Coordenação de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Engenheiro Eletricista.

Universidade Federal de Campina Grande - UFCG

Centro de Engenharia Elétrica e Informática - CEEI

Departamento de Engenharia Elétrica - DEE

Coordenação de Graduação em Engenharia Elétrica - CGEE

Gutemberg Gonçalves dos Santos Júnior, D.Sc.

(Orientador)

Campina Grande - PB

Junho de 2023

Heriberto Gomes da Fonseca Junior

DESIGN E VERIFICAÇÃO FUNCIONAL DE UM BLOCO *I²C CONTROLLER*

Trabalho de Conclusão de Curso submetido à Coordenação de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Engenheiro Eletricista.

Aprovado em ____ /____ /____

Marcos Ricardo Alcântara Morais
Universidade Federal de Campina Grande
Avaliador

Gutemberg Gonçalves dos Santos Júnior
Universidade Federal de Campina Grande
Orientador

Campina Grande - PB
Junho de 2023

Dedico este trabalho a Deus e a minha família, especialmente a minha mãe Andréa, meu pai Heriberto, minha irmã Gabriela e minha noiva Ester, por todo o apoio, amor e incentivo ao longo de todos esses anos. Certamente vocês são o motivo de eu chegar até aqui.

Agradecimentos

Ao Senhor Deus, autor da fé e criador de todas as coisas, por me dar forças e me sustentar em todas as tribulações e intempéries vividas durante a minha vida.

A minha mãe Andréa, meu pai Heriberto e minha irmã Gabriela que fizeram com que esse sonho fosse possível e estiveram presente durante todas as etapas da minha formação.

A minha noiva e eterna companheira Ester, que sempre acreditou em mim e me amou em todos os momentos, decidiu viver um relacionamento a distância e superou esse desafio ao meu lado, fazendo com que todos esses anos fossem mais felizes e divertidos.

Aos amigos e irmãos que fiz durante a vida, que dividiram momentos alegres e tristes comigo. Em especial Alan Pessoa, Arthur Macena, Ezequiel Brito, Hélder Guimarães, Lucas Dechenier, Lourival Netto e Matheus Petrúcio que dividiram os anos de estudo e formação comigo, bem como suas dificuldades e barreiras. Espero que todos vocês estejam sempre comigo na jornada da vida e que possamos compartilhar muitos momentos.

A todos que fazem parte do laboratório XMEN em especial a Guilherme Martins e André Igor, por me incentivarem a realizar as atividades (inclusive esse trabalho) dentro do prazo e compartilharem conhecimentos em microeletrônica e boas risadas.

Aos professores que tive durante a minha formação, em especial os professores Gutemberg Júnior e Marcos Morais, por todas as portas abertas, pelos anos de ensinamentos e por terem sido meus tutores para minha carreira profissional.

Enfim, agradeço a todos que cruzaram meu caminho nessa graduação e que tive o prazer de ajudar e ser ajudado.

"Tudo o que fizerem, façam de todo o coração, como para o Senhor, e não para os homens, sabendo que receberão do Senhor a recompensa da herança. É a Cristo, o Senhor, que vocês estão servindo."

Colossenses 3:23-24

Resumo

Este trabalho tem como propósito o desenvolvimento (*design* digital) e a verificação funcional de um bloco *I²C controller*, um componente crucial na comunicação de dados em sistemas embarcados. O protocolo *I²C* é amplamente utilizado na indústria eletrônica para a interconexão de dispositivos integrados, e o bloco desempenha o papel de controlador principal nesse barramento. O objetivo deste projeto é projetar e verificar um bloco *I²C controller*, considerando aspectos de hardware e/ou software, como a definição da arquitetura, a interface de comunicação, a lógica de controle e os modos de operação básicos de leitura e escrita. A verificação funcional é uma etapa crucial para garantir o correto funcionamento do bloco *I²C controller*, e serão desenvolvidos testes para verificar seu comportamento em um cenário de comunicação definido pela presença de apenas um *target* e a transmissão de um dado por transação no modo normal. O projeto também abordará o desenvolvimento do *design* digital do bloco *I²C controller* que atuará nas condições já mencionadas, levando em conta a especificação feita pela NXP Semiconductors©. A verificação funcional e o *design* digital serão implementadas utilizando a linguagem de descrição de *hardware SystemVerilog*. Para garantir o funcionamento do bloco, foi feita a sua síntese e a simulação do ambiente de verificação utilizando os *softwares*: Cadence© Xcelium Logic Simulator™, Cadence© Genus Synthesis Solution™, Cadence© Simvision Waveform™ e Cadence© Integrated Metrics Center™.

Palavras-chave: Desenvolvimento de *hardware*, Microeletrônica, Verificação funcional, UVM, *Design* digital, Protocolo *I²C*.

Abstract

This work aims to develop a digital design and functional verification of an I²C controller, a crucial component in data communication for embedded systems. The I²C protocol is widely used in the electronics industry for interconnecting integrated devices, and the controller plays the role of the main controller in this bus. The objective of this project is to design and verify an I²C controller block, considering hardware and/or software aspects such as architecture definition, communication interface, control logic, and basic read and write operation modes. Functional verification is a crucial step to ensure the correct functioning of the I²C controller block, and tests will be developed to verify its behavior in a communication scenario defined by the presence of only one target and the transmission of one data per transaction in normal mode. The project will also address the digital design development of the I²C controller block, which will operate under the mentioned conditions, taking into account the specifications provided by NXP Semiconductors©. Functional verification and digital design will be implemented using the hardware description language SystemVerilog. To ensure the block's functionality, synthesis and verification environment simulation were performed using the following software: Cadence© Xcelium Logic Simulator™, Cadence© Genus Synthesis Solution™, Cadence© Simvision Waveform™, and Cadence© Integrated Metrics Center™.

Keywords: Hardware development, Microelectronics, Functional verification, UVM, Digital design, I²C protocol.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ABNT	Associação Brasileira de Normas Técnicas
ASIC	*Application-Specific Integrated Circuit*
FPGA	*Field-Programmable Gate Array*
PDK	*Process Development Kit*
OOP	*Object Orientated Programming*
UVM	*Universal Verification Methodology*
IEEE	*Institute of Electrical and Electronics Engineers*
I^2C	*Inter-Integrated Circuit*
IP	*Intelectual Property*
EDA	*Electronic Design Automation*
DUT	*Design Under Test*
TLM	*Transaction-Level Modeling*
SDA	*Serial Data Line*
SCL	*Serial Clock Line*
RTL	*Register Transfer Level*
FSM	*Finity State Machine*

Sumário

1 Introdução

O avanço tecnológico e a crescente demanda por dispositivos eletrônicos de alta performance têm impulsionado o desenvolvimento de sistemas cada vez mais complexos e integrados. Nesse contexto, a comunicação entre componentes eletrônicos se torna fundamental para o correto funcionamento desses sistemas. O barramento I^2C (Inter-Integrated Circuit) tem sido amplamente utilizado como uma solução eficiente e versátil para a interconexão de dispositivos em diversos segmentos da indústria.

Muitos periféricos populares para sistemas embarcados suportam interfaces seriais. O I^2C é um protocolo simples de curto alcance e baixa largura de banda, desenvolvido pela Philips Semiconductors© no início dos anos 1980. Atualmente, o protocolo é gerenciado e especificado pela NXP Semiconductors©, a qual faz manutenções e atualizações de forma periódica. Como pode ser visto (SEMICONDUCTOR, 2021), ele pode ter endereçamento de 7 bits ou 10 bits e utiliza duas linhas bidirecionais: *serial clock* (SCL) e *serial data* (SDA). Todos os dispositivos na linha podem atuar como alvos ou controladores. A linha de clock é controlada apenas pelos dispositivos controladores. O envio de dados de 8 bits e endereçamento de 7 bits com bits de controle é feito em série por meio da interface. Além disso, os dispositivos controladores tem o controle, na maior parte do tempo, dos dados enviados pela linha de dados seriais, atuando como transmissor e gerenciando o início e o final da transmissão de dados. Dessa forma, tais dispositivos são essenciais para o funcionamento do protoco I^2C e do controle do fluxo de dados seriais, fazendo com que a maior parte do esforço na confecção sejam voltados para eles.

Este trabalho tem como objetivo o *design* digital e a verificação funcional de um bloco controlador I^2C. Serão explorados conceitos de projeto de circuitos digitais, linguagem de descrição de *hardware* e metodologias de verificação, com o intuito de desenvolver um bloco confiável, eficiente e compatível com os padrões e especificações do protocolo I^2C. Para que isso fosse possível, foi pensado em um cenário no qual a comunicação ocorre entre um controlador e um único alvo, com a transmissão de um único dado por transação. Como esse é o cenário básico e primordial, garantindo-se sua confiabilidade e eficiência, é possível reutilizá-lo e expandí-lo para os cenários práticos encontrado nos sistemas que utilizam múltiplos dispositivos embarcados.

O *design* digital foi pensado a partir das metodologias de desenvolvimento ASIC (*Application-Specific Integrated Circuit*) e a sua implementação foi feita utilizando a linguagem de descrição de hardware *SystemVerilog*. Já a verificação funcional do bloco foi feita a partir da metodologia UVM (*Universal Verification Methodology*), tendo em vista sua reusabilidade, eficiência e praticidade. Isso é possível uma vez que com o *SystemVerilog*,

foi trazida uma *feature* que compatibiliza o uso de OOP (*Object Orientated Programming*) nessa linguagem, fazendo possível o uso das classes UVM e seus recursos, tais como herança, derivação e virtualização.

Para a realização da síntese digital, foram utilizados o *PDK* da X-FAB® e a ferramenta de síntese da Cadence®, Genus Synthesis Solution™.

A utilização *PDK* X-FAB® proporciona aos projetistas de ASIC a capacidade de personalizar seus circuitos de acordo com requisitos específicos, permitindo alcançar desempenho, eficiência energética e outras características desejadas. Ele encontra ampla aplicação em diversos setores, como comunicação sem fio, dispositivos de consumo, sistemas automotivos e outras áreas de alta tecnologia.

Para a simulação do *design* digital, bem como sua depuração para garantir o funcionamento correto foram utilizadas as ferramentas da Cadence®, Xcelium Logic Simulator™ e SimVision Waveform™. Tais ferramentas são essenciais para que o código desenvolvido tenha sua confiabilidade testada.

Por fim, para a análise da cobertura de código e da cobertura funcional, foi utilizada a ferramenta da Cadence®, Integrated Metrics Center™. Tal ferramenta permite visualizar informações essenciais para garantir que o *design* foi estimulado com as mais diversas entradas, além de gerar relatórios para análise dessas métricas.

1.1 Objetivos Gerais

O objetivo geral do trabalho é propor e desenvolver um *design* digital de um controlador do protocolo I^2C e construir um ambiente de verificação funcional utilizando *UVM* para validar o funcionamento do bloco, garantindo sua confiabilidade e eficiência em um cenário de utilização básico.

1.2 Objetivos Específicos

Os objetivos específicos do trabalho são listados abaixo:

- Fixar e praticar os métodos de desenvolvimento de circuitos digitais, bem como a prática da linguagem de descrição de *hardware SystemVerilog*
- Aprender o protocolo de comunicação I^2C e suas especificações
- Estudar os diferentes métodos existentes na metodologia *UVM* para o desenvolvimento de um ambiente de verificação funcional;
- Desenvolver um *testbench* capaz de estimular das mais diversas formas o *design* e cobrir o máximo de testes possíveis para a validação.

- Consolidar a prática no uso das ferramentas da Cadence®, bem como seus recursos e ferramentas.

1.3 Organização do Trabalho

O trabalho está estruturado em 5 capítulos, incluindo este introdutório, conforme a seguir.

O **Capítulo 2** apresenta a fundamentação teórica, expondo os conceitos básicos de desenvolvimento de circuitos digitais e verificação funcional, a metodologia *UVM* e as especificações do protocolo I^2C.

O **Capítulo 3** aborda a metodologia utilizada para o desenvolvimento do *design* digital e do ambiente de verificação funcional, bem como suas peculiaridades e as soluções criadas para a resolução de erros e problemas.

No **Capítulo 4** são apresentados os resultados das simulações do *design* digital, da síntese digital do bloco e as métricas de validação obtidas.

O **Capítulo 5** apresenta as conclusões do trabalho, expondo as impressões gerais e possíveis formas de extender e melhorar o projeto.

2 Fundamentação Teórica

Neste capítulo, será apresentada uma fundamentação teórica sobre *design* de circuitos digitais, verificação funcional e *UVM*, e algumas especificações de funcionamento do protocolo I^2C.

