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

Trabalho de Conclusão de Curso

IMPLEMENTAÇÃO DE UM BARRAMENTO *AXI4-LITE* PARA CIRCUITOS INTEGRADOS

Ezequiel Apolonio Brito de Araújo

Campina Grande - PB

Maio de 2024

Ezequiel Apolonio Brito de Araújo

IMPLEMENTAÇÃO DE UM BARRAMENTO *AXI4-LITE* PARA CIRCUITOS INTEGRADOS

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

Maio de 2024

Ezequiel Apolonio Brito de Araújo

IMPLEMENTAÇÃO DE UM BARRAMENTO *AXI4-LITE* PARA CIRCUITOS INTEGRADOS

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
Maio de 2024

Minha sincera dedicação vai para Deus e minha amada família, com menção especial à minha mãe Maria, ao padrasto Olacildo, ao avô Cosme e à avó Nazaré. O apoio inabalável, o amor ilimitado e o incentivo constante foram fundamentais em minha jornada. Sem dúvida, eles são a força motriz das minhas realizações

Agradecimentos

Em minha jornada, expressarei para sempre gratidão a Deus que é a fonte de toda a existência, aquele que criou todas as coisas. E por amor ilimitado pela humanidade, deu o seu único Filho para que todos nele crer não pereça, mas tenha a vida eterna. Além disso, ele orquestrou eventos notáveis neste reino terrestre. Quer os dias tenham sido repletos de triunfo ou de desespero, esta presença divina permaneceu ao meu lado.

Devo muita gratidão à minha família, especialmente à minha mãe Maria, ao padrasto Olacildo, aos avós e aos irmãos, com quem compartilhei a maior parte da minha vida.

Aos meus colegas de trabalho da Redepharma, que nos últimos meses estiveram comigo, me ajudando não só no profissional, mas me moldando como pessoa. Em particular a Erika Tarsila, Camila, Juliana, Emmanuella, Marília, Robério e Allen, que além de me ensinarem os primeiros passos no mercado de trabalho, me deram total apoio a seguir em frente, em busca dos meus sonhos.

A todos membros da Igreja Assembleia de Deus em Campina Grande que durante todo esse tempo até aqui tens sido de relevância imensurável para minha vida. Sou especialmente grato as irmãs Andrade, à minha família EJC e a casa da tia Rô, com quem Deus me abençoou de maneira extraordinária. A Ruan e sua esposa Michelle que sempre estiveram me aconselhando e compartilhando momentos incríveis comigo. E ao Pr Daniel Carlos que tanto me ensinou sobre Deus e me motivou a continuar estudando.

Quero expressar também minha gratidão aos amigos e companheiros que estiveram ao meu lado, durante todos esses anos. Uma menção especial para Alan Pessoa, Gustavo Vilar, Heriberto Gomes e Hélder Guimarães, que não só partilharam comigo os seus anos de aprendizagem e preparação, mas também os seus desafios e obstáculos, principalmente na reta final. Acalento a esperança de que nossos caminhos continuem a se entrelaçar à medida que embarcamos na jornada da vida, criando inúmeras memórias juntos.

Sou imensamente grato aos meus professores, em especial a Gutemberg Júnior, pelo apoio inabalável sabendo dos meus desafios, me concedeu a oportunidade de ser seu orientando.

Finalmente, grato a todos que cruzaram minha rota nessa graduação e que tive o privilégio de apoiar e a honra de ser apoiado.

"Tu és o meu abrigo; tu me preservarás das angústias e me cercarás de canções de livramento."

Salmos 32:7

Resumo

O protocolo *AXI4-Lite* é uma arquitetura de barramento desenvolvida pela *ARM Holdings*, projetada para simplificar a comunicação eficiente entre um processador *manager* (como um processador *ARM*) e dispositivos *subordinates* em sistemas digitais. Este trabalho oferece uma visão geral dos princípios fundamentais do *AXI4-Lite*, seus sinais, operações e sua aplicabilidade em diferentes contextos de design. Foi implementado um barramento *AXI4-Lite* de alto desempenho utilizando *SystemVerilog*. Todo esse projeto está dividido em duas partes: a primeira é o design *manager* e *subordinate* do AXI-Lite. A segunda parte foi um testbench para avaliar os componentes de projeto *manager* e *subordinate*. Este trabalho serve como uma introdução acessível ao protocolo AXI4-Lite, fornecendo uma base sólida para estudos futuros e implementações práticas em projetos de sistemas digitais. Fazendo uso das ferramentas da Cadence©.

Palavras-chave: *AXI4-Lite*, barramento, comunicação *manager-subordinate*, *SystemVerilog*.

Abstract

The AXI4-Lite protocol is a bus architecture developed by ARM Holdings designed to simplify efficient communication between a manager processor (such as an ARM processor) and subordinate devices in digital systems. This work provides an overview of the fundamental principles of AXI4-Lite, its signals, operations and its applicability in different design contexts. A high-performance AXI4-Lite bus was implemented using SystemVerilog. This entire project is divided into two parts: the first is the manager and subordinate design of AXI-Lite. The second part was a testbench to evaluate the manager and subordinate project components. This work serves as an accessible introduction to the AXI4-Lite protocol, providing a solid foundation for future studies and practical implementations in digital systems projects. Making use of Cadence © tools.

Keywords: AXI4-Lite, bus, manager-subordinate communication, SystemVerilog

Lista de ilustrações

Lista de abreviaturas e siglas

ABNT	Associação Brasileira de Normas Técnicas
ASIC	*Application-Specific Integrated Circuit*
FPGA	*Field-Programmable Gate Array*
AMBA	*Advanced Microcontroller Bus Architecture*
OOP	*Object Orientated Programming*
UVM	*Universal Verification Methodology*
IEEE	*Institute of Electrical and Electronics Engineers*
AXI	*Advanced eXtensible Interface*
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

A contínua evolução tecnológica tem impulsionado a complexidade dos sistemas embarcados, desde dispositivos móveis até sistemas de controle industrial. Nesse contexto, a comunicação eficiente entre os componentes desses sistemas torna-se crucial para garantir desempenho, confiabilidade e flexibilidade. O barramento *AXI4 (Advanced Extensible Interface) Lite*, uma versão simplificada do protocolo *AXI4*, surge como uma solução chave.

Nos sistemas digitais, a comunicação entre os diferentes blocos de hardware é crucial para o funcionamento correto e eficiente do sistema como um todo. Os barramentos desempenham um papel fundamental nessa comunicação, permitindo a transferência de dados entre os diversos componentes de um sistema integrado. Nesse contexto, o barramento *AXI4 Lite*, uma versão simplificada do protocolo *AXI4* desenvolvido pela ARM®, destaca-se como uma solução robusta e eficiente para interligação de IPs em circuitos integrados.

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

1.1 Objetivos Gerais

O objetivo geral do trabalho é sugerir e implementar um *design* digital de protocolo de comunicação *AXI4 Lite* e aplicá-lo em 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:

- Colocar em prática as metodologias de desenvolvimento de circuitos digitais, bem como a prática da linguagem de descrição de *hardware SystemVerilog*
- Aprender o protocolo de comunicação *AXI4 Lite* e suas especificações
- Analisar os diferentes métodos existentes na metodologia

- Desenvolver um *testbench* capaz de estimular das mais diversas formas o *design* e cobrir o máximo de testes possíveis para a validação.
- Fazer a verificação utilizando *UVM* desenvolvido anteriormente.
- Fixar 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 conteúdo introdutório, conforme a seguir.

No **Capítulo 2** é apresentado a fundamentação teórica, expondo os conceitos básicos de desenvolvimento de circuitos digitais bem como as especificações do *AXI4 Lite*.

O **Capítulo 3** aborda a metodologia utilizada para o desenvolvimento do *design* digital, assim também 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.

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 *AXI4 Lite*.

2.1 *Design* de Circuitos Digitais

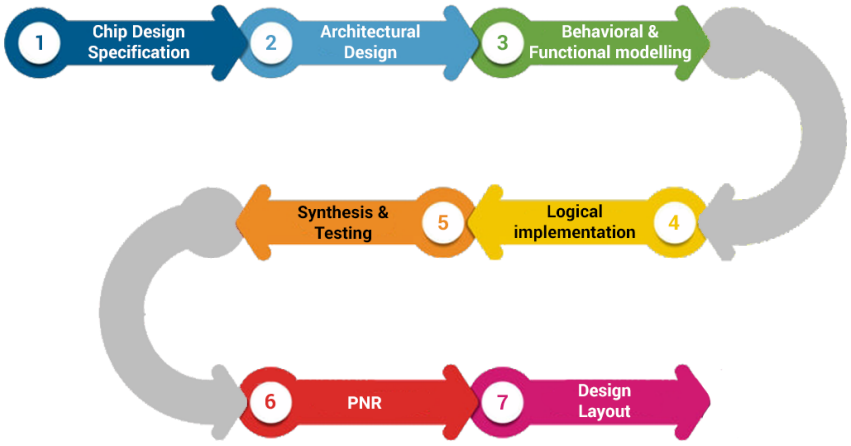
O *design* de circuitos digitais é um campo especializado preocupado com o conceito, projeto e implementação de circuitos digitais de integração extremamente baixa, como chips e sistemas em um chip (SoC). Esses circuitos digitais são úteis para operação de dispositivos eletrônicos modernos, como computadores, *smartphones* e sistemas embarcados. Eles envolvem a integração de componentes eletrônicos como transistores, resistores, capacitores, etc. em um único chip.