2.1 *Design* de Circuitos Digitais

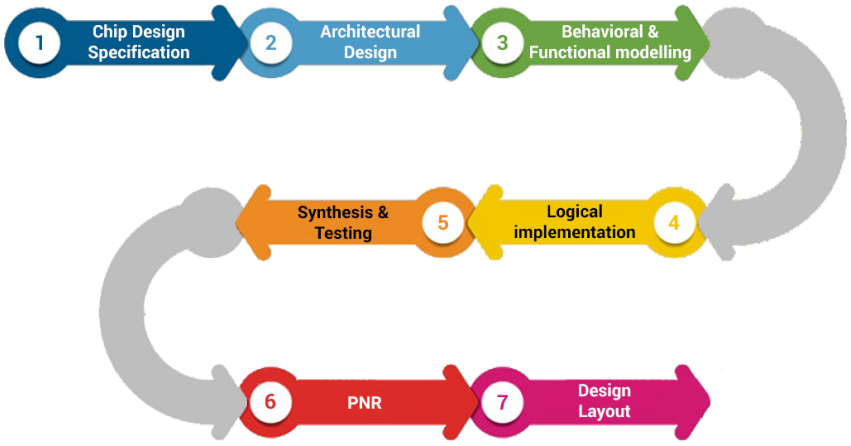
O *design* de circuitos digitais é uma área especializada que lida com a concepção, projeto e implementação de circuitos digitais em níveis muito baixos de integração, como chips e sistemas em um único chip (SoCs). Esses circuitos digitais são essenciais para o funcionamento de dispositivos eletrônicos modernos, como computadores, *smartphones*, sistemas embarcados e muito mais. Eles envolvem a integração de componentes eletrônicos, como transistores, resistores, capacitores, entre outros, em um único chip.

Ao projetar circuitos digitais para microeletrônica, alguns dos principais desafios incluem o aumento da densidade de integração, a redução do consumo de energia, o aumento da velocidade de operação e a garantia de confiabilidade e robustez.

Uma das áreas de desenvolvimento de circuitos digitais mais abrangentes nos dias atuais é a área de *design ASIC*. É uma área especializada e que se concentra na criação de chips personalizados para aplicações específicas. Diferentemente dos chips de propósito geral, como microprocessadores, ASICs são projetados para executar tarefas específicas e são otimizados para desempenho, consumo de energia e custo.

O processo de *design ASIC* envolve várias etapas, como a definição dos requisitos do circuito, a criação da especificação de design, a implementação do circuito em um nível de abstração de alto nível, a simulação e verificação, a síntese lógica, o layout físico, a verificação pós-layout e a fabricação do chip, como pode ser visto em (MARTIN, 2002).

Figura 1 – Fluxo de *design ASIC*



Fonte: (CHAUHAN, 2020).

2.2 Síntese Lógica

A síntese lógica de *hardware* desempenha um papel essencial no desenvolvimento de circuitos integrados. Ela consiste em converter uma descrição de alto nível de um circuito digital em uma representação de baixo nível adequada para a implementação em um chip. O principal objetivo da síntese lógica é otimizar o *design* em termos de área ocupada, consumo de potência e desempenho, garantindo que o circuito funcione corretamente (HOLLINGWORTH, 1990).

Durante o processo de síntese, várias etapas são executadas. Inicialmente, ocorre o mapeamento de portas lógicas, em que as descrições de alto nível, como *RTL*, são transformadas em uma representação utilizando uma biblioteca de células lógicas padrão. Essas células lógicas representam as operações básicas, como portas *AND, OR e NOT*, que são implementadas usando transistores. Em seguida, é realizada a otimização de nível e tecnologia, na qual são aplicadas técnicas para melhorar o desempenho do circuito em termos de área, potência e velocidade. Isso envolve a substituição de células lógicas por outras mais eficientes e a reorganização do circuito para reduzir a área ocupada e minimizar as conexões entre os elementos. Por fim, é feita a alocação de recursos, que determina como os elementos físicos, como transistores e interconexões, serão posicionados no chip. Nessa etapa, são considerados fatores como a localização dos elementos e a minimização do atraso de propagação entre eles.

No mercado, existem diversas ferramentas de síntese lógica disponíveis, que suportam diferentes níveis de abstração e oferecem recursos avançados para otimização do *design*. Essas ferramentas desempenham um papel crucial no desenvolvimento de circuitos integrados eficientes e funcionais, permitindo que os projetistas aproveitem ao máximo os

recursos disponíveis.

2.3 *SystemVerilog*

A linguagem de descrição de *hardware SystemVerilog* desempenha um papel fundamental no projeto de circuitos digitais, fornecendo uma linguagem de alto nível para modelagem e simulação do comportamento de sistemas digitais complexos (SUTHERLAND; DAVIDMANN; FLAKE, 2006). É uma extensão poderosa do *Verilog*, contendo recursos adicionais que aprimoram a capacidade de descrever, simular e verificar projetos de *hardware*.

Uma das principais vantagens do *SystemVerilog* é sua capacidade de descrever circuitos digitais em diferentes níveis de abstração. Ele suporta descrições estruturais, onde os componentes do circuito são conectados por portas e sinais, e descrições comportamentais, onde o comportamento do circuito é especificado usando regras e equações. Essa flexibilidade permite que os projetistas escolham o nível de detalhe que melhor se adapta à tarefa em questão. Uma característica importante do *SystemVerilog* é a digitação de dados. Tipos de dados poderosos, como números inteiros, reais, vetores e estruturas definidas pelo usuário, são introduzidos para tornar a modelagem de circuitos mais precisa e robusta. Além disso, a linguagem suporta primitivas de concorrência que permitem a escrita de circuitos assíncronos e paralelos que podem executar múltiplas operações simultaneamente.

Outro aspecto importante do *SystemVerilog* é a verificação do projeto. Ele fornece recursos poderosos para criar asserções (VIJAYARAGHAVAN; RAMANATHAN, 2005), que são instruções lógicas para testar propriedades específicas do circuito. Isso ajuda os *designers* a encontrar erros e garantir que o design esteja correto. Além disso, o *SystemVerilog* oferece suporte à verificação formal, uma técnica avançada que permite a verificação automática das propriedades do projeto sem simulação, contribuindo para uma verificação mais abrangente e eficiente.

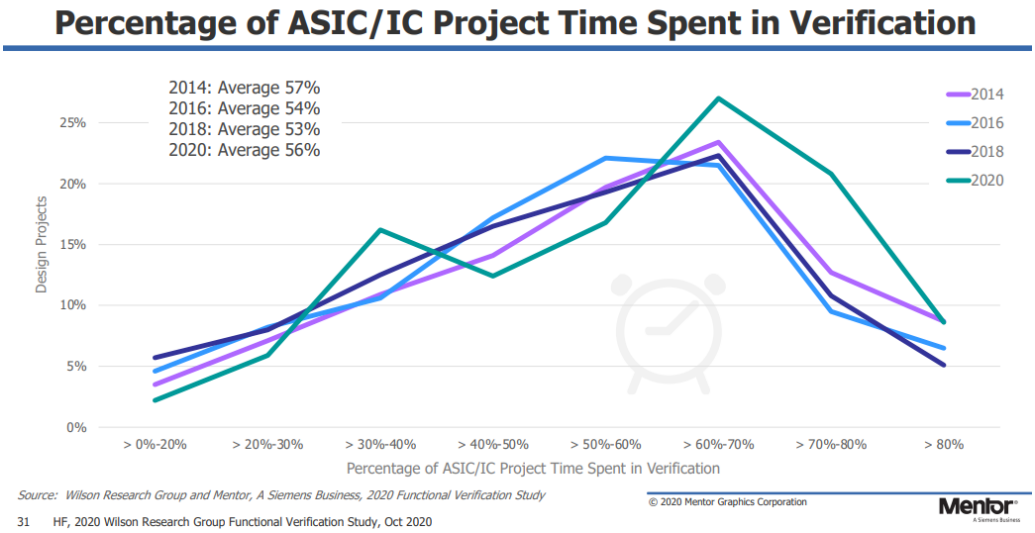
O SystemVerilog é especificado e mantido pela *IEEE* através do padrão *IEEE Std 1800*. O padrão define a linguagem, incluindo sua sintaxe, semântica e recursos.

2.4 Verificação Funcional

A verificação funcional é uma etapa essencial no desenvolvimento de circuitos digitais, que visa garantir a equivalência entre as especificações definidas e a implementação em *RTL*. Com a crescente complexidade dos blocos *ASIC* modernos, tornou-se inviável realizar testes exaustivos, exigindo estratégias mais eficazes para cobrir o maior número possível de casos de teste, considerando as restrições de tempo e recursos.

A verificação representa cerca de 70% do esforço total de *design*, sendo uma etapa crucial no processo de desenvolvimento (KUMAR; GOPINATH, 2016). Ela está no caminho crítico do projeto, demandando tempo significativo. No entanto, é possível reduzir o tempo de verificação por meio da utilização de abstração e uso de *IP* e *designs* pré-verificados. A abstração na verificação permite simplificar o processo, reduzindo o foco em detalhes de baixo nível. Embora isso possa resultar em menos controle sobre esses detalhes, é uma estratégia eficaz para agilizar a verificação. A Figura 2 ilustra o tempo gasto na verificação em relação ao tempo total do desenvolvimento de um projeto *ASIC*.

Figura 2 – Percentual do tempo do projeto de *ASIC/IC* gasto em verificação.



Fonte: (FOSTER, 2020).

Além disso, a automação é uma ferramenta essencial para a redução do tempo de verificação. Ao automatizar tarefas repetitivas, é possível acelerar o processo e aumentar a eficiência. A aleatorização também pode ser utilizada como uma ferramenta de automação na verificação. Ao gerar estímulos de teste de forma aleatória, é possível explorar uma ampla gama de cenários de uso e identificar potenciais problemas de forma mais eficiente.

As principais métricas da verificação funcional são as coberturas de código e funcional. Elas permitem identificar o quão abrangente são os estímulos gerados pelo ambiente de verificação em comparação com as possibilidades existentes, a partir do tamanho em *bits* das variáveis.

2.4.1 Cobertura de Código

A cobertura de código é uma métrica que avalia o quanto o código do *design* foi testado usando o banco de testes. Ela mede a extensão em que diferentes partes do código

foram exercitadas, incluindo expressões, alternância de pinos e máquinas de estados finitos (FATHY; SALAH; GUINDI, 2015). No entanto, é importante destacar que a cobertura de código não fornece informações sobre se o *design* foi testado de acordo com sua intenção original.

Embora seja uma ferramenta útil para avaliar a abrangência dos testes, ela não garante que todos os possíveis caminhos e comportamentos do *design* tenham sido exercitados. Para uma avaliação mais abrangente, é necessário complementar a cobertura de código com a cobertura funcional, que verifica se as funcionalidades e requisitos específicos do design foram adequadamente testados.

A cobertura de código é geralmente gerada automaticamente pela ferramenta de verificação utilizada, permitindo identificar quais partes do código foram executadas durante os testes. Essa métrica auxilia os engenheiros de verificação a identificar lacunas nos testes e a direcionar esforços para garantir uma cobertura mais completa do design.

2.4.2 Cobertura Funcional

A cobertura funcional é uma medida que analisa se o *design* está em conformidade com sua especificação funcional e identifica possíveis desvios em relação a essa especificação. O uso das palavras-chaves *cover, covergroups, bins e cross* permitem verificar se determinadas condições ou funcionalidades estão sendo adequadamente exercitadas durante os testes(SALAH, 2014).

A cobertura funcional vai além da simples execução do código, pois envolve uma análise mais profunda das funcionalidades do design e sua correspondência com a especificação. Ela ajuda a garantir que todas as principais funcionalidades do design sejam testadas e fornece informações valiosas sobre a integridade e conformidade do sistema.

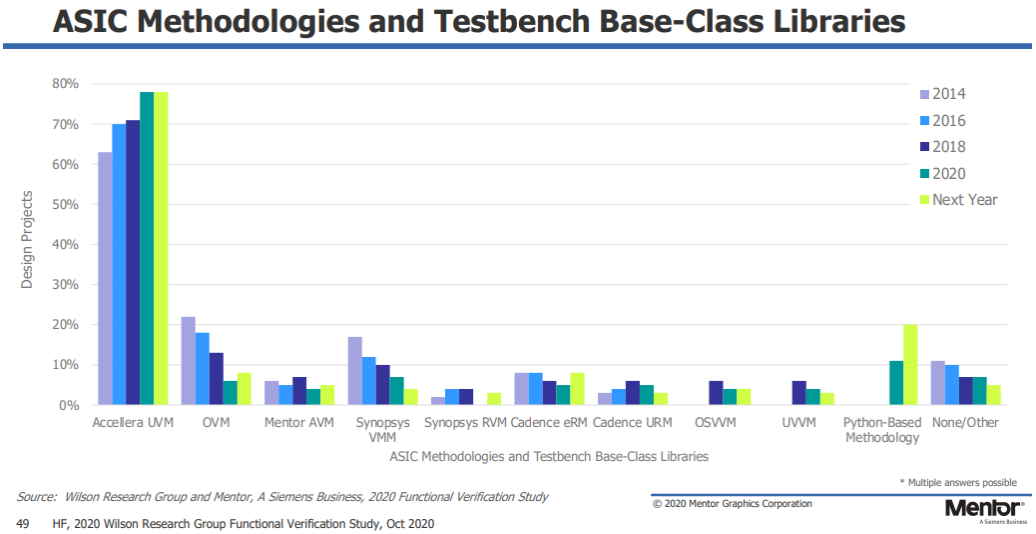
Ao estabelecer metas de cobertura funcional, os engenheiros de verificação definem os critérios para avaliar se o design está atendendo às expectativas e requisitos funcionais estabelecidos. Essa abordagem é fundamental para garantir que o design esteja corretamente implementado e que as funcionalidades importantes estejam sendo testadas de maneira adequada.