Ao projetar circuitos digitais microeletrônicos, alguns dos principais desafios incluem aumentar a densidade de integração, reduzir o consumo de energia, aumentar a velocidade operacional e garantir confiabilidade e robustez.

Uma das áreas mais abrangentes de desenvolvimento de circuitos digitais hoje é a do *design* ASIC. Este é um campo especializado focado na criação de chips personalizados para aplicações específicas. Ao contrário dos chips de uso geral, como microprocessadores, os ASICs são projetados para executar tarefas específicas e são otimizados em termos de desempenho, consumo de energia e custo.

O processo de *design* ASIC envolve várias etapas, como definição de requisitos de circuito, criação de especificações de projeto, implementação do circuito em um nível de abstração de alto nível, simulação e verificação, síntese lógica, layout físico, verificação pós-layout e fabricação de chip. Ver (MARTIN, 2002).

Figura 1 – Fluxograma *design ASIC*



Fonte: (CHAUHAN, 2020).

2.2 *SystemVerilog*

SystemVerilog é uma linguagem de descrição de *hardware* realiza uma tarefa essencial no projeto de circuitos digitais, sendo uma linguagem de alto nível para modelagem comportamental e simulação de sistemas digitais complexos (SUTHERLAND; DAVIDMANN; FLAKE, 2006). É uma extensão avançada do *Verilog* que possui ainda mais recursos que expandem a capacidade de descrever, simular e verificar projetos *hardware*.

Uma dos benefícios porderosos da Linguagem *SystemVerilog* é seu potencial em descrever circuitos digitais em diferentes níveis de abstração. Ela realiza descrição estrutural (os componentes do circuito são conectados através de portas e sinais) e descrição comportamental (o comportamento do circuito é especificado por meio de regras e equações). Essa flexibilidade permite que os *designers* escolham o nível de detalhe que melhor se adapta à tarefa em questão.

A entrada de dados é um aspecto significativo do *SystemVerilog*. Ele aumenta a precisão e a confiabilidade da modelagem de circuitos, incorporando tipos de dados robustos, como números inteiros, números reais, vetores e estruturas definidas pelo usuário. Além disso, a linguagem facilita a criação de circuitos assíncronos e paralelos com suporte a primitivas de simultaneidade, possibilitando a execução de múltiplas operações simultaneamente.

A verificação de projeto é um elemento crucial do SystemVerilog, oferecendo capacidades robustas para gerar asserções (VIJAYARAGHAVAN; RAMANATHAN, 2005) que servem como instruções lógicas para testar propriedades específicas do circuito. Isso auxilia na detecção de erros e garante a precisão do projeto. Além disso, o *SystemVerilog* facilita a verificação formal, um método avançado que permite a verificação automática

das propriedades do projeto sem a necessidade de simulação, aumentando assim o rigor e a eficácia do processo de verificação.

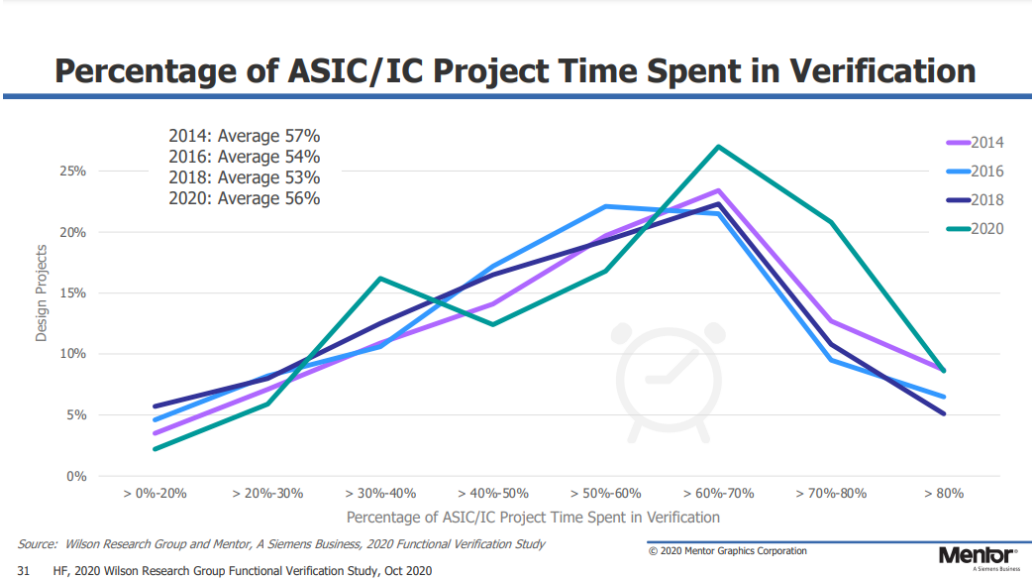
O padrão *IEEE Std 1800* é responsável pela especificação e manutenção do *SystemVerilog*. Este padrão descreve a linguagem, abrangendo sua sintaxe, semântica e vários recursos.

2.3 Verificação Funcional

No processo de desenvolvimento de circuitos digitais, a verificação funcional desempenha um papel crucial, garantindo que a implementação *RTL* esteja alinhada com os requisitos especificados. No entanto, devido à crescente complexidade dos blocos *ASIC* contemporâneos, testes exaustivos já não são uma opção viável. Consequentemente, há necessidade de abordagens mais eficientes que possam abranger uma ampla gama de casos de teste dentro das restrições de tempo e recursos.

O processo de verificação, que é um aspecto vital do processo de desenvolvimento, é responsável por aproximadamente 70% do esforço geral de design (KUMAR; GOPINATH, 2016). Ocupa posição de destaque no caminho crítico do projeto e demanda bastante tempo. No entanto, é viável diminuir a duração da verificação empregando técnicas de abstração e incorporando propriedade intelectual (PI) e designs pré-verificados. Ao empregar abstração no processo de verificação, pode-se agilizar o procedimento, minimizando a atenção em detalhes intrincados. Embora isto possa levar a uma diminuição do controlo sobre essas especificidades, revela-se uma táctica bem sucedida para agilizar o processo de verificação.A Figura 2 mostra o tempo usado na verificação em relação ao tempo total do desenvolvimento de um projeto *ASIC*.

Figura 2 – Tempo do projeto em porcentagem de *ASIC/IC* usado em verificação.



Fonte: (FOSTER, 2020).

Além disso, a implementação da automação é crucial para diminuir o tempo gasto na verificação. Através da automatização de tarefas repetitivas, o processo de verificação pode ser acelerado, resultando numa maior eficiência. A randomização é outra ferramenta eficaz que pode ser utilizada na automação da verificação. Ao gerar estímulos de teste de forma aleatória, um espectro mais amplo de cenários de utilização pode ser explorado, permitindo a identificação de potenciais problemas de forma mais eficiente.

A cobertura funcional e de código servem como métricas primárias para verificação funcional. Essas métricas fornecem um meio de avaliar até que ponto os estímulos gerados pelo ambiente de verificação se alinham com as opções disponíveis, conforme determinado pelo tamanho variável em *bits*.

2.3.1 Cobertura de Código

A avaliação dos testes do código utilizando o banco de testes é medida por uma métrica conhecida como cobertura de código. Essa métrica avalia a minuciosidade dos testes, determinando até que ponto vários componentes do código, como expressões, alternância de pinos e máquinas de estados finitos, foram exercitados (FATHY; SALAH; GUINDI, 2015). Vale a pena notar, entretanto, que a cobertura do código não indica se o teste está alinhado com as intenções originais do *design*.

Embora a cobertura de código seja valiosa para avaliar o rigor dos testes, ela não fornece garantia absoluta de que todos os possíveis caminhos e comportamentos de *design* foram explorados. Para conseguir uma avaliação mais abrangente, é essencial aumentar

a cobertura do código com cobertura funcional. Esta medida adicional garante que os elementos e requisitos específicos do projeto foram suficientemente testados.

A cobertura do código normalmente é produzida automaticamente pela ferramenta de verificação empregado. Esta ferramenta identificará quais partes do código foram realmente executadas durante o teste. Esta função facilita a identificação de falhas no processo de teste e direciona esforços para uma cobertura de projeto mais abrangente.

2.3.2 Cobertura Funcional

A cobertura funcional é o grau em que um *design* atende aos seus requisitos funcionais e identifica quaisquer discrepâncias potenciais em relação aos requisitos. A utilização das palavras-chave, *cover, covergroups, bins e cross* tem como objetivo avaliar se determinados pré-requisitos ou funcionalidades estão sendo cumpridos adequadamente durante os testes (SALAH, 2014).

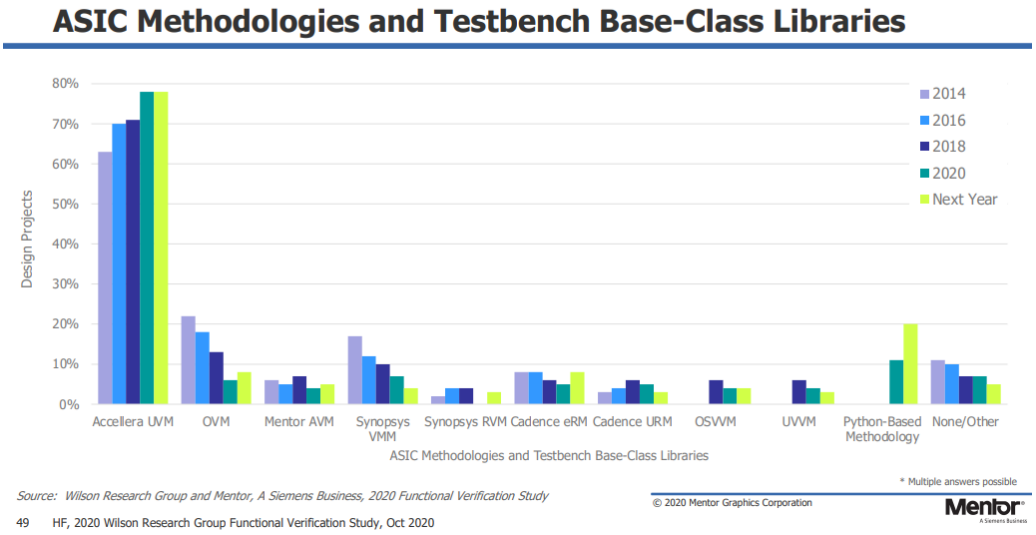
A cobertura funcional é mais extensa do que simplesmente executar o código, envolve uma análise mais aprofundada das funções de design e sua associação com as especificações.Ajuda a garantir que todos os recursos críticos do projeto sejam testados e fornece informações valiosas sobre a integridade e a conformidade do sistema.

Ao definir metas de cobertura funcional, os engenheiros de verificação determinam critérios para analisar se *design* suprir às expectativas e requisitos funcionais estipulados. Esta abordagem é crítica para garantir que o *design* seja implementado de modo correto e que funcionalidades principais sejam testadas corretamente.

2.4 *Universal Verification Methodology (UVM)*

A *UVM* é uma metodologia popular de verificação da funcionalidade do *hardware*. Baseada na programação orientada a objetos e na linguagem *SystemVerilog*, fornece uma estrutura que pode ser usada para criar ambientes de verificação eficazes. Com a sua portabilidade, a *UVM* pode ser combinada com diversas ferramentas *EDA*, esta combinação permite a reutilização do ambiente de verificação em diferentes plataformas de simular. É elaborado pela Accellera Systems Initiative® e promove a reúso de *testbenches* e elementos verificados, o que facilita a criação de testes e o desenvolvimento de modelos transacionais que irão melhorar a competência do processo de verificação.

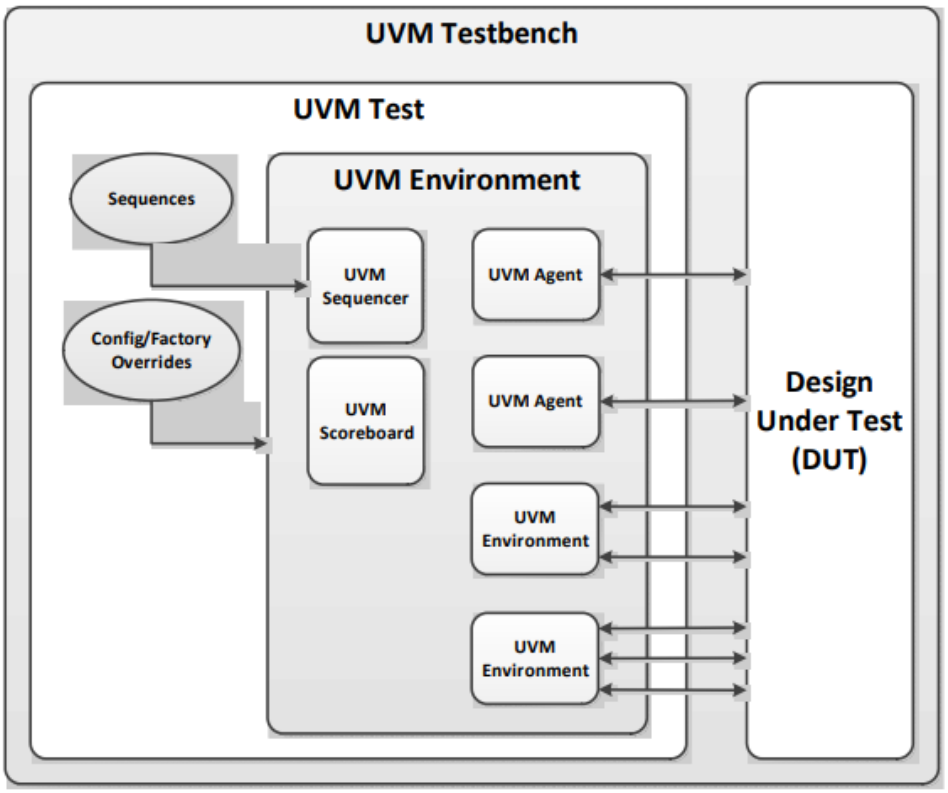
Figura 3 – Metodologias durante a História *ASIC* e Bibliotecas de Classe Base de *Testbench*



Fonte: (FOSTER, 2020)

A abordagem metodológica UVM é típica de um processo de design sigular. A Figura 4 mostra a hierarquia da arquitetura e suas partes constituintes. Posteriormente, as partes constituintes desta arquitetura serão descritas em (INITIATIVE, 2015).

Figura 4 – Arquitetura típica de um *Testbench UVM*



Fonte: (INITIATIVE, 2015)

2.4.1 *UVM Testbench*

O *UVM Testbench*, comumente referido como módulo *top*, consiste no módulo de instanciação *DUT* e na classe de teste *UVM Test*, que servem como conexões entre eles. Caso os componentes de verificação incluam elementos baseados em módulo, esses componentes também são instanciados no *UVM Testbench*. O *UVM Test* é instanciado dinamicamente durante o tempo de execução, possibilitando que o *UVM Testbench* seja compilado em uma circunstância e executado com inúmeros testes.

2.4.2 *UVM Test*

O *UVM Test* é o módulo principal do *UVM Testbench*, responsável por efetuar três tarefas principais. Essas tarefas incluem a criação de uma instância do ambiente de nível superior, a configuração do ambiente substituindo as configurações de fábrica ou usando o banco de dados de configuração e a geração de estímulos executando sequências *UVM* através do ambiente e direcionando-as para o *DUT*.

Na maioria dos casos, é estabelecida uma base de *UVM Test*, abrangendo a

instanciação do ambiente *UVM* e outros componentes compartilhados. Posteriormente, testes individuais baseiam-se nesse teste base, permitindo configurações de ambiente exclusivas e a seleção de sequências distintas para execução.

2.4.3 *UVM Environment*

O *UVM Environment* é composto pelos componentes de *agents* e *scoreboards*, bem como potencialmente outros ambientes dentro de sua hierarquia. Ele serve como um hub central para vários componentes essenciais do testbench, proporcionando a conveniência de configurá-los todos em um só lugar sempre que indispensável. Com o ambiente *UVM*, você tem a flexibilidade de incorporar vários *agents* para diferentes interfaces e vários *scoreboards* de avaliação para verificar diversos tipos de troca de dados. Essa abordagem simplificada permite ativar ou desativar facilmente componentes de verificação específicos para tarefas específicas, tudo em um único local.