2.5 *Universal Verification Methodology (UVM)*

A *UVM* é uma metodologia amplamente adotada para a verificação funcional de *hardware*. Baseada em orientação a objetos e utilizando a linguagem *SystemVerilog*, ela oferece uma estrutura reutilizável para criar ambientes de verificação eficientes. Com sua portabilidade, a *UVM* pode ser utilizada com diversas ferramentas de *EDA*, permitindo a reutilização do ambiente de verificação em diferentes plataformas de simulação. Foi

desenvolvida pela Accellera Systems Initiative® e incentiva a reutilização de *testbenches* e componentes de verificação, facilitando a criação de casos de teste e o desenvolvimento de modelos transacionais para melhorar a eficiência do processo de verificação.

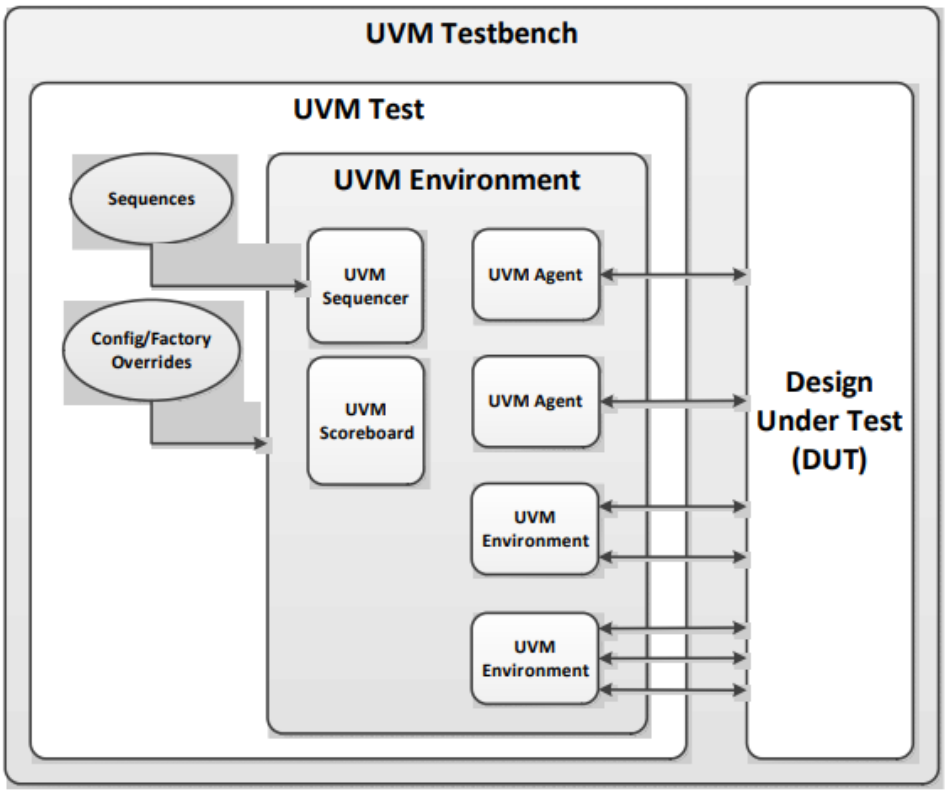
Figura 3 – Histórico de uso de Metodologias *ASIC* e Bibliotecas de Classe Base de *Testbench*



Fonte: (FOSTER, 2020)

A metodologia *UVM* segue uma arquitetura típica para a concepção de *testbenches*. A Figura 4 ilustra a hierarquia da arquitetura e seus componentes. Logo após, serão definidos os componentes dessa arquiteturas especificados em (INITIATIVE, 2015).

Figura 4 – Arquitetura típica de um *Testbench UVM*



Fonte: (INITIATIVE, 2015)

2.5.1 *UVM Testbench*

A arquitetura típica do *UVM Testbench*, também conhecido como módulo *top*, envolve a instanciação do módulo *DUT* e da classe de teste *UVM Test*, além da configuração das conexões entre eles. Se os elementos de verificação incluírem componentes baseados em módulos, eles também são instanciados sob o *UVM Testbench*. O *UVM Test* é instanciado dinamicamente em tempo de execução, permitindo que o *UVM Testbench* seja compilado uma vez e executado com vários testes diferentes.

2.5.2 *UVM Test*

O *UVM Test* é o componente de nível superior no *UVM Testbench*. Sua função principal é realizar três atividades: instanciar o ambiente de nível superior, configurar o ambiente (por meio de substituições de fábrica ou do banco de dados de configuração) e aplicar estímulos invocando Sequências *UVM* através do ambiente para o *DUT*.

Geralmente, há um *UVM Test* base que contém a instanciação do ambiente UVM e outros elementos comuns. Em seguida, outros testes individuais estendem esse teste base

e podem configurar o ambiente de forma diferente ou selecionar sequências diferentes para executar.

2.5.3 *UVM Environment*

Um *UVM Environment* abrange os componentes *agents* e *scoreboards* e, frequentemente, outros ambientes em sua hierarquia. Ele reúne diversos componentes críticos do *testbench*, permitindo que sejam configurados facilmente em um único local, se necessário, em qualquer estágio. Pode ter vários *agents* para diferentes interfaces e múltiplos *scoreboards* para verificar diferentes tipos de transações de dados. Dessa forma, o ambiente permite habilitar ou desabilitar diferentes componentes de verificação para tarefas específicas em um único local.

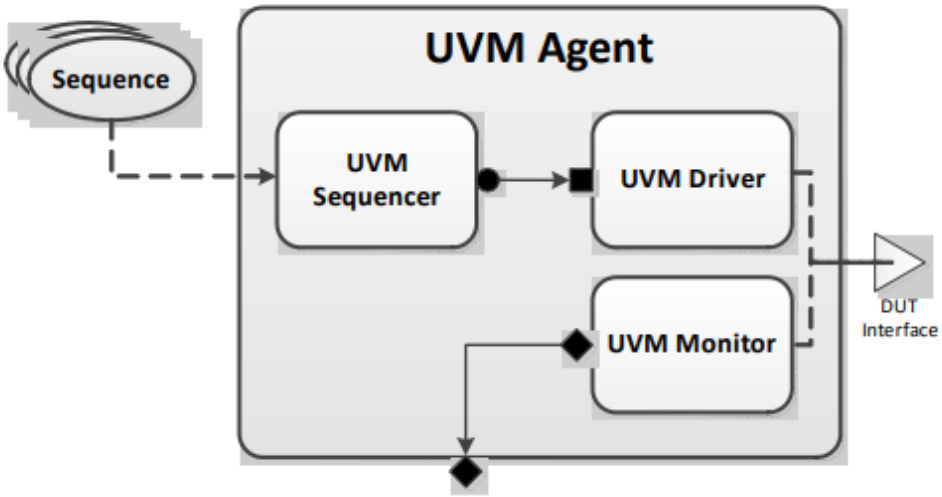
2.5.4 *UVM Scoreboard*

A função principal do *UVM Scoreboard* é verificar o comportamento de um determinado DUT. O *UVM Scoreboard* geralmente recebe transações contendo as entradas e saídas do DUT por meio das portas de análise do *UVM Agent*. Ele executa as transações de entrada por meio de um modelo de referência (também conhecido como preditor) para produzir transações esperadas e, em seguida, compara as saídas esperadas com as saídas reais. Existem diferentes metodologias para implementar o *scoreboard*, variando na natureza do modelo de referência e na forma de comunicação entre o *scoreboard* e o restante do *testbench*.

2.5.5 *UVM Agent*

Um *UVM Agent* consiste em um agrupamento dos componentes *sequenciador, driver e monitor* de uma interface. Pode-se utilizar vários agentes para controlar múltiplas interfaces, e todos eles são conectados ao *testbench* por meio do componente de ambiente. Existem agentes ativos e passivos. Os agentes ativos possuem um *driver* e têm a capacidade de controlar sinais, enquanto os agentes passivos possuem apenas o monitor e não podem controlar os pinos. Mesmo que um agente passivo seja composto apenas pelo monitor, é importante manter o nível de abstração prometido pelo *UVM* e preservar sua estrutura, colocando todos os agentes no ambiente em vez de subcomponentes isolados, como monitores. Por padrão, um agente é considerado ativo, mas isso pode ser alterado usando o método *set*() do banco de dados de configuração UVM. A Figura 5 ilustra a arquitetura típica de um *UVM Agent*.

Figura 5 – Arquitetura típica de um *UVM Agent*



Fonte: (INITIATIVE, 2015)

2.5.6 *UVM Sequence Item*

Um *UVM Sequence Item* é o elemento fundamental da arquitetura *UVM*. É uma transação que contém dados, métodos e possivelmente restrições associadas ao *DUT*. O *UVM Sequence Item* representa a menor unidade transacional que pode ocorrer em um ambiente de verificação.

2.5.7 *UVM Sequence*

Um *UVM Sequence* é um conjunto de transações. Uma sequência possibilita a capacidade de usar a transação conforme os requisitos e utilizar quantas transações quiser. A principal função de uma sequência é gerar transações e enviá-las para o sequenciador.

2.5.8 *UVM Sequencer*

Um *UVM Sequencer* desempenha o papel de intermediário entre as transações e o driver, controlando o fluxo de transações provenientes de várias sequências. Uma interface *TLM* possibilita a comunicação entre o *driver* e o sequenciador.

2.5.9 *UVM Driver*

Um *UVM Driver* converte as transações recebidas do sequenciador em atividades de nível de sinal no *DUT*. Para realizar essa conversão, são utilizados métodos como

get_next_item(), *try_next_item*(), *item_done*() e *put*(). O *UVM Driver* é, geralmente, uma classe parametrizada com o tipo de transação utilizada por ela.

2.5.10 *UVM Monitor*

O *UVM Monitor* monitora a atividade em nível de sinal do DUT e a converte em transações para serem enviadas a outros componentes para análise adicional. Normalmente, o monitor processa as transações, como coleta de cobertura e registro, antes de enviá-las aos *scoreboards*.

2.6 Protocolo *I²C*

O protocolo de comunicação serial *I²C* é uma solução amplamente empregada para a comunicação entre dispositivos eletrônicos em uma variedade de aplicações. Desenvolvido pela Philips® (atualmente NXP Semiconductors®), o *I²C* oferece um meio eficiente e de fácil implementação para a troca de dados entre diferentes componentes de um sistema. O *I²C* é baseado em um barramento serial de dois fios, composto por uma linha de dados (*SDA*) e uma linha de clock (*SCL*). Essas linhas são compartilhadas por todos os dispositivos conectados ao barramento, com cada dispositivo possuindo um endereço único que possibilita sua identificação durante as comunicações.

O protocolo *I²C* suporta dois modos de operação: controlador e alvo. O dispositivo controlador é responsável por iniciar e controlar as transmissões, enquanto os dispositivos alvos respondem aos comandos enviados pelo controlador. O controlador também gerencia o acesso ao barramento, evitando colisões e garantindo uma comunicação adequada. A comunicação *I²C* ocorre por meio de quadros de *bytes*, chamados de quadros *I²C*, que contêm um endereço de destino, identificando o dispositivo de destino, juntamente com os dados a serem transmitidos. O protocolo *I²C* também oferece recursos como repetição de início (*start*) e término (*stop*), permitindo a comunicação contínua entre dispositivos.

Uma das principais vantagens do protocolo *I²C* é sua simplicidade e baixo consumo de recursos. Ele possibilita a conexão de vários dispositivos em um único barramento, facilitando a integração de componentes em um sistema. Além disso, o *I²C* permite a configuração da velocidade de comunicação, tornando-a ajustável conforme as necessidades do projeto. O protocolo *I²C* é amplamente adotado em diversas aplicações, incluindo comunicação entre sensores e microcontroladores, controle de periféricos e memórias, entre outros. É uma solução confiável, amplamente suportada pelos fabricantes e amplamente utilizada na indústria eletrônica.

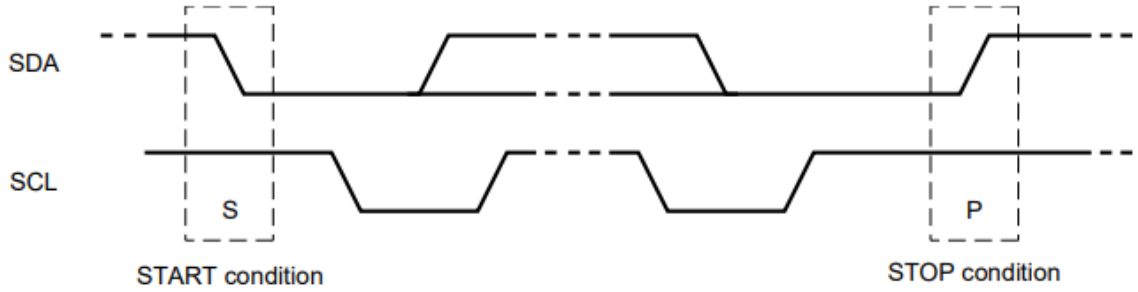
O protocolo possui diversos recursos mostrados em (SEMICONDUCTOR, 2021). No entanto, serão abordados apenas os seguintes recursos:

- Condição de início e parada (*Start and stop condition*)
- Operações de leitura e escrita
- Reconhecimento (*acknowledgement*) e validade do dado serial

2.6.1 Condição de Início e Parada

Como descrito em (KAITH; PATEL; GUPTA, 2015), toda transmissão de dados via protocolo I^2C inicia-se a partir de uma condição de início e se encerra por meio de uma condição de parada, ambas caracterizadas pelos sinais do barramento *SDA e SCL*. A condição de início é feita pelo controlador ao proporcionar uma transição de nível lógico alto (1) para o nível lógico baixo (0) em *SDA*, enquanto que *SCL* é mantido em nível lógico alto (1). Já a condição de parada acontece quando há um transição de nível lógico baixo (0) para o nível lógico alto (1) em *SDA*, enquanto que *SCL* é mantido em nível lógico alto (1).