2.4.4 *UVM Scoreboard*

O objetivo principal do *UVM Scoreboard* é validar as ações de um *DUT* designado. Através das portas de análise do *UVM Agent*, o *UVM Scoreboard* recebe transações que abrangem as entradas e saídas do *DUT*. Essas transações de entrada são processadas por um modelo de referência, também denominado preditor, para gerar transações antecipadas. Posteriormente, o *scoreboard* confronta as realizações projetadas com as realizações reais. Existem várias abordagens para a implementação do *scoreboard*, que diferem em termos do modelo de referência utilizado e do modo de comunicação entre o *scoreboard* e os demais componentes do *testbench*.

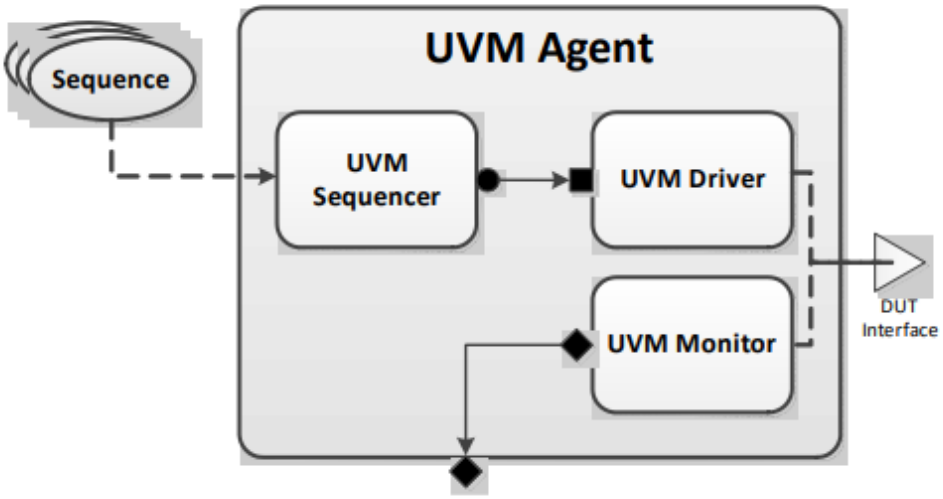
2.4.5 *UVM Agent*

O *UVM Agent* é composto pelos elementos *sequenciador, driver e monitor* de uma interface e funciona como uma unidade coletiva. Para gerenciar múltiplas interfaces, vários agentes podem ser utilizados, todos interligados ao *testbench* por meio do elementos de ambiente. Dentro da estrutura *UVM*, os agentes podem ser categorizados como ativos ou passivos.

Os agentes ativos dispõem de um *driver* e é capaz de manipular sinais, ao mesmo tempo que os agentes passivos consistem unicamente em um monitor e não têm controle sobre os pinos. É crucial manter o nível de abstração e integridade estrutural prometido pelo *UVM*, mesmo quando se trata de agentes passivos que compreendem apenas um monitor. Para conseguir isso, todos os agentes devem ser colocados no ambiente, em vez de serem tratados como subcomponentes solitários. Por via de regra, os agentes são considerados ativos, mas esta designação pode ser alterada utilizando o método *set*() no

banco de dados de configuração textitUVM. A Figura 5 fornece uma representação visual da arquitetura usual do *UVM Agent*.

Figura 5 – Arquitetura típica de um *UVM Agent*



Fonte: (INITIATIVE, 2015)

2.4.6 *UVM Sequence Item*

O *UVM Sequence Item* é o componente essencial da arquitetura UVM, abrangendo dados, métodos e restrições potencialmente relacionadas ao *DUT*. Servindo como a menor entidade transacional possível dentro de um ambiente de verificação, ela incorpora a essência de uma transação.

2.4.7 *UVM Sequence*

Um bloco de transações, conhecido como *UVM Sequence*, oferece a flexibilidade de utilizar transações conforme necessário, permitindo o uso de múltiplas transações. O objetivo principal de uma sequência é gerar transações e transmiti-las ao sequenciador.

2.4.8 *UVM Sequencer*

O *UVM Sequencer* assume a tarefa crucial de servir como mediador entre o *driver* e as transações, gerenciando efetivamente o fluxo de transações originadas de diferentes sequências. Facilitar a comunicação perfeita entre o *driver* e o sequenciador é o que a interface *TLM* torna possível.

2.4.9 *UVM Driver*

O *UVM Driver* é responsável por transformar as transações do sequenciador em ações de nível de sinal dentro do *DUT*. Esta conversão é obtida através da utilização de métodos como *try_next_item*(), *get_next_item*(), *item_done*() e *put*(). Normalmente, *UVM Driver* é implementado como uma classe parametrizada com o tipo de transação específico em que opera.

2.4.10 *UVM Monitor*

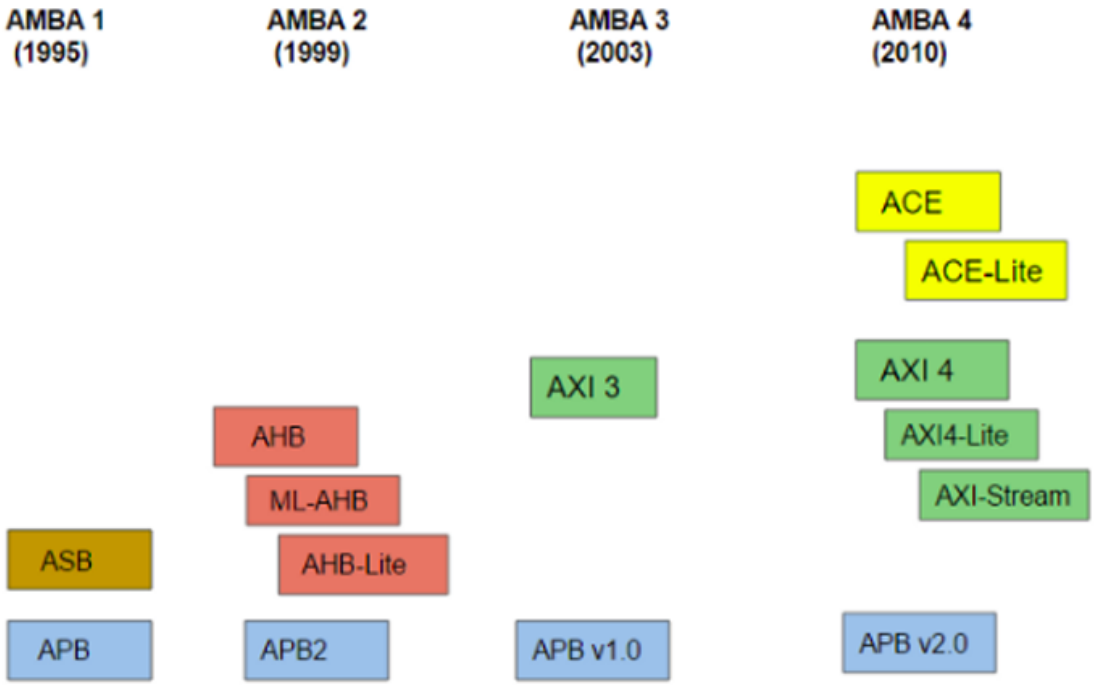
O*UVM Monitor* rastreia e analisa efetivamente a atividade em nível de sinal do *DUT*, convertendo-a em transações que podem ser encaminhadas para outros componentes para análise aprofundada.Habitualmente, o monitor lida com vários tipos de transações, incluindo coleta e registro de cobertura, antes de transmiti-las aos *scoreboards*.

2.5 Protocolo *AXI4-Lite*

O barramento *AMBA (Advanced Microcontroller Bus Architecture)* é um conjunto de especificações desenvolvido pela ARM Holdings para padronizar a comunicação entre componentes em sistemas baseados em processadores ArmDeveloper®. No âmbito da *AMBA*, o protocolo *AXI (Advanced eXtensible Interface)* é uma das opções mais utilizadas, oferecendo uma estrutura robusta e flexível para transferência de dados em sistemas digitais.

O *AXI4-Lite* é uma versão simplificada do protocolo *AXI4*, projetado para comunicação mestre-escravo de baixa complexidade em sistemas digitais. Ele mantém muitas das características fundamentais do *AXI4* completo, mas elimina recursos mais avançados para oferecer uma solução mais leve e eficiente em termos de recursos. A Figura 6 mostra a evolução do *AMBA* ao decorrer dos anos.

Figura 6 – Evolução do *AMBA (Advanced Microcontroller Bus Architecture)*



Fonte: (ARM, 2011)

2.5.1 Características Principais

2.5.1.1 Descomplicado e eficiente

O *AXI4-Lite* foi projetado para priorizar a simplicidade e a eficiência, garantindo uma ótima taxa de transferência de dados. Este protocolo de comunicação fornece uma solução simplificada para conectar um *manager* (geralmente um processador) e um *subordinate* (como um periférico), sem os requisitos de recursos adicionais do *AXI4* completo.

2.5.1.2 Comunicação *manager* e *Subordinate*

A arquitetura *manager-subordinate* é empregada no *AXI4-Lite*, onde o *manager* é responsável por gerenciar o acesso aos registros de um ou vários dispositivos *subordinates*. Esta escolha de design simplifica a comunicação entre os componentes de *hardware*, permitindo que o *manager* leia e grave facilmente em endereços específicos nos registradores do *subordinate*.

2.5.1.3 Comunicação Síncrona

Semelhante ao protocolo *AXI4* completo, o *AXI4-Lite* também depende de comunicação síncrona. Isto implica que todas as transferências de dados são sincronizadas usando

um sinal de *clock* comum. Essa sincronização garante um comportamento consistente e previsível do sistema, tornando a integração e o desenvolvimento de software mais prático.

2.5.1.4 Operações de Leitura e Escrita

O *AXI4-Lite* agiliza a troca de dados entre o *manager* e o *subordinates*, permitindo operações rápidas e eficientes de leitura e gravação de ciclo único. Isso permite que o *manager* envie instruções de leitura e gravação de forma rápida e eficaz para endereços designados nos registradores do *subordinate*.

2.5.1.5 Endereçamento

O *AXI4-Lite* emprega um método de endereçamento simples no qual dispositivos escravos individuais são atribuídos a espaços de endereçamento de memória distintos. Esta abordagem simplificada facilita a interação perfeita entre os dispositivos *manager* e escravo, permitindo acesso direto e eficiente aos registros *subordinate*.

2.5.2 Sinais e Métodos

O *AXI4-Lite* faz uso de todas as cinco funcionalidades básicas e seus referidos sinais. Como mostra a Figura 7 abaixo:

Figura 7 – Sinais e métodos

Write Address channel	Write data channel	Write response channel	Read address channel	Read data channel
AWVALID	WVALID	BVALID	ARVALID	RVALID
AWREADY	WREADY	BREADY	ARREADY	RREADY
AWADDR	WDATA		ARADDR	RDATA

Fonte: (ARM, 2011)

2.5.2.1 Sinais:

- *AWADDR (Address Write Address)*: Este sinal é usado pelo *manager* para enviar o endereço de destino da transação de escrita ao *subordinate*.
- *AWVALID (Address Write Valid)*: Indica que o sinal *AWADDR* contém um endereço de escrita válido.

- *AWREADY (Address Write Ready)*: Indica que o *subordinate* está pronto para aceitar o endereço de escrita enviado pelo *manager*.
- *WDATA (Write Data)*: Este sinal transporta os dados a serem escritos no *subordinate* pelo *manager*.
- *WVALID (Write Valid)*: Indica que o sinal *WDATA* contém dados válidos para escrita.
- *WREADY (Write Ready)*: Indica que o *subordinate* está pronto para aceitar os dados escritos enviados pelo *manager*.
- *BVALID (Response Valid)*: Indica que o sinal *WVALID* contém uma resposta válida do *subordinate*.
- *BREADY (Response Ready)*: Indica que o *manager* está pronto para aceitar a resposta do *subordinate*.
- *ARADDR (Address Read Address)*: Usado pelo *manager* para enviar o endereço de destino da transação de leitura ao *subordinate*.
- *ARVALID (Address Read Valid)*: Indica que o sinal *ARADDR* contém um endereço de leitura válido.
- *ARREADY (Address Read Ready)*: Indica que o *subordinate* está pronto para aceitar o endereço de leitura enviado pelo *manager*.
- *RDATA (Read Data)*: Este sinal transporta os dados lidos do *subordinate* para o *manager*.
- *RVALID (Response Valid)*: Indica que o sinal RDATA contém dados de leitura válidos.
- *RREADY (Response Ready)*: Indica que o *manager* está pronto para aceitar os dados de leitura enviados pelo *subordinate*.

Todos os cinco canais de transação usam o mesmo processo de *handshake VALID/READY* para transferir endereços, dados e informações de controle. Este mecanismo de controle de fluxo bidirecional significa que tanto o *manager* quanto o *subordinate* podem controlar a taxa na qual a informação se move entre o *manager* e o textitsubordinate. A fonte de informação gera o sinal *VALID* para indicar quando o endereço, dados ou informações de controle estão disponíveis. O destino da informação gera o sinal READY para indicar que pode aceitar a informação. O handshake é concluído se os sinais *VALID* e *READY* em um canal forem ativados durante uma transição de *clock* ascendente.

2.5.2.2 Métodos:

- *Write Transaction (Transação de Escrita)*: O *manager* envia um endereço e os dados a serem escritos no *subordinate*. O *subordinate* processa a operação e, se bem-sucedida, envia uma resposta ao *manager* indicando o resultado da transação.
- *Read Transaction (Transação de Leitura)*: O *manager* envia um endereço de leitura ao *subordinate*. O *subordinate* lê os dados no endereço especificado e os envia de volta ao *manager*. Além disso, o *subordinate* envia uma resposta ao *manager* indicando o resultado da transação.

Esses sinais e métodos formam a base do protocolo AXI4-Lite, proporcionando uma interface simples e eficiente para comunicação entre componentes em sistemas digitais. Eles permitem que o *manager* e o *subordinate* cooperem de maneira coordenada, facilitando operações de leitura e escrita em registros e memória do *subordinate*.

3 Metodologia de Desenvolvimento

Esse capítulo trata acerca da metodologia empregada para o desenvolvimento do *design* digital do barramento *AXI4-Lite*, bem como suas características e limitações. Outrossim, discute também sobre o ambiente de verificação funcional, suas características e elementos, e sua arquitetura geral.

3.1 *Design* Digital

O *design* do barramento *AXI4-Lite* foi elaborado de acordo com as especificações do protocolo citadas em seções anteriores. O enlace de comunicação proposto é de cárater básico e essencial, dado por:

- Comunicação entre apenas um *manager* e um *subordinate*.
- Transmissão de um único dado por vez (seja de leitura ou escrita).
- Descrever obedecendo o padrão (ARM, 2011)

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 *manager* e fazer testes mais robustos com a presença de mais *subordinate* no barramento.

Diante disso, faz-se necessário uma descrição da lógica programacional utilizada para conceber o barramento *AXI4-Lite*, bem como suas características funcionais e limitações, visto que o protocolo *AXI4-Lite* completo possui mais recursos que serão abordados no tópico.

3.1.1 Interface

O controlador *AXI4-Lite* projetado foi pensado com o foco no controle de barramento. Dessa forma, esses sinais foram colocados como entradas do bloco, facilitando a geração de estímulos e simplificando a lógica do algoritmo. O barramento possui a seguinte interface de entrada:

```
//...
    ////////////////////////////////////////////////////////////////////...

// axi4_lite_if.sv - Interface for AXI4 Lite Bus
//
// Description:
// ------------
// Defines the interface between master and subordinate
//of the AXI4 Lite bus.
//...
    ////////////////////////////////////////////////////////////////////...


// import the global definitions of AXI4 Lite bus
import axi4_lite_Defs::*;

interface axi4_lite_if(
                        input logic ACLK,       // System clock
                        input logic ARESETN     // System reset, active ...
    low
                      );

// declare the signals in Read Address Channel

logic [Addr_Width-1:0] ARADDR;    // Read Address

logic ARVALID;                    // Read Address Valid, master generates...
     this signal when the read address and the control signals are valid

logic ARREADY;                    // Read Address Ready, subordinate ...
    generates this signal when it can accept the read address and control...
     signals


//declare the signals in Read Data Channel

logic [Data_Width-1:0] RDATA;     // Read Data

logic RVALID;                     // Read valid, subordinate generate ...
    this signal when read data is valid

logic RREADY;                     // Read Ready, master generates this ...
    signal when it can accept read data


// declare the signals in Write Address Channel