Figura 6 – Condição de Início e Parada

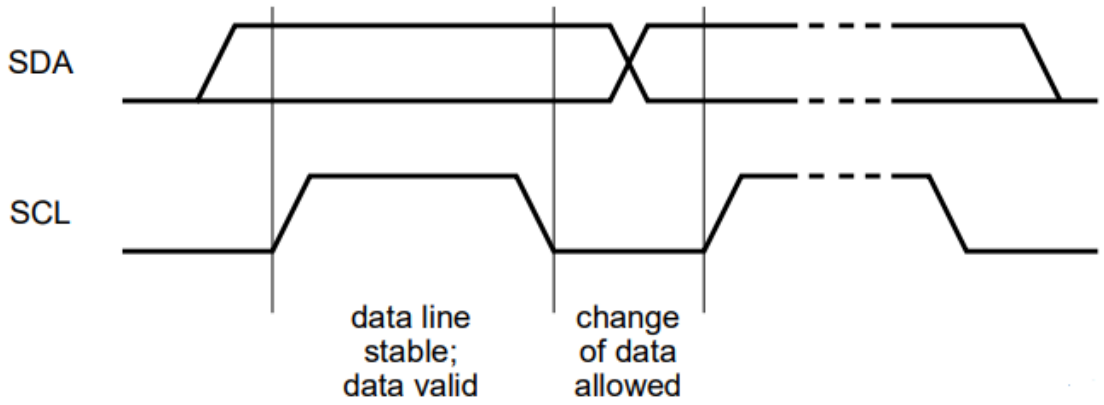


Fonte: (SEMICONDUCTOR, 2021)

2.6.2 Validade do Dado

A validade do dado serial na linha *SDA*, segundo (KAITH; PATEL; GUPTA, 2015), ocorre quando a linha *SCL* está em nível lógico baixo (0). Portanto, nesse momento, é possível que os alvos e controladores obtenham dados válidos em *SDA* para escrita ou leitura. Já quando a linha *SCL* está em nível lógico alto (1), é permitida a troca de dados na linha *SDA*.

Figura 7 – Validade do Dado



Fonte: (SEMICONDUCTOR, 2021)

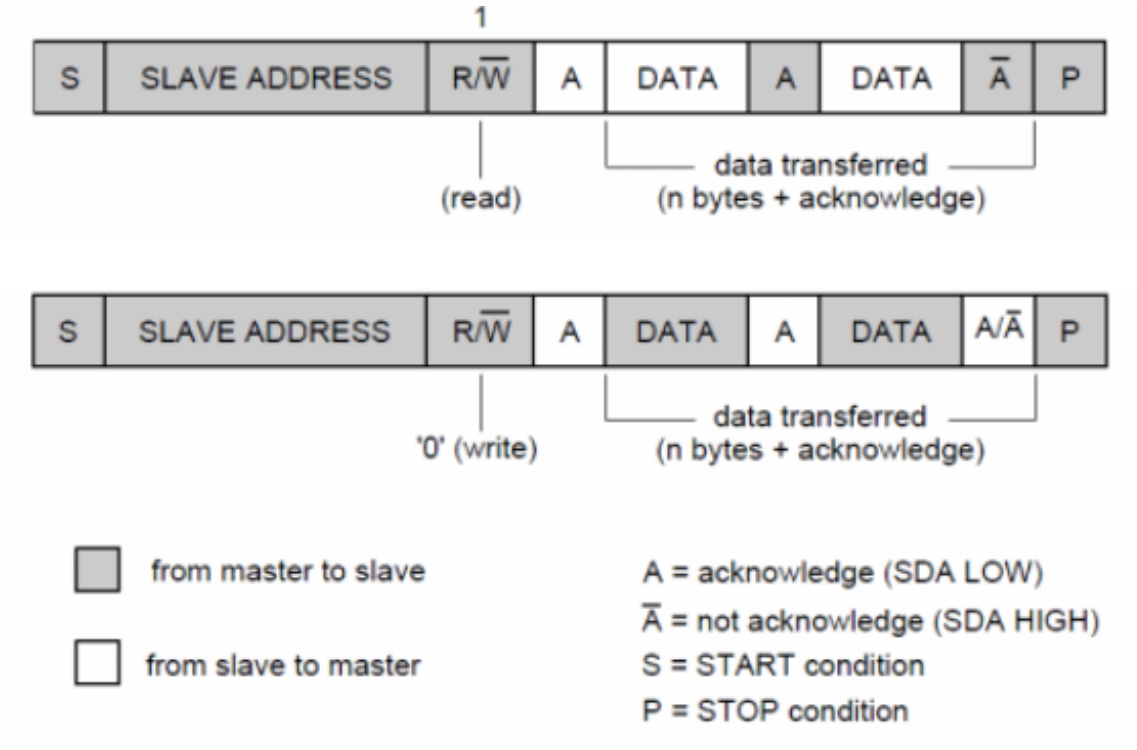
2.6.3 Operação de Leitura e Escrita

Durante uma operação de escrita no I^2C, o dispositivo controlador inicia a transmissão enviando um quadro I^2C contendo o endereço do dispositivo alvo e os dados a serem escritos. O dispositivo alvo correspondente reconhece seu endereço enviando um bit de reconhecimento (em nível lógico alto) e recebe os dados, realizando a ação apropriada com base nas informações recebidas.

Por outro lado, na operação de leitura, o dispositivo controlador inicia a transmissão enviando um quadro I^2C com o endereço do dispositivo alvo e um bit indicando que uma leitura será realizada. O dispositivo alvo correspondente reconhece o endereço e prepara os dados para serem enviados de volta ao dispositivo controlador. Em seguida, o dispositivo controlador recebe os dados enviados pelo dispositivo alvo e manda um bit de reconhecimento (1 para ler o próximo dado e 0 para encerrar a transmissão).

No contexto de um sistema eletrônico, as operações de leitura e escrita no protocolo I^2C são controladas pelo dispositivo controlador, que coordena a comunicação, envia os comandos e recebe as respostas dos dispositivos alvos. É crucial que os dispositivos estejam configurados corretamente com os endereços apropriados e que as sequências de escrita e leitura sejam seguidas corretamente para garantir uma comunicação eficaz entre os dispositivos. A flexibilidade bidirecional do protocolo I^2C, que permite a troca de dados em ambos os sentidos, torna-o amplamente utilizado em uma variedade de aplicações eletrônicas. Ele oferece uma solução confiável e eficiente para a comunicação entre dispositivos, permitindo a integração efetiva de componentes em sistemas eletrônicos complexos.

Figura 8 – Operação de Leitura e Escrita



Fonte: (SEMICONDUCTOR, 2021)

3 Metodologia de Desenvolvimento

Esse capítulo discute acerca da metodologia utilizada para o desenvolvimento do *design* digital do controlador I^2C, bem como suas características e limitações. Além disso, trata também sobre o ambiente de verificação funcional, suas características e componentes, e sua arquitetura geral.

3.1 *Design* Digital

O *design* do controlador I^2C foi pensado de acordo com as especificações do protocolo mencionadas na Seção 2.6. O enlace de comunicação proposto é de cárater básico e essencial, dado por:

- Comunicação entre apenas um controlador e um alvo.
- Transmissão de um único dado por vez (seja de leitura ou escrita).
- O alvo possui endereço único e, portanto, seu reconhecimento é sempre positivo à chamada do controlador no barramento.
- O controlador é isento do tratamento do não reconhecimento de um dado, apenas reportando a falha.

Apesar de ser um enlace simples, ele tem um caráter fundamental no funcionamento correto da comunicação dos dispositivos. Tendo isso como ponto de partida, é totalmente viável expandir os recursos do controlador e fazer testes mais robustos com a presença de mais alvos no barramento.

Diante disso, faz-se necessário uma descrição da lógica programacional utilizada para conceber o controlador I^2C, bem como suas características funcionais e limitações, visto que o protocolo I^2C completo possui mais recursos que serão abordados no tópico.

3.1.1 Interface

O controlador I^2C projetado foi pensado com o foco no controle do barramento serial e nas operações do protocolo. A geração dos sinais que executam esse controle foi abstraída, uma vez que a verificação será feita segundo o escopo do protocolo I^2C. Dessa forma, esses sinais foram colocados como entradas do bloco, facilitando a geração de estímulos e simplificando a lógica do algoritmo. O controlador possui a seguinte interface de entrada:

```
interface i2c_interface (input bit clk, input bit reset);

  // Inputs
  logic start;
  logic stop;
  logic rw;
  logic [6:0] addr;
  logic [7:0] w_data;

  // Outputs
  logic i2c_scl;
  logic i2c_sda;


  modport master(input clk, reset, start, stop, rw, addr, w_data,
                 output i2c_scl, inout i2c_sda);
  // modport mst(input  scl, inout sda, input clk, input rst);
endinterface
```

Fonte: Autoria própria.

Os sinais *start e stop* são os responsáveis pela sinalização ao controlador que deve ser iniciada/finalizada uma transmissão de dados. A partir deles, o controlador executa a condição de início/parada no barramento, especificada na Seção 2.6.1. O sinal *rw* indica ao controlador a operação que deverá ser feita na transmissão, seja ela escrita ou leitura. O valor desse sinal será enviado ao barramento logo após a finalização do envio do endereço.

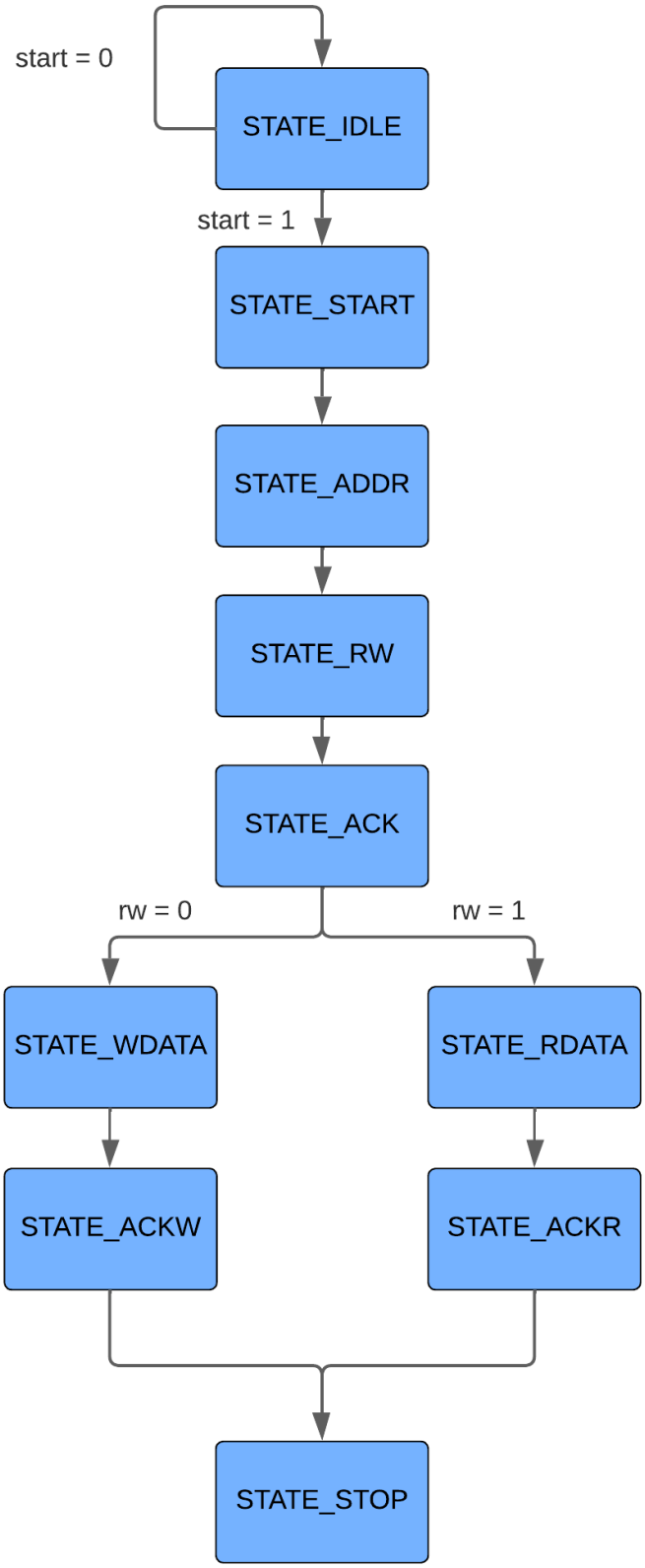
O sinal *addr* contém o valor do endereço de 7 bits (o protocolo permite endereços de 7 ou de 10 bits, nesse caso foi escolhido um enlace com endereços de 7 bits). Ele é enviado de forma serial pelo barramento. Já o sinal *w_addr* contém o dado a ser escrito, caso a operação seja de escrita.

Os sinais *i2c_scl* e *i2c_sda* são os sinais *SCL e SDA* do barramento. Note que *SDA* é definido como *inout* uma vez que pode servir de entrada ou saída para o controlador, devido a bi-direcionalidade do barramento.

3.1.2 Lógica Programacional

Para descrever a lógica de funcionamento e o controle dos sinais transmitidos pra a linha *SDA*, foi utilizada uma máquina de estados, ilustrada pela Figura 9:

Figura 9 – Máquina de Estados: Controlador I^2C



Fonte: Autoria própria.

Cada estado define uma etapa do envio de dados em uma transmissão pelo barramento serial. Com a lógica de máquina de estados, torna-se mais fácil definir a saída *SDA* e obter os dados quando ela é uma entrada do bloco. Para utilizar *i2c_sda* como uma variável *inout*, foi usado um *buffer tri-state* com o sinal de controle *io*, que permite identificar quando é saída e quando é entrada.

O código abaixo mostra o *design* completo, com as lógicas de cada estado:

```
1  `include "uvm_macros.svh"
2  `include "i2c_pkg.svh"
3  `include "i2c_interface.svh"
4
5  module i2c_master (i2c_interface.master vif);
6
7      typedef enum logic [3:0]
8      {
9          STATE_IDLE,       //00
10         STATE_START,      //01
11         STATE_ADDR,       //02
12         STATE_RW,         //03
13         STATE_ACK,        //04
14         STATE_WDATA,      //05
15         STATE_RDATA,
16         STATE_ACK_WDATA,  //06
17         STATE_ACK_RDATA,
18         STATE_STOP        //07
19     } state;
20
21     state current, next;
22
23     always_ff @(posedge vif.clk or posedge vif.reset) begin
24         if(vif.reset) begin
25             current <= STATE_IDLE;
26         end else begin
27             current <= next;
28         end
29     end
30
31     logic [7:0] count;
32     logic i2c_scl_enable;
33
34     logic [6:0] saved_addr;
35     logic [7:0] r_data;
36     logic ack_addr;
37     logic ack_data;
38     logic [7:0] saved_wdata;
```

```
39
40
41      // criar variavel io para definir a dire   o de i2c_sda
42      logic io;   // in (io = 0); out (io = 1);
43      logic sda;
44      logic sda_enable;
45
46      assign vif.i2c_scl = (i2c_scl_enable == 0) ? 1 : ~vif.clk;
47
48      assign vif.i2c_sda = (sda_enable) ? sda : 'bz;
49
50
51      always_comb begin
52          if (vif.reset == 1) begin
53              i2c_scl_enable = 0;
54          end
55          else begin
56              if((current == STATE_IDLE) || (current == STATE_START) || (...
    current == STATE_STOP)) begin
57                  i2c_scl_enable = 0;
58              end
59              else begin
60                  i2c_scl_enable = 1;
61              end
62          end
63      end
64
65
66      always_ff @(posedge vif.clk) begin
67              case (current)
68
69                  STATE_IDLE: begin
70                      if(vif.start) begin
71                          next <= STATE_START;
72                          //io <= 1;
73                      end
74
75                      else next <= STATE_IDLE;
76
77                  end
78
79                  STATE_START: begin
80                      next <= STATE_ADDR;
81                      count <= 6;
82                      //io <= 1;
83
84                  end
```

```
85
86                  STATE_ADDR: begin
87                      if(count == 1) begin
88                          next <= STATE_RW;
89                          //io <= 1;
90                      end
91
92                      if(count != 0) begin
93                          count <= count - 1;
94                      end
95
96                  end
97
98                  STATE_RW: begin
99                      next <= STATE_ACK;
100                     //io <= 1;
101
102                 end
103
104                 STATE_ACK: begin
105
106                     if(!vif.rw) next <= STATE_WDATA;
107                     else next <= STATE_RDATA;
108                     count <= 7;
109                     //io <= 0;
110
111                 end
112
113                 STATE_WDATA: begin
114                     if(count == 1) begin
115                         next <= STATE_ACK_WDATA;
116                     end
117
118                     if(count != 0) begin
119                         count <= count - 1;
120                     end
121
122                 end
123
124                 STATE_RDATA: begin
125                     if(count == 1) begin
126                         next <= STATE_ACK_RDATA;
127                     end
128
129                     if(count != 0) begin
130                         count <= count - 1;
131                     end
```

```
132                 end
133
134                 STATE_ACK_WDATA: begin
135                if(vif.stop) next ≤ STATE_STOP;
136                         //io ≤ 0;
137                 end
138
139                 STATE_ACK_RDATA: begin
140                      next ≤ STATE_STOP;
141                 end
142
143                 STATE_STOP: begin
144                      next ≤ STATE_IDLE;
145                      //io ≤ 1;
146
147                 end
148             endcase
149         end
150
151     always_comb begin
152         case (current)
153             STATE_IDLE: begin
154                 sda_enable = 1;
155                 io = 1;
156                 if(io)  sda = 1;
157                 else    sda = sda;
158             end
159
160             STATE_START: begin
161                 io = 1;
162                 sda_enable = 1;
163     saved_addr = vif.addr;
164                 saved_wdata = vif.w_data;
165                 if(io) sda = 0;
166                 else   sda = sda;
167
168             end
169
170             STATE_ADDR: begin
171                 io = 1;
172                 sda_enable = 1;
173                 if(io) sda = vif.addr[count];
174                 else   sda = sda;
175
176             end
177
178             STATE_RW: begin
```

```
179                 io = 1;
180                 sda_enable = 1;
181                 if(io) sda = vif.rw;
182                 else   sda = sda;
183 
184             end
185 
186             STATE_ACK: begin
187                 io = 0;
188                 sda_enable = 1;
189                 if(!io) begin
190                     ack_addr = vif.i2c_sda;
191         sda = vif.i2c_sda;
192                 end
193 
194                 else begin
195                     sda = sda;
196                 end
197 
198             end
199 
200             STATE_WDATA: begin
201                 io = 1;
202                 sda_enable = 1;
203 
204                 if(io) sda = vif.w_data[count];
205                 else sda = sda;
206 
207             end
208 
209             STATE_RDATA: begin
210                 io = 0;
211                 sda_enable = 1;
212 
213                 if(!io) begin r_data[count] = vif.i2c_sda;
214             sda = vif.i2c_sda;
215     end
216                 else sda = sda;
217             end
218 
219             STATE_ACK_WDATA: begin
220                 io = 0;
221                 sda_enable = 1;
222                 if(!io) begin
223                     ack_data = vif.i2c_sda;
224         sda = vif.i2c_sda;
225                 end
```

```
226
227                 else begin
228                     sda = sda;
229                 end
230
231             end
232
233             STATE_ACK_RDATA: begin
234                 io = 1;
235                 sda_enable = 1;
236
237                 if(io) sda = 1;
238                 else sda = sda;
239             end
240
241             STATE_STOP: begin
242                 io = 1;
243                 sda_enable = 1;
244                 if(io)    sda = 1;
245                 else      sda = sda;
246
247              end
248
249             default: sda = 0;
250         endcase
251
252     end
253
254
255 endmodule
```

Fonte: Autoria própria.

3.1.3 Limitações

As principais limitações do *design* digital referem-se a sua simplicidade e são listadas abaixo:

- Transmissão de um dado por vez no barramento serial.
- Não há tratamento do não reconhecimento de uma operação.
- Não é otimizado para um transmissão ideal de um único byte com todas as informações, uma vez que não possui um gerador de frequência pra linha *SCL*.

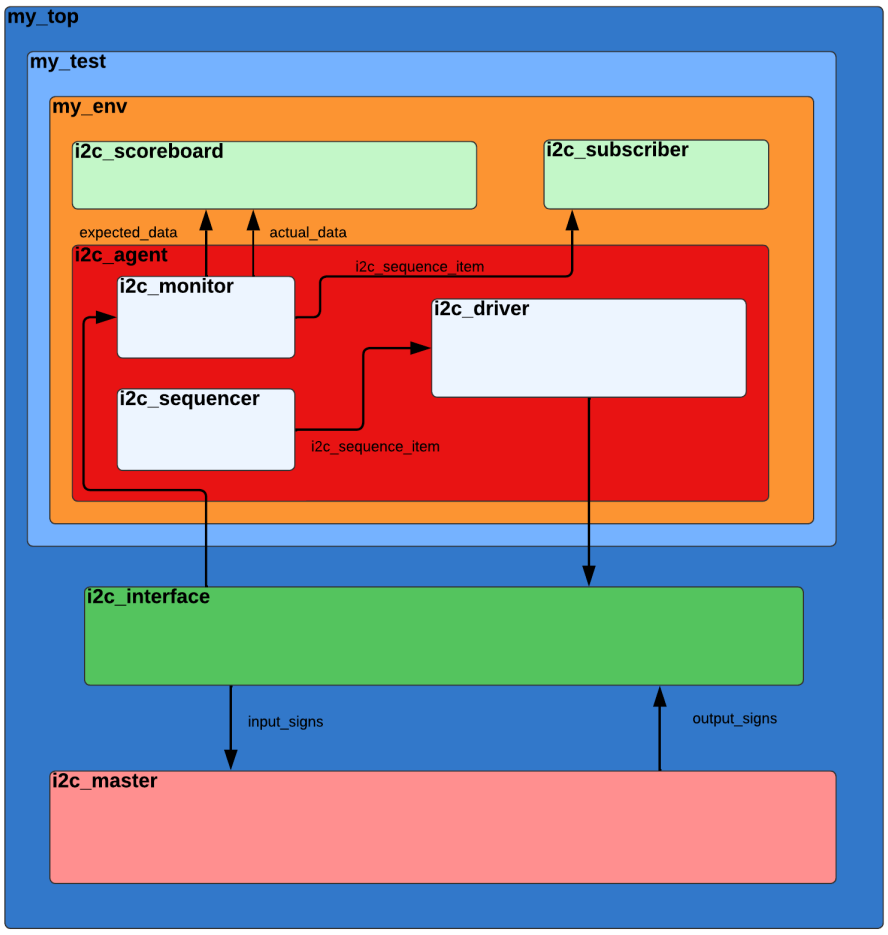
- Não apresenta mais de um modo de funcionamento, apenas o modo normal sincronizado com o *clock* do bloco controlador.
- Não tem definição para arbitrar em um barramento com mais de um alvo.

Essas limitações reiteram o caráter básico e essencial do bloco projetado e abrem novos horizontes para a continuidade da pesquisa, a partir da implementação das mais diversas funcionalidades descritas em (SEMICONDUCTOR, 2021).

3.2 Ambiente UVM de Verificação

O ambiente de verificação funcional feito para testar o controlador I^2C foi pensado a partir da metodologia *UVM*. Cada componente utilizado faz parte da arquitetura geral do ambiente, que foi ilustrada pela Figura 10. Os códigos referentes a cada componente está disposto na seção Apêndice A.

Figura 10 – Arquitetura do ambiente de verificação do controlador I^2C



Fonte: Autoria própria.

3.2.1 Módulo *top*

O módulo *top* é o componente mais externo da arquitetura. Nele são instanciados os módulos de interface e o *DUT*, também é gerado o *clock* e os pacotes são incluídos, para que ele tenha acesso a todos os componentes do ambiente.

3.2.2 Módulo *test*

No módulo *test* são instanciados os objetos *environment* e *sequence* e também é dado início a sequência de transações enviadas para estimular o *DUT*.

3.2.3 Módulo *env*

No módulo *environment* são instanciados os componentes *agent*, *scoreboard* e *subscriber*. Também é onde são conectadas as portas de análise, por onde são enviadas as transações para a checagem do funcionamento do *DUT*.

3.2.4 Módulo *scoreboard*

No módulo *scoreboard* são feitas as checagens do funcionamento do *DUT*. Foram feitos três testes para cobrir o funcionamento do controlador I^2C:

- Teste de condição de início: checa se os valores da linha serial *SDA* no momento de iniciar a transmissão condizem com o especificado na Figura 6.
- Teste de operação de leitura/escrita: checa se os valores recuperados a partir da linha serial *SDA* condizem com os valores aleatórios gerados como entrada do *DUT*. No caso da leitura, checa se o valor de *SDA* é 0 ao final da leitura, uma vez que esse valor simboliza o reconhecimento e o término da transmissão do dado.
- Teste de condição de parada: Teste de condição de parada: checa se os valores da linha serial *SDA* no momento de finalizar a transmissão condizem com o especificado na Figura 6.