```

```
38  logic [Addr_Width-1:0] AWADDR;   // Write Address
39
40  logic AWVALID;                   // Write Address valid, master ...
      generates this signal when write address and control signals are ...
      valid
41
42  logic AWREADY;                   // Write Address Ready, subordinate ...
      generates this signal when it can accept write address and control ...
      signals
43
44
45 // declare the signals in Write Data Channel
46
47 logic [Data_Width-1:0] WDATA;      // Write Data
48
49 logic WVALID;                      // Write Valid, master generates this ...
     signal when write data is valid
50
51 logic WREADY;                      // Write ready, subordinate generates ...
     this signal when it can accept write data
52
53
54 // declare the signals in Write Response Channel
55
56 logic BVALID;                      // Write Response valid, subordinate ...
     generates this signal when write response on bus is valid.
57
58 logic BREADY;                      // Write Response Ready, master ...
     generates this signal when it can accept write response
59
60
61 // declare the modport for master interface
62
63 modport master_if (
64
65 input ARREADY,
66
67 input RDATA,
68
69 input RVALID,
70
71 input AWREADY,
72
73 input BVALID,
74
75 input ACLK,
76
```

```
77 input ARESETN,
78
79 output ARADDR,
80
81 output ARVALID,
82
83 output RREADY,
84
85 output AWADDR,
86
87 output AWVALID,
88
89 output WDATA,
90
91 output BREADY,
92
93 output WVALID,
94
95 input WREADY
96
97 );
98
99
100
101 // declare the modport for subordinate interface
102
103 modport subordinate_if (
104
105 output ARREADY,
106
107 output RDATA,
108
109 output RVALID,
110
111 output AWREADY,
112
113 output BVALID,
114
115 input ARADDR,
116
117 input ARVALID,
118
119 input RREADY,
120
121 input AWADDR,
122
123 input AWVALID,
```

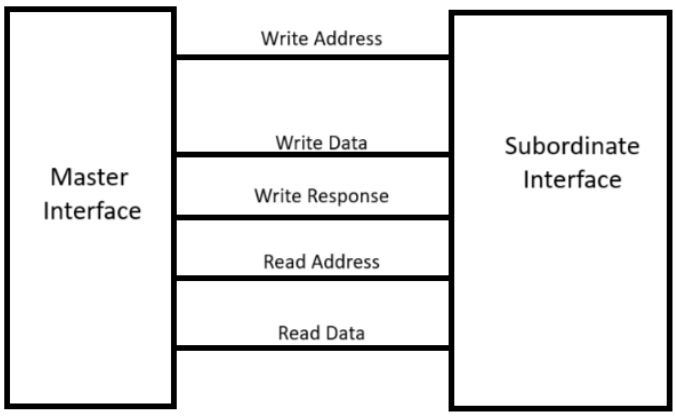
```
124
125 input WDATA,
126
127 input BREADY,
128
129 input ACLK,
130
131 input ARESETN,
132
133  input WVALID,
134
135 output WREADY
136
137 );
138
139
140
141 endinterface: axi4_lite_if
```

Fonte: Autoria própria.

3.1.2 Lógica Programacional

Para descrever a lógica de funcionamento e o controle dos sinais transmitidos no barramento, foi utilizada os gráficos, ilustrados pela Figura 8:

Figura 8 – Conexão do manager com o Subordinate



Fonte: Autoria própria.

Cada linha define uma etapa de envios de informações. Juntando ao que foi disertado até aqui, obteêm-se uma visão mais clara para se deifinir os parâmetros, os sinais e os

métodos necessários para descrever o design.

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

```
1  //...
       ////////////////////////////////////////////////////////////////////...
2  // axi4_lite_manager.sv - manager module for AXI4 Lite Bus
3  //
4  // Description:
5  // ------------
6  // The read and write operations are controlled by a central module
7  //known as the manager module. To carry out these operations,
8  //two separate finite state machines (FSMs) have been implemented.
9  //Each FSM consists of four states, which guide the sequence of
10 //steps required to successfully complete the transaction.
11 ////////////////////////////////////////////////////////////////////////
12
13 // import the global definitions of AXI4 Lite bus
14 import axi4_lite_Defs::*;
15
16 module axi4_lite_manager(
17
18 input logic rd_en,                                    // read enable ...
       signal
19 input logic wr_en,                                    // write enable ...
       signal
20 input logic [Addr_Width-1:0] Read_Address,            // input read ...
       address
21 input logic [Addr_Width-1:0] Write_Address,           // input write ...
       address
22 input logic [Data_Width-1:0] Write_Data,              // input write data
23 axi4_lite_if.manager_if M                          // interface as a ...
       manager
24 );
25
26
27 // declare the state variables for the FSM
28 //typedef enum logic [1:0] {IDLE, ADDR, DATA, RESP} state;              ...
                      // four states for read and write operation
29 state current_state_read, next_state_read, current_state_write, ...
       next_state_write;       // current and next state variables for read ...
       and write operation
30
31
32 ////////////////////////////////////////////////////////////
33 // FSM's to implement read and write operations of the manager
```

```
34 ////////////////////////////////////////////////////////////
35
36 //////////////////FSM to implement read operation////////////
37
38 // state register for read operation
39 always_ff@ (posedge M.ACLK, negedge M.ARESETN)
40 begin
41   if(M.ARESETN == 0)                                          // active...
       low reset
42     current_state_read <= IDLE;                          // go to IDLE state
43   else
44     current_state_read <= next_state_read;                 // else go to ...
     next state
45 end     // state register for read operation
46
47 // next state and output logic for read operation
48 always_comb
49 begin
50 //  M.RREADY  = 1'b0;
51 //  M.ARVALID = 1'b0;
52 //  M.ARADDR = 'z;
53
54
55   unique case(current_state_read)
56
57   IDLE: begin                                                 // ...
     initialize the signals of the read channels to zero
58       M.RREADY  = 1'b0;
59       M.ARVALID = 1'b0;
60       if(rd_en)                                        // when read enable ...
     is asserted, go to ADDR state if there is a new input address else ...
     stay in IDLE state
61       begin
62         if (Read_Address == M.ARADDR)
63         begin
64           next_state_read = IDLE;
65         end
66         else
67         begin
68           next_state_read = ADDR;
69         end
70       end
71       else
72       begin
73         next_state_read = IDLE;
74       end
75     end
```

```
76
77
78    ADDR:    begin
79        M.ARADDR  = Read_Address;                    // assign the input ...
      address to the read address(ARADDR) signal of the read address ...
      channel
80        M.ARVALID = 1'b1;                            // assert ARVALID to ...
      indicate that ARADDR is valid
81        M.RREADY  = 1'b1;                            // assert RREADY to ...
      indicate that manager is ready to receive the data from subordinate
82        if(M.ARREADY)                                // when ARREADY is ...
      asserted by the subordinate, then go to DATA state else stay in ADDR ...
      state
83            next_state_read = DATA;
84        else
85            next_state_read = ADDR;
86     end
87
88
89    DATA: begin
90        M.ARVALID = 1'b0;                            // deassert ARVALID ...
      indicating that address has been transferred from manager to ...
      subordinate
91        if(M.RVALID)                                 // when RVALID is ...
      asserted by subordinate, then go to RESP state else stay in the same ...
      state
92            next_state_read = RESP;
93        else
94            next_state_read = DATA;
95     end
96
97
98    RESP: begin
99        M.RREADY = 1'b0;                             // deassert RREADY ...
      indicating that data has been transferred from subordinate to manager...
       and go to IDLE state
100       next_state_read = IDLE;
101    end
102
103   endcase
104
105 end    // next state and output logic for read operation
106
107
108
109 ////////////////////////////FSM to implement write operation...
      ////////////////////////////////////
```

```
110
111 // state register for write operation
112 always_ff@(posedge M.ACLK, negedge M.ARESETN)
113 begin
114   if(M.ARESETN == 0)                                          // active...
      low reset
115     current_state_write <= IDLE;                              // go to IDLE ...
     state
116   else
117     current_state_write <= next_state_write;                  // else go to next ...
     state
118 end   // state register for write operation
119
120
121 // next state and output logic for write operation
122 always_comb
123 begin
124 //      M.AWVALID = 1'b0;
125 //      M.WVALID  = 1'b0;
126 //      M.BREADY  = 1'b0;
127 //      M.AWADDR = 'z;
128 //      M.WDATA = 'z;
129   unique case(current_state_write)
130
131   IDLE: begin                                                 // ...
     initialize the signals of the write channels to zero
132       M.AWVALID = 1'b0;
133       M.WVALID  = 1'b0;
134       M.BREADY  = 1'b0;
135       if(wr_en)                                              // when write enable ...
     is asserted, go to ADDR state if there is a new input address and ...
     data else stay in IDLE state
136       begin
137         if (Write_Address == M.AWADDR && Write_Data == M.WDATA)
138         begin
139           next_state_write = IDLE;
140         end
141         else
142         begin
143           next_state_write = ADDR;
144         end
145       end
146       else
147       begin
148         next_state_write = IDLE;
149       end
150     end
```

```
151
152
153   ADDR:    begin
154
155       M.AWADDR  = Write_Address;                    // assign the input ...
      write address to the write address(AWADDR) signal of the write ...
      address channel
156       M.AWVALID = 1'b1;                             // asssert AWVALID to ...
      indicate that AWADDR is valid
157       M.WDATA   = Write_Data;                       // assign the input ...
      write data to the write data(WDATA) signal of the write data channel
158       M.WVALID  = 1'b1;                             // assert WVALID to ...
      indicate that WDATA is valid
159       M.BREADY  = 1'b1;                             // assert BREADY to ...
      indicate that manager is ready to receive the response from the ...
      subordinate
160       if(M.WREADY)                                  // when AWREADY is ...
      asserted by the subordinate, then go to DATA state else stay in ADDR ...
      state
161         next_state_write = DATA;
162       else
163         next_state_write = ADDR;
164     end
165
166
167
168   DATA: begin
169       M.AWVALID = 1'b0;                             // deassert AWVALID ...
      indicating that address has been transferred from manager to ...
      subordinate
170       M.WVALID  = 1'b0;                             // deassert WVALID ...
      indicating that data has been transferred from manager to subordinate
171       if(M.BVALID)                                  // when BVALID is ...
      asserted by the subordinate, then go to RESP state else stay in DATA ...
      state
172         next_state_write = RESP;
173       else
174         next_state_write = DATA;
175     end
176
177
178   RESP: begin
179       M.BREADY = 1'b0;                              // deassert BREADY ...
      indicating that write response transaction is over and go to IDLE ...
      state
180       next_state_write = IDLE;
181     end
```

```
182
183   endcase
184 end    // next state and output logic for write operation
185
186 endmodule: axi4_lite_manager
```

Fonte: Autoria própria.

3.1.3 Limitações

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

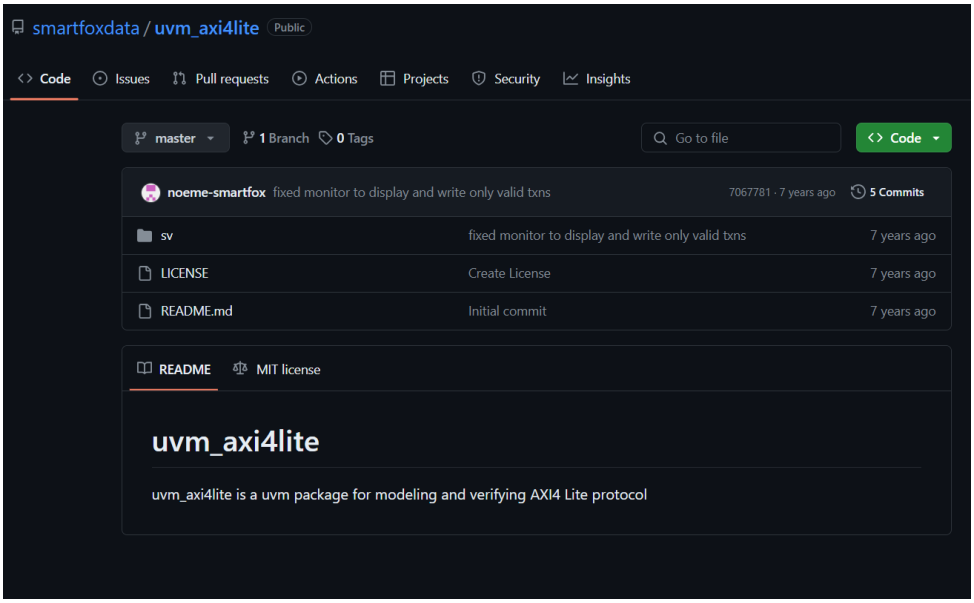
- Transmissão de um dado por vez no barramento .
- Não há tratamento do não reconhecimento de uma operação.
- Não apresenta todos os modos de funcionamento, apenas o modo normal sincronizado com o *clock*.

As limitações servem como um lembrete da natureza fundamental e crucial do bloco projetado, ao mesmo tempo que apresentam oportunidades interessantes para pesquisas futuras, incorporando uma ampla gama de funcionalidades descritas em (ARM, 2011).

3.2 Ambiente UVM de Verificação

O ambiente de verificação funcional feito para testar o controlador *AXI4-Lite* foi pensado a partir da metodologia *UVM*. Cada componente utilizado faz parte da arquitetura geral do ambiente. Para isso utilizou-se da UVM desenvolvida pela (INC., 2017). O código relacionado a cada componente é fornecido na plataforma GitHub® como se obverva na Figura 9.

Figura 9 – Home page do código no GitHub do $UVM_axi4lite$



Fonte: Autoria própria.

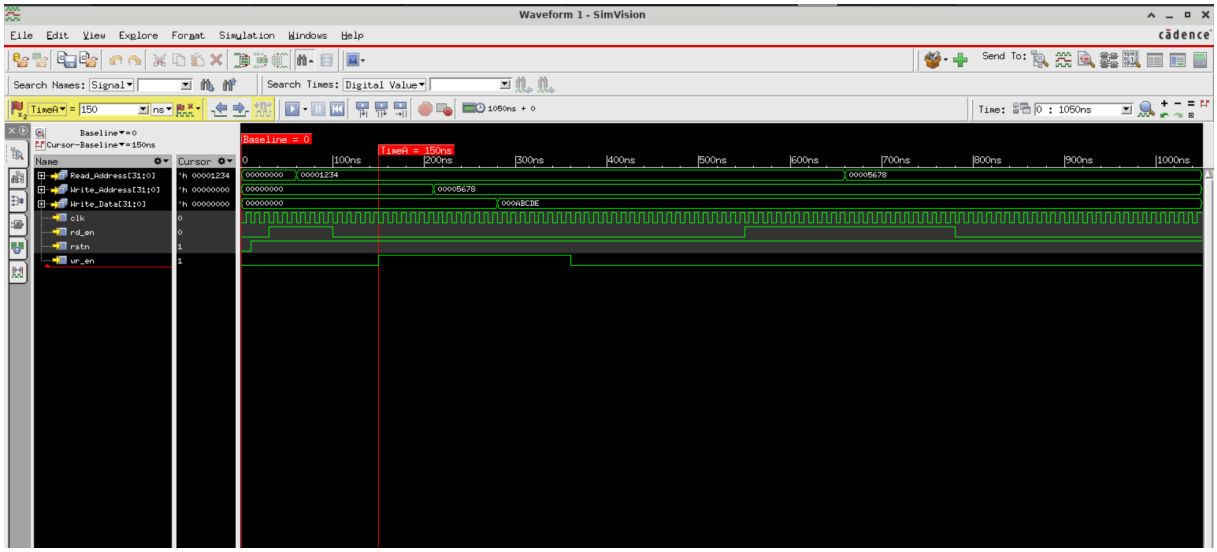
4 Resultados Obtidos

Neste capítulo, os resultados da simulação serão exibidos e analisados *RTL*, o resultado das confrontos do *scoreboard* e as métricas de cobertura funcional.

4.1 Simulação *RTL* e Síntese Lógica

Para efetuar a simulação *RTL* foi utilizada a ferramenta da Cadence© Xcelium Logic Simulator™ e para o visualização do funcionamento por meio das formas de onda foi usada a software SimVision Waveform™. Como visualiza-se na Figura 10, a simulação ocorreu sem erros e seu funcionamento ocorre de acordo com o especificado em (ARM, 2011).

Figura 10 – Simulação RTL do barramento *AXI4-Lite*



Fonte: Autoria própria.

4.2 Resultados e Cobertura Funcional

Em se tratando dos resultado da Cobertura Funcional do AXI4-Lite não foi possível obeter sucesso. Pois o mesmo iniciou um ciclo de transações e não teve fim. Assim tornou-se expectativas para pesquisas futuras, alinhar o RTL com a UVM sugerida para construir uma cobertura funcional.

Contudo o *testbench* decrito no *RTL* se mostrou bastante eficiente para dar indícios que o design descrito anteriormente pode sim ser verificado. Como mostra a Figura 11 a

seguir:

Figura 11 – Resultado do *testbench RTL* do barramento *AXI4-Lite*

```
        Design hierarchy summary:
                               Instances  Unique
                Modules:               4       4
                Interfaces:            1       1
                Verilog packages:      1       1
                Registers:            35      35
                Scalar wires:          4       -
                Vectored wires:        3       -
                Always blocks:        12      12
                Initial blocks:        1       1
                Pseudo assignments:    7       -
                Compilation units:     1       1
                Process Clocks:        3       2
        Writing initial simulation snapshot: worklib.axi4_lite_top_test:sv
Loading snapshot worklib.axi4_lite_top_test:sv ..................... Done
xcelium> source /Tools/cadence/XCELIUM2309/tools/xcelium/files/xmsimrc
xcelium> run
Success: The data was written and read correctly.
Simulation complete via $finish(1) at time 1050 NS + 0
./testbench.sv:77         #140 $finish;
```

Fonte: Autoria própria.