Além disso, são gerados arquivos .txt que contém a quantidade de acertos (*matches*) e erros (*mismatches*), podendo inferir a confiabilidade do *design* ao juntar com as métricas de cobertura funcional.

3.2.5 Módulo *subscriber*

No módulo *subscriber* são criados os *coverpoints* para medir a quantidade de valores cobertos no estímulo do *DUT*. Cada um dos *coverpoints* geram *bins* que são contadores

responsáveis por aumentar cada vez que o valor que eles representam for gerado. É utilizada a função *cg_sample*() para a amostragem dos *bins*.

3.2.6 Módulo *agent*

No módulo *agent* são instanciados os componentes *driver*, *sequencer* e *monitor*. É criada uma porta de análise derivada da transação que é conectada ao *monitor* e, por meio dela, são enviadas as transações recuperadas da interface para o *scoreboard*. Além disso, o *sequencer* é conectado ao *driver* e passa a enviar as sequências de transações para ele.

3.2.7 Módulo *driver*

No módulo *driver* são criados uma interface virtual e uma transação. A partir disso são feitas duas tarefas que estimulam o *DUT*: *reset_phase*() e *main_phase*(). Na primeira delas, é feito o *reset* das variáveis da interface. Já na outra, tarefas auxiliares são utilizadas para estimular a interface com valores aleatórios. Uma analogia válida para esse caso é que o *driver* funciona como um alvo funcionaria: estimulando o barramento com respostas às ações do controlador e enviando dados quando necessário.

3.2.8 Módulo *sequencer*

O módulo *sequencer* é responsável por gerar as sequências de transações e enviá-las para o *driver*.

3.2.9 Módulos *sequence* e *sequence_item*

O módulo *sequence_item* contém as variáveis responsáveis por estimular e coletar o comportamento do *DUT*. Ja o módulo *sequence* é responsável por iniciar a transação e randomizá-la.

3.2.10 Módulo *monitor*

O módulo *monitor* tem um papel contrário ao do *driver*. Ele faz o uso de tarefas para coletar os valores da interface e enviá-los para os módulos que executam a checagem do funcionamento.

Uma necessidade primordial para o *monitor* é a sincronização com as operações do *DUT*, para que os sinais recuperados sejam correspondentes às operações a serem checadas. Para isso faz-se uso de controle de eventos temporais, um dos recursos da linguagem *SystemVerilog*.

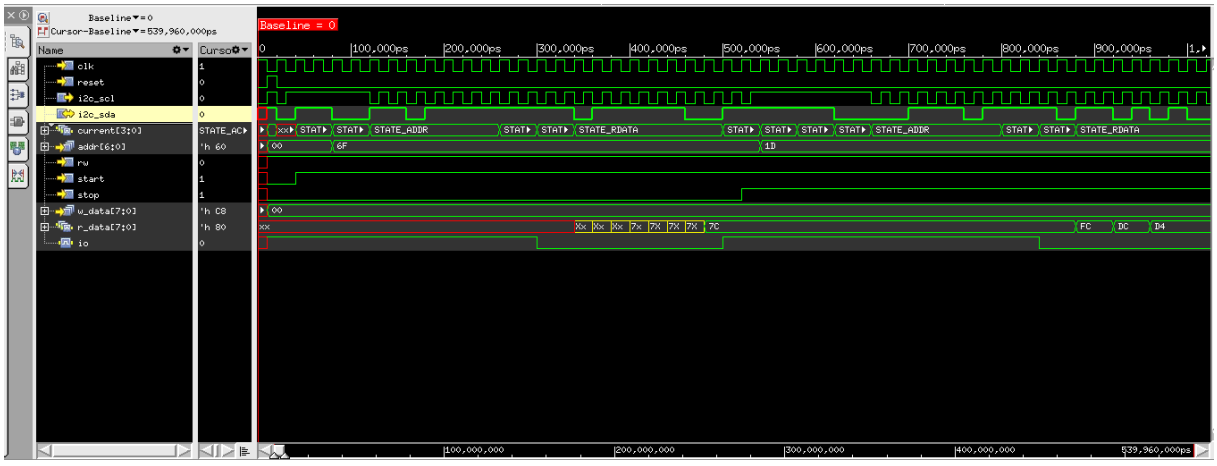
4 Resultados Obtidos

Neste capítulo, serão mostrados e analisados os resultados da simulação *RTL*, a síntese lógica do *design*, o resultado das comparações do *scoreboard* e as métricas de cobertura funcional.

4.1 Simulação *RTL* e Síntese Lógica

Para realizar a simulação *RTL* foi utilizada a ferramenta da Cadence© Xcelium Logic Simulator™ e para o *debug* do funcionamento por meio das formas de onda foi utilizada a ferramenta SimVision Waveform™. Como pode ser visto na Figura 11, a simulação ocorreu sem erros e seu funcionamento ocorre de acordo com o especificado em (SEMICONDUCTOR, 2021). É possível visualizar também o pequeno atraso para a mudança de alguns estados da *FSM* contruída, resultando no problema de sincronismo mencionado na Seção 3.2.10. Isso pode ser resolvido a partir do uso de um gerador de frequência pra linha *SCL* permitindo um maior controle sobre a frequência dos dados no barramento.

Figura 11 – Simulação RTL do controlador I^2C



Fonte: Autoria própria.

Já para a síntese lógica do bloco, foi utilizada a ferramenta Genus Synthesis Solution™ e o *PDK* de $180nm$ da XFAB©. Tanto para a simulação , quanto para a síntese lógica foi utilizado o período de *clock* utilizado foi $20ns$. A Tabela 1 mostra os resultados da síntese como potência, área e *slack*.

Tabela 1 – Relatório de Área e Potência

Área (nm^2)	Área Combinacional	815,360
	Área Sequencial	561,971
	Área Buffer/Inverter	75,264
	Área da Célula	1532,877
	Área dos Fios	1159,907
	Área Total	2692,784
Power (μW)	Potência Interna da Célula	60,505
	Potência Consumida na Troca de Fios	47,218
	Potência Dissipada na Célula	0,022
	Potência Total	107,746
Slack (ps)	Delay do Pior Caminho	158

Fonte: Autoria própria.

4.2 Resultados e Cobertura Funcional

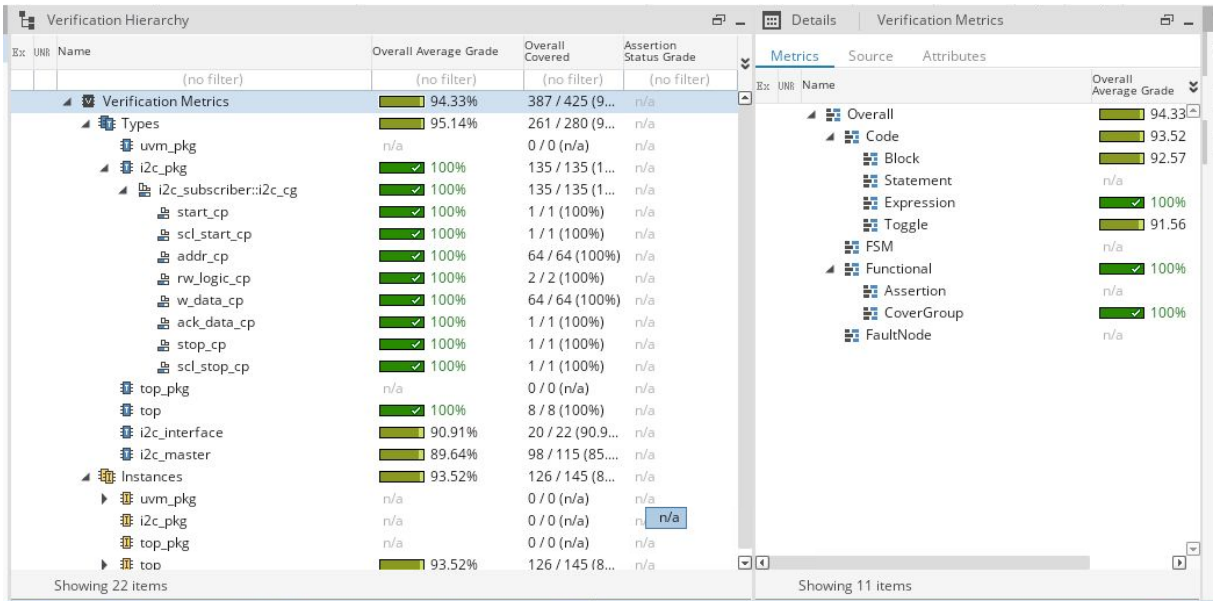
Para realizar a análise dos testes do *scoreboard*, foram gerados arquivos de *report* para os três testes mencionados na Seção 3.2.4. Já para a cobertura funcional, usou-se uma *seed* de valor 125 e foi realizada uma sequência de 1000 transações para estimular o *design*. Para as métricas de cobertura, utilizou-se a ferramenta Integrated Metrics Center™. A Tabela 2 mostra a quantidade de *matches* e *mismatches* de cada teste e a Figura 12 mostra as métricas de cobertura funcional.

Tabela 2 – Tabela de Testes

Testes	Condição de Início	Leitura/Escrita	Condição de Parada
Matches	999	970	998
Mismatches	0	29	1

Fonte: Autoria própria.

Figura 12 – Métricas de Cobertura



Fonte: Autoria própria.

A partir dos dados acima, pode-se concluir que a verificação do bloco foi bem sucedida, atingindo uma métrica de cobertura funcional de 100% e uma porcentagem de *mismatches* de aproximadamente 1%.

5 Conclusões

Esse trabalho teve como objetivo a concepção de um *design digital* de um controlador I^2C, bem como a sua verificação funcional utilizando a metodologia *UVM*.

Foi feito o estudo da especificação do protocolo I^2C e identificado suas funcionalidades básicas e essenciais. A partir disso montou-se um enlace de comunicação básico entre um controlador e um alvo para o teste das funcionalidades tidas como primordiais.

A partir disso, foi feito o *design* digital e desenvolvido um ambiente de verificação que agisse conforme um alvo, a fim de estimular o bloco do controlador. Com os resultados da simulação, foi possível identificar o bom funcionamento do bloco controlador e com as métricas obtidas na verificação funcional, foi constatado o seu funcionamento coerente.

Além disso, foi feita a síntese lógica do bloco, obtendo-se informações relativas a área, potência e *performance* do bloco. Com isso, é dado o primeiro passo para a confecção e projeto de um chip *ASIC*.

Como trabalhos e melhorias futuras, pode-se elencar alguns tópicos:

- Expandir as funcionalidades do *design* digital para lidar com um barramento com mais de um alvo e controlador, criando as lógicas necessária para o controle do barramento.
- Aumentar a quantidade de *features* do barramento, tais como *clock synchronization and stretching*, modos rápido e ultra-rápidos de operação, *10-bit adrressing*, entre outros.
- Resolver os problemas relacionados ao sincronismo de byte implementando um gerador de frequência para linha *SCL* do barramento I^2C.
- Aumentar a quantidade de dados possíveis de serem enviados em uma única transmissão.
- Fazer uma funcionalidade para o controlodar lidar com o não reconhecimento do alvo e criar métodos que garantam a entrega do dado.

Referências Bibliográficas

CHAUHAN, K. Asic design flow in vlsi engineering services (a quick guide). *eInfoChips*, 2020. Citado na página 5.

FATHY, K.; SALAH, K.; GUINDI, R. A proposed methodology to improve uvm-based test generation and coverage closure. In: *2015 10th International Design Test Symposium (IDT)*. [S.l.: s.n.], 2015. p. 147–148. Citado na página 8.

FOSTER, H. 2020 wilson research group functional verification study. In: . [s.n.], 2020. Disponível em: <https://blogs.sw.siemens.com/verificationhorizons/2020/11/05/part-1-the-2020-wilson-research-group-functional-verification-study/>. Citado 2 vezes nas páginas 7 e 9.

HOLLINGWORTH, P. Logic synthesis. In: *[Proceedings] EURO ASIC '90*. [S.l.: s.n.], 1990. p. 250–256. Citado na página 5.

INITIATIVE, A. S. *Universal Verification Methodology (UVM) 1.2 User's Guide*. 2.0. ed. [S.l.], 2015. Disponível em: <https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf>. Citado 3 vezes nas páginas 9, 10 e 12.

KAITH, D.; PATEL, J. B.; GUPTA, N. An introduction to functional verification of i2c protocol using uvm. *International Journal of Computer Applications*, Foundation of Computer Science, v. 121, n. 13, 2015. Citado na página 14.

KUMAR, T. T.; GOPINATH, C. Verification of i2c master core using system verilog uvm. *International Journal of Science and Research (IJSR)*, v. 5, n. 1, p. 641–645, 2016. Citado na página 7.

MARTIN, K. *Digital Integrated Circuit Design*. second. [S.l.]: Oxford University Press, 2002. Citado na página 4.

SALAH, K. A uvm-based smart functional verification platform: Concepts, pros, cons, and opportunities. In: *2014 9th International Design and Test Symposium (IDT)*. [S.l.: s.n.], 2014. p. 94–99. Citado na página 8.

SEMICONDUCTOR, N. *I²C-bus specification and user manual*. 7.0. ed. [S.l.], 2021. Disponível em: <https://www.nxp.com/docs/en/user-guide/UM10204.pdf>. Citado 7 vezes nas páginas 1, 13, 14, 15, 16, 26 e 29.

SUTHERLAND, S.; DAVIDMANN, S.; FLAKE, P. *SystemVerilog for Design: A Guide to using SystemVerilog for Hardware Design and Modeling*. second. [S.l.]: Springer Science Business Media, 2006. Citado na página 6.

VIJAYARAGHAVAN, S.; RAMANATHAN, M. *A practical guide for SystemVerilog assertions*. first. [S.l.]: Springer Science Business Media, 2005. Citado na página 6.

ANEXO A – Códigos do Ambiente de Verificação *UVM* em *SystemVerilog*

Neste apêndice é apresentado os códigos para a implementação dos módulos utilizados no ambiente de verificação *UVM*, em linguagem de descrição de *hardware SystemVerilog*.

- Top:

```
1  `include "i2c_master.sv"
2
3  module top;
4
5      import uvm_pkg::*;
6      import i2c_pkg::*;
7      import top_pkg::*;
8
9      bit clk;
10     bit reset;
11
12     i2c_interface i2c_if(.clk(clk), .reset(reset));
13     i2c_master mst(i2c_if);
14
15      initial begin
16       clk = 1;
17       reset = 0;
18       #10ns reset = 1;
19       #10ns reset = 0;
20     end
21
22     always #10ns clk = !clk;
23
24     initial begin
25         uvm_config_db#(virtual i2c_interface)::set(uvm_root::get(),...
       "*", "vif", i2c_if);
26         run_test("my_test");
27     end
28 endmodule
```

- Top Package e I²C Package:

```
package top_pkg;
    `include "uvm_macros.svh"
    import uvm_pkg::*;
    import i2c_pkg::*;

    `include "../tb/my_env.sv"
    `include "../tb/my_test.sv"

endpackage
```

```
`ifndef I2C_PKG
`define I2C_PKG

`timescale 1ns/1ps

package i2c_pkg;
    `include "uvm_macros.svh"
    import uvm_pkg::*;

    `include "../src/i2c_sequence_item.svh"
    `include "../src/i2c_sequence.svh"
    `include "../src/i2c_sequencer.svh"
    `include "../src/i2c_driver.svh"
    `include "../src/i2c_monitor.svh"
    `include "../src/i2c_agent.svh"
    `include "../src/i2c_sb_subscriber.svh"
    `include "../src/i2c_scoreboard.svh"
    `include "../src/i2c_subscriber.svh"

endpackage

`endif//I2C_PKG
```

- Test:

```
class my_test extends uvm_test;
    `uvm_component_utils(my_test)

    i2c_sequence seq;
    my_env env;
    //int cycles = 10;

    function new(string name, uvm_component parent = null);
        super.new(name, parent);
```

```
10      endfunction
11
12      function void build_phase(uvm_phase phase);
13          super.build_phase(phase);
14          seq = i2c_sequence::type_id::create("seq", this);
15          env = my_env::type_id::create("env", this);
16      endfunction
17
18      task main_phase(uvm_phase phase);
19          phase.raise_objection(this);
20          seq.start(env.i2c_agt.sqr);
21          phase.drop_objection(this);
22      endtask
23
24  endclass
```

- Environment:

```
1   class my_env extends uvm_env;
2       `uvm_component_utils(my_env)
3
4       i2c_agent       i2c_agt;
5       i2c_subscriber  i2c_sub;
6       i2c_scoreboard  i2c_sb;
7
8
9       function new(string name, uvm_component parent = null);
10          super.new(name, parent);
11      endfunction
12
13      virtual function void build_phase(uvm_phase phase);
14          super.build_phase(phase);
15          i2c_agt = i2c_agent::type_id::create("i2c_agt", this);
16          i2c_sub = i2c_subscriber::type_id::create("i2c_sub", this);
17          i2c_sb  = i2c_scoreboard::type_id::create("i2c_sb", this);
18      endfunction
19
20      virtual function void connect_phase(uvm_phase phase);
21          super.connect_phase(phase);
22          i2c_agt.mon_ap.connect(i2c_sub.analysis_export);
23          i2c_agt.mon_ap.connect(i2c_sb.i2c_analysis_export);
24      endfunction
25  endclass
```

- Scoreboard:

```
class i2c_scoreboard extends uvm_scoreboard;
   `uvm_component_utils(i2c_scoreboard)

   uvm_analysis_export#(i2c_sequence_item) i2c_analysis_export;
   local i2c_sb_subscriber i2c_sb_sub;
  int count_match_start;
  int count_mismatch_start;
  int count_match_rw;
  int count_mismatch_rw;
  int count_match_stop;
  int count_mismatch_stop;

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction: new

   function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      i2c_analysis_export = new( .name("i2c_analysis_export"), ....
    parent(this));
      i2c_sb_sub = i2c_sb_subscriber::type_id::create(.name("...
   i2c_sb_sub"), .parent(this));
      count_match_start = 0;
      count_mismatch_start = 0;
      count_match_rw = 0;
      count_mismatch_rw = 0;
      count_match_stop = 0;
      count_mismatch_stop = 0;
   endfunction: build_phase

   function void connect_phase(uvm_phase phase);
      super.connect_phase(phase);
      i2c_analysis_export.connect(i2c_sb_sub.analysis_export);
   endfunction: connect_phase


   virtual function void check_i2c_start(i2c_sequence_item i2c_tx);

      uvm_table_printer p = new;
      int f_start = 0;
      f_start = $fopen("./start_condition.txt", "a");

      if (i2c_tx.start == 0 && i2c_tx.scl_start == 1) begin
         `uvm_info("i2c_scoreboard",
                   { "Start condition passed.\n", i2c_tx.sprint...
   (p) }, UVM_LOW);
    count_match_start++;
```

```
45          if (f_start) begin
46             $fwrite(f_start, "i2c_scoreboard \n Start condition ...
    passed.\n");
47             $fwrite(f_start, "Matches: %d", count_match_start);
48          end else begin
49             `uvm_error("i2c_scoreboard", {"Failed to open file: ...
    start_condition.txt"});
50          end
51       end else begin
52         `uvm_error("i2c_scoreboard",
53                         { "Start condition failed.\n", i2c_tx....
    sprint(p) });
54       count_mismatch_start++;
55          if (f_start) begin
56             $fwrite(f_start, "i2c_scoreboard \n Start condition ...
    failed.\n");
57             $fwrite(f_start, "Mismatches: %d", count_mismatch_start...
    );
58          end else begin
59             `uvm_error("i2c_scoreboard", {"Failed to open file: ...
    start_condition.txt"});
60          end
61       end
62
63       close_file(f_start);
64
65    endfunction: check_i2c_start
66
67
68
69    virtual function void check_i2c_rw(i2c_sequence_item i2c_tx);
70       int count = 0;
71      int f_rw = 0;
72       uvm_table_printer p = new;
73      f_rw = $fopen("./rw_check.txt", "a");
74
75       if (i2c_tx.rw_logic == 1) begin
76         if(i2c_tx.ack_data == 1) begin
77             `uvm_info("i2c_scoreboard",
78                         { "Read operation passed.\n", i2c_tx.sprint(...
    p) }, UVM_LOW);
79       count_match_rw++;
80           if (f_rw) begin
81              $fwrite(f_rw, "i2c_scoreboard \n Read operation ...
    passed.\n");
82             $fwrite(f_rw, "Matches: %d", count_match_rw);
83          end else begin
```

```
84                `uvm_error("i2c_scoreboard", {"Failed to open file:...
    rw_check.txt"});
85              end
86            end else begin
87                `uvm_error("i2c_scoreboard",
88                           { "Read operation failed.\n", i2c_tx.sprint...
    (p) });
89          count_mismatch_rw++;
90              if (f_rw) begin
91                 $fwrite(f_rw, "i2c_scoreboard \n Read operation ...
    failed.\n");
92                $fwrite(f_rw, "Mismatches: %d", count_mismatch_rw);
93              end else begin
94                `uvm_error("i2c_scoreboard", {"Failed to open file:...
    rw_check.txt"});
95               end
96            end
97          end else begin
98            foreach(i2c_tx.w_data[i])begin
99                if(i2c_tx.recovery_w_data[i] == i2c_tx.w_data[i]) count...
    ++;
100           end
101           if(count == 8) begin
102               `uvm_info("i2c_scoreboard",
103                          { "Write operation passed.\n", i2c_tx.sprint...
    (p) }, UVM_LOW);
104         count_match_rw++;
105               if (f_rw) begin
106                 $fwrite(f_rw, "i2c_scoreboard \n Write operation ...
    passed.\n");
107               $fwrite(f_rw, "Matches: %d", count_match_rw);
108             end else begin
109               `uvm_error("i2c_scoreboard", {"Failed to open file:...
    rw_check.txt"});
110             end
111           end else begin
112               `uvm_error("i2c_scoreboard",
113                          { "Write operation failed.\n", i2c_tx....
    sprint(p) });
114         count_mismatch_rw++;
115               if (f_rw) begin
116                 $fwrite(f_rw, "i2c_scoreboard \n Write operation ...
    failed.\n");
117               $fwrite(f_rw, "Mismatches: %d", count_mismatch_rw);
118             end else begin
119               `uvm_error("i2c_scoreboard", {"Failed to open file:...
    rw_check.txt"});
```

```
120             end
121           end
122         end
123
124         close_file(f_rw);
125
126     endfunction: check_i2c_rw
127
128     virtual function void check_i2c_stop(i2c_sequence_item i2c_tx);
129         uvm_table_printer p = new;
130       int f_stop = 0;
131         f_stop = $fopen("./stop_condition.txt", "a");
132
133         if (i2c_tx.stop == 1 && i2c_tx.scl_stop == 1) begin
134           `uvm_info("i2c_scoreboard",
135                       { "Stop condition passed.\n", i2c_tx.sprint(...
    p) }, UVM_LOW);
136     count_match_stop++;
137           if (f_stop) begin
138             $fwrite(f_stop, "i2c_scoreboard \n Stop condition ...
    passed.\n");
139             $fwrite(f_stop, "Matches: %d", count_match_stop);
140           end else begin
141             `uvm_error("i2c_scoreboard", {"Failed to open file: ...
    stop_condition.txt"});
142           end
143         end else begin
144           `uvm_error("i2c_scoreboard",
145                       { "Stop condition failed.\n", i2c_tx.sprint...
    (p) });
146         count_mismatch_stop++;
147           if (f_stop) begin
148             $fwrite(f_stop, "i2c_scoreboard \n Stop condition ...
    failed.\n");
149             $fwrite(f_stop, "Mismatches: %d", count_mismatch_stop);
150           end else begin
151             `uvm_error("i2c_scoreboard", {"Failed to open file: ...
    stop_condition.txt"});
152           end
153         end
154
155         close_file(f_stop);
156
157     endfunction: check_i2c_stop
158
159
160 // Fun  o para fechar o arquivo
```

```
161     function void close_file(int f);
162         if (f) begin
163             $fclose(f);
164             f = 0;
165         end
166     endfunction : close_file
167 
168 
169 endclass: i2c_scoreboard
```

- Subscriber:

```
1  class i2c_subscriber extends uvm_subscriber#(i2c_sequence_item);
2      `uvm_component_utils(i2c_subscriber)
3  
4      i2c_sequence_item tr;
5  
6      covergroup i2c_cg;
7          start_cp:          coverpoint tr.start {bins str_null = {0};}
8          scl_start_cp:      coverpoint tr.scl_start {bins str_scl_true ...
       = {1};}
9          addr_cp:           coverpoint tr.addr;
10         rw_logic_cp:       coverpoint tr.rw_logic;
11         w_data_cp:         coverpoint tr.w_data;
12         ack_data_cp:       coverpoint tr.ack_data {bins ack_true = ...
       {1};}
13         stop_cp:           coverpoint tr.stop {bins stp_true = {1};}
14         scl_stop_cp:       coverpoint tr.scl_start {bins stp_scl_true ...
       = {1};}
15     endgroup: i2c_cg
16 
17     function new(string name, uvm_component parent);
18         super.new(name, parent);
19         i2c_cg = new;
20     endfunction: new
21 
22     function void write(i2c_sequence_item t);
23         tr = t;
24         i2c_cg.sample();
25     endfunction: write
26 endclass: i2c_subscriber
```

```
1  typedef class i2c_scoreboard;
2  
3  class i2c_sb_subscriber extends uvm_subscriber#(i2c_sequence_item);
```

```
4     `uvm_component_utils(i2c_sb_subscriber)
5
6     function new(string name, uvm_component parent);
7        super.new(name, parent);
8     endfunction: new
9
10    function void write(i2c_sequence_item t);
11       i2c_scoreboard i2c_sb;
12
13       $cast( i2c_sb, m_parent );
14       i2c_sb.check_i2c_start(t);
15       i2c_sb.check_i2c_rw(t);
16       i2c_sb.check_i2c_stop(t);
17    endfunction: write
18 endclass: i2c_sb_subscriber
```

- Agent:

```
1  `ifndef I2C_AGENT
2  `define I2C_AGENT
3
4  class i2c_agent extends uvm_agent;
5      `uvm_component_utils(i2c_agent)
6
7      uvm_analysis_port #(i2c_sequence_item) mon_ap;
8
9       typedef uvm_sequencer#(i2c_sequence_item) sequencer;
10      sequencer sqr;
11      //i2c_agent_config cfg;
12      i2c_monitor mon;
13      i2c_driver drv;
14
15     function new(string name, uvm_component parent);
16         super.new(name, parent);
17     endfunction
18
19     function void build_phase(uvm_phase phase);
20         super.build_phase(phase);
21
22         mon_ap = new(.name("mon_ap"), .parent(this));
23         sqr = sequencer::type_id::create("sqr",this);
24         drv = i2c_driver::type_id::create("drv",this);
25         mon = i2c_monitor::type_id::create("mon",this);
26     endfunction
27
28     function void connect_phase(uvm_phase phase);
```

```
29          super.connect_phase(phase);
30          drv.seq_item_port.connect(sqr.seq_item_export);
31          mon.mon_ap.connect(mon_ap);
32      endfunction
33
34  endclass
35  `endif //I2C_AGENT
```

- Sequencer:

```
1  class i2c_sequencer extends uvm_sequencer #(i2c_sequence_item);
2      `uvm_component_utils(i2c_sequencer)
3
4      //calling and allocating the sequencer
5      function new (string name = "i2c_sequencer", uvm_component ...
       parent = null);
6          super.new(name, parent);
7      endfunction
8  endclass: i2c_sequencer
```

- Sequence:

```
1  /*class write_op extends i2c_sequence_item;
2
3  endclass*/
4
5  class i2c_sequence extends uvm_sequence #(i2c_sequence_item);
6    `uvm_object_utils(i2c_sequence)
7
8     i2c_sequence_item tr;
9
10   function new(string name="i2c_sequence");
11     super.new(name);
12   endfunction: new
13
14   task body;
15     repeat(1000) begin
16       tr = i2c_sequence_item::type_id::create("tr");
17       start_item(tr);
18       assert(tr.randomize());
19       finish_item(tr);
20     end
21   endtask: body
22 endclass: i2c_sequence
```

- Sequence Item:

```
class i2c_sequence_item  extends uvm_sequence_item;

  bit start;
  bit stop;
  rand logic [6:0] addr;
  rand logic [7:0] w_data;
  logic [7:0] recovery_w_data;
  rand logic [7:0] r_data;
  bit ack_addr;
  bit ack_data;
  rand bit rw_logic;
  bit scl_start;
  bit scl_stop;

  function new(string name = "i2c_sequence_item");
    super.new(name);
  endfunction

  `uvm_object_param_utils_begin(i2c_sequence_item)
    `uvm_field_int(start, UVM_ALL_ON)
    `uvm_field_int(stop, UVM_ALL_ON)
    `uvm_field_int(addr, UVM_ALL_ON)
    `uvm_field_int(w_data, UVM_ALL_ON)
    `uvm_field_int(recovery_w_data, UVM_ALL_ON)
    `uvm_field_int(r_data, UVM_ALL_ON)
    `uvm_field_int(ack_addr, UVM_ALL_ON)
    `uvm_field_int(ack_data, UVM_ALL_ON)
    `uvm_field_int(rw_logic, UVM_ALL_ON)
  `uvm_object_utils_end

   virtual function void do_print(uvm_printer printer);
    super.do_print(printer);
    //Print fields specific to this sequence item
    printer.print_field("start", start, $bits(start), UVM_DEC);
    printer.print_field("stop", stop, $bits(stop), UVM_DEC);
    printer.print_field("addr", addr, $bits(addr), UVM_BIN);
    printer.print_field("w_data", w_data, $bits(w_data), UVM_BIN);
    printer.print_field("recovery_w_data", recovery_w_data, $bits(...
   recovery_w_data), UVM_BIN);
    printer.print_field("r_data", r_data, $bits(r_data), UVM_BIN);
    printer.print_field("ack_addr", ack_addr, $bits(ack_addr), ...
   UVM_BIN);
    printer.print_field("ack_data", ack_data, $bits(ack_data), ...
   UVM_BIN);
```

```
42      printer.print_field("rw_logic", rw_logic, $bits(rw_logic), ...
        UVM_BIN);
43    endfunction
44
45  endclass
```

- Driver:

```
1  /*
2   * Parameters:
3   *
4   *  ADDR_WIDTH: Sets I2C protocol address size, either 7 or 10 bits...
        ; This
5   *  version supports only 7 (DEFAULT=7)
6   *
7   *  DATA_WIDTH: Sets word data word size to be driven by byte. ...
        Should be byte
8   *  multiples, i.e. 8/16/24/32/... ; This version supports only 8 (...
        DEFAULT=8)
9   *
10  *  cfg.initial_delay_clock: Sets the number of interface clock ...
        periods before
11  *  driving a transaction (DEFAULT=0)
12  *
13  *  cfg.start_condition_post_delay: Sets the number of interface ...
        clock periods before
14  *  driving a transaction (DEFAULT=0)
15  *
16  *  cfg.clk_divider: Sets SCL clock period based on interface clock...
        (DEFAULT=2)
17  *
18  */
19 class i2c_driver extends uvm_driver #(i2c_sequence_item);
20   `uvm_component_utils(i2c_driver)
21
22   virtual i2c_interface vif;
23   //i2c_agent_config cfg;
24   i2c_sequence_item tr;
25   //bit stop_scl;
26
27   function new(string name = "i2c_driver", uvm_component parent = ...
       null);
28       super.new(name, parent);
29   endfunction
30
31   virtual function void build_phase(uvm_phase phase);
```

```
32        super.build_phase(phase);
33        if(!uvm_config_db#(virtual i2c_interface)::get(this, "", "vif...
    ", vif)) begin
34        `uvm_fatal("NOVIF", "failed to get virtual interface")
35        end
36    endfunction : build_phase
37
38    task reset_phase(uvm_phase phase);
39        phase.raise_objection(this);
40        @(posedge vif.reset);
41
42        //Reset write data channel
43        vif.start  <= '0;
44        vif.stop   <= '0;
45        vif.rw   <= '1;
46      vif.addr <= '0;
47      vif.w_data <= '0;
48
49        //tr = null;
50        @(negedge vif.reset);
51        phase.drop_objection(this);
52      endtask : reset_phase
53
54    task main_phase (uvm_phase phase);
55      forever begin
56        //repeat (cfg.initial_delay_clock) @(posedge vif.clk);
57        seq_item_port.get_next_item(tr);
58        //repeat(4) @(posedge vif.clk);
59        drive_start();
60        if(!vif.i2c_sda) @(negedge vif.i2c_sda);
61
62        fork
63          drive_addr();
64          drive_rw();
65          drive_addr_acknowledge();
66        join
67
68        //@(negedge vif.i2c_scl);
69
70        if(!vif.rw) begin
71          @(negedge vif.i2c_scl);
72
73          drive_write_data();
74          drive_data_acknowledge();
75
76          //@(negedge vif.i2c_scl);
77        end
```

```
78        else begin
79          repeat(2) @(negedge vif.i2c_scl);
80          drive_read_data();
81          repeat(2) @(negedge vif.i2c_scl);
82        end
83
84        //repeat(3) @(posedge vif.clk);
85
86        drive_stop();
87        //if(vif.i2c_sda) @(posedge vif.i2c_sda);
88
89        seq_item_port.item_done();
90      end
91    endtask
92
93    virtual task drive_start();
94      @(posedge vif.clk);
95      vif.start <= 1;
96      vif.rw <= 1;
97      //$display("DRIVER: START STATE");
98    endtask
99
100   virtual task drive_addr();
101     foreach (tr.addr[i]) begin
102       vif.addr[6-i] <= tr.addr[6-i];
103     end
104     //$display("DRIVER: ADDR STATE");
105   endtask
106
107   virtual task drive_write_data();
108     foreach (tr.w_data[i])begin
109       vif.w_data[7-i] <= tr.w_data[7-i];
110     end
111     //$display("DRIVER: DATA STATE");
112   endtask
113
114   virtual task drive_read_data();
115     for (int i = 0; i<8; i++) begin
116       @(negedge vif.i2c_scl);
117       vif.i2c_sda <= tr.r_data[7-i];
118     end
119   endtask
120
121   virtual task drive_rw();
122     repeat(8) @(negedge vif.i2c_scl);
123     vif.rw <= tr.rw_logic;
124     //$display("DRIVER: RW STATE");
```

```
125   endtask
126
127   virtual task drive_addr_acknowledge();
128     repeat(9) @(negedge vif.i2c_scl);
129     vif.i2c_sda <= 1;
130     //$display("DRIVER: ADDR_ACK STATE");
131   endtask
132
133   virtual task drive_data_acknowledge();
134     repeat(10) @(negedge vif.i2c_scl);
135     vif.i2c_sda <= 0;
136     //$display("DRIVER: DATA_ACK STATE");
137   endtask
138
139   virtual task drive_stop();
140     vif.stop <=1;
141     //$display("DRIVER: STOP STATE");
142   endtask
143
144
145 endclass
```

- Monitor:

```
1  class i2c_monitor extends uvm_monitor;
2      `uvm_component_utils(i2c_monitor)
3
4      virtual i2c_interface vif;
5      i2c_sequence_item tr;
6      //i2c_agent_config cfg;
7
8      uvm_analysis_port #(i2c_sequence_item) mon_ap;
9
10     function new(string name, uvm_component parent);
11         super.new(name, parent);
12         mon_ap = new ("mon_ap", this);
13     endfunction
14
15     virtual function void build_phase(uvm_phase phase);
16         super.build_phase(phase);
17         tr  = i2c_sequence_item::type_id::create("tr", this);
18         if(!uvm_config_db#(virtual i2c_interface)::get(this, "","...
       vif", vif))
19     begin
20       `uvm_fatal("NOVIF", "The virtual connection wasn't succesful!...
       ");
```

```
21      end
22      endfunction
23
24      task main_phase(uvm_phase phase);
25    forever begin
26      //super.run_phase(phase);
27      repeat(4) @(posedge vif.clk);
28      collect_start();
29      //if(!vif.i2c_sda) @(negedge vif.i2c_sda);
30
31          fork
32            collect_addr();
33            collect_rw();
34            collect_addr_acknowledge();
35          join
36
37          //@(negedge vif.i2c_scl);
38
39          if(!vif.rw) begin
40        repeat(3) @(negedge vif.i2c_scl);
41
42              collect_write_data();
43              //collect_data_acknowledge();
44          end
45          else begin
46            @(negedge vif.i2c_scl);
47                      //collect_read_data();
48                      collect_data_acknowledge();
49
50          end
51
52          repeat(3) @(posedge vif.clk);
53
54          collect_stop();
55          //if(vif.i2c_sda) @(posedge vif.i2c_sda);
56
57              mon_ap.write(tr);
58    end
59      endtask
60
61      virtual task collect_start();
62
63              @(negedge vif.clk);
64          tr.start <= vif.i2c_sda;
65              tr.scl_start <= vif.i2c_scl;
66      //$display("MONITOR: START STATE");
67
```

```
68    endtask
69
70    virtual task collect_addr();
71
72              foreach (tr.addr[i]) begin
73                  @(posedge vif.i2c_scl);
74            tr.addr[i] <= vif.i2c_sda;
75          end
76      //$display("MONITOR: ADDR STATE");
77
78    endtask
79
80    virtual task collect_write_data();
81
82        foreach (tr.recovery_w_data[i])begin
83                  @(posedge vif.i2c_scl);
84            tr.recovery_w_data[i] <= vif.i2c_sda;
85          tr.w_data[i] <= vif.w_data[i];
86          end
87      //$display("MONITOR: WRITE DATA STATE");
88
89    endtask
90
91    /*virtual task collect_read_data();
92
93      foreach (tr.r_data[i])begin
94        @(negedge vif.i2c_scl);
95        tr.r_data[7-i] <= vif.i2c_sda;
96      end
97      $display("MONITOR: READ DATA STATE");
98
99    endtask*/
100
101   virtual task collect_rw();
102
103             repeat(8) @(posedge vif.i2c_scl);
104         tr.rw_logic <= vif.i2c_sda;
105     //$display("MONITOR: RW STATE");
106
107   endtask
108
109   virtual task collect_addr_acknowledge();
110
111     repeat(9) @(posedge vif.i2c_scl);
112     tr.ack_addr <= vif.i2c_sda;
113     //$display("MONITOR: ADDR_ACK STATE");
114
```

```
115   endtask
116
117   virtual task collect_data_acknowledge();
118
119     repeat(11) @(posedge vif.i2c_scl);
120     tr.ack_data <= vif.i2c_sda;
121     //$display("MONITOR: DATA_ACK STATE");
122
123   endtask
124
125   virtual task collect_stop();
126
127     @(negedge vif.clk);
128     tr.stop <= vif.i2c_sda;
129     tr.scl_stop <= vif.i2c_scl;
130     //$display("MONITOR: STOP STATE");
131
132   endtask
133
134 endclass
```