Tanto o código do testbench e quanto o do *manager* e do *subordinate* faltantes nesse conteúdo, encontram-se nos apêndices deste documento.

5 Conclusões

Este projeto buscou criar um *design digital* de um barramento AXI4-Lite, bem como fazer uso do método UVM da verificação funcional. Foram estudadas as especificações do protocolo e reconhecidas as funções fundamentais e essenciais. A partir disso, foi criado um elo de comunicação primário entre o *manager* e o *subordinate* que serviria para avaliar a funcionalidade considerada essencial.

Baseado nisso, foi realizado o *design* digital e buscou-se um ambiente de verificação que agisse conforme a comunicação do *manager* e o *subordinate*. Com os resultados da simulação, foi possível identificar o bom funcionamento do barramento.

- Ampliar as funcionalidades do *design* digital chegando próximo de um AXI completo.
- Interligá-lo a um microcontrolador e fazer testes com o inúmeros componentes.
- Resolver os problemas relacionados a cobertura funcional
- Aumentar a quantidade de dados possíveis de serem enviados em uma única transmissão.
- E Fazer a síntese do barramento AXI4-Lite para melhores resultados.

Referências Bibliográficas

ARM. *Documentation – Arm Developer,*. 7.0. ed. [S.l.], 2011. Disponível em: <http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf>. Citado 5 vezes nas páginas 13, 14, 17, 27 e 29.

CHAUHAN, K. Asic design flow in vlsi engineering services (a quick guide). *eInfoChips*, 2020. Citado na página 4.

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 6.

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 6 e 8.

INC., S. D. S. *Code UVM*. 7.0. ed. [S.l.], 2017. Disponível em: <https://github.com/smartfoxdata/uvm_axi4lite?tab=MIT-1-ov-file>. Citado na página 27.

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 8, 9 e 11.

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 5.

MARTIN, K. *Digital Integrated Circuit Design*. second. [S.l.]: Oxford University Press, 2002. Citado na página 3.

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 7.

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 4.

VIJAYARAGHAVAN, S.; RAMANATHAN, M. *A practical guide for SystemVerilog assertions*. first. [S.l.]: Springer Science Business Media, 2005. Citado na página 4.

ANEXO A – Códigos *RTL* em *SystemVerilog*

Neste apêndice é exibido os códigos para a implementação dos módulos usados na linguagem de descrição de *hardware SystemVerilog*.

- Manager:

```
//...
    ////////////////////////////////////////////////////////////////////...

// axi4_lite_manager.sv - manager module for AXI4 Lite Bus
//
// Description:
// ------------
// The read and write operations are controlled by a central module
//known as the manager module. To carry out these operations,
//two separate finite state machines (FSMs) have been implemented.
//Each FSM consists of four states, which guide the sequence of
//steps required to successfully complete the transaction.
//...
    ////////////////////////////////////////////////////////////////////...


// import the global definitions of AXI4 Lite bus
import axi4_lite_Defs::*;

module axi4_lite_manager(

input logic rd_en,                                      // read enable...
     signal
input logic wr_en,                                      // write ...
    enable signal
input logic [Addr_Width-1:0] Read_Address,              // input read ...
    address
input logic [Addr_Width-1:0] Write_Address,             // input write...
     address
input logic [Data_Width-1:0] Write_Data,                // input write...
     data
axi4_lite_if.manager_if M                           // interface as a ...
    manager
);
```

```
25
26
27 // declare the state variables for the FSM
28 //typedef enum logic [1:0] {IDLE, ADDR, DATA, RESP} state;          ...
                            // four states for read and write ...
    operation
29 state current_state_read, next_state_read, current_state_write, ...
    next_state_write;       // current and next state variables for ...
    read and write operation
30
31
32 ////////////////////////////////////////////////////////////
33 // FSM's to implement read and write operations of the manager
34 ////////////////////////////////////////////////////////////
35
36 /////////////////FSM to implement read operation////////////
37
38 // state register for read operation
39 always_ff@ (posedge M.ACLK, negedge M.ARESETN)
40 begin
41   if(M.ARESETN == 0)                                          // ...
    active low reset
42     current_state_read <= IDLE;                          // go to IDLE ...
    state
43   else
44     current_state_read <= next_state_read;                 // else go ...
    to next state
45 end     // state register for read operation
46
47 // next state and output logic for read operation
48 always_comb
49 begin
50 //  M.RREADY  = 1'b0;
51 //  M.ARVALID = 1'b0;
52 //  M.ARADDR = 'z;
53
54
55   unique case(current_state_read)
56
57   IDLE: begin                                                 // ...
    initialize the signals of the read channels to zero
58       M.RREADY  = 1'b0;
59       M.ARVALID = 1'b0;
60       if(rd_en)                                        // when read ...
    enable is asserted, go to ADDR state if there is a new input ...
    address else stay in IDLE state
61       begin
```

```
62          if (Read_Address == M.ARADDR)
63          begin
64            next_state_read = IDLE;
65          end
66          else
67          begin
68            next_state_read = ADDR;
69          end
70        end
71        else
72        begin
73          next_state_read = IDLE;
74        end
75      end
76
77
78    ADDR:    begin
79        M.ARADDR  = Read_Address;                    // assign the ...
      input address to the read address(ARADDR) signal of the read ...
      address channel
80        M.ARVALID = 1'b1;                            // assert ...
      ARVALID to indicate that ARADDR is valid
81        M.RREADY  = 1'b1;                            // assert RREADY...
       to indicate that manager is ready to receive the data from ...
      subordinate
82        if(M.ARREADY)                                // when ARREADY ...
      is asserted by the subordinate, then go to DATA state else stay ...
      in ADDR state
83          next_state_read = DATA;
84        else
85          next_state_read = ADDR;
86      end
87
88
89    DATA: begin
90        M.ARVALID = 1'b0;                            // deassert ...
      ARVALID indicating that address has been transferred from ...
      manager to subordinate
91        if(M.RVALID)                                 // when RVALID ...
      is asserted by subordinate, then go to RESP state else stay in ...
      the same state
92          next_state_read = RESP;
93        else
94          next_state_read = DATA;
95      end
96
97
```

```
98    RESP: begin
99        M.RREADY = 1'b0;                                    // deassert ...
      RREADY indicating that data has been transferred from ...
      subordinate to manager and go to IDLE state
100       next_state_read = IDLE;
101    end
102
103   endcase
104
105 end    // next state and output logic for read operation
106
107
108
109 //////////////////////////FSM to implement write operation...
      ////////////////////////////////////
110
111 // state register for write operation
112 always_ff@(posedge M.ACLK, negedge M.ARESETN)
113 begin
114   if(M.ARESETN == 0)                                                // ...
      active low reset
115     current_state_write <= IDLE;                                  // go to ...
      IDLE state
116   else
117     current_state_write <= next_state_write;          // else go to next...
      state
118 end   // state register for write operation
119
120
121 // next state and output logic for write operation
122 always_comb
123 begin
124 //       M.AWVALID = 1'b0;
125 //       M.WVALID  = 1'b0;
126 //       M.BREADY  = 1'b0;
127 //       M.AWADDR = 'z;
128 //       M.WDATA = 'z;
129   unique case(current_state_write)
130
131   IDLE: begin                                                     // ...
      initialize the signals of the write channels to zero
132       M.AWVALID = 1'b0;
133       M.WVALID  = 1'b0;
134       M.BREADY  = 1'b0;
135       if(wr_en)                                          // when write ...
      enable is asserted, go to ADDR state if there is a new input ...
      address and data else stay in IDLE state
```

```
136         begin
137           if (Write_Address == M.AWADDR && Write_Data == M.WDATA)
138           begin
139             next_state_write = IDLE;
140           end
141           else
142           begin
143             next_state_write = ADDR;
144           end
145         end
146         else
147         begin
148           next_state_write = IDLE;
149         end
150       end
151
152
153   ADDR:    begin
154
155         M.AWADDR  = Write_Address;                    // assign the ...
      input write address to the write address(AWADDR) signal of the ...
      write address channel
156         M.AWVALID = 1'b1;                             // asssert ...
      AWVALID to indicate that AWADDR is valid
157         M.WDATA   = Write_Data;                       // assign the ...
      input write data to the write data(WDATA) signal of the write ...
      data channel
158         M.WVALID  = 1'b1;                             // assert WVALID ...
      to indicate that WDATA is valid
159         M.BREADY  = 1'b1;                             // assert BREADY ...
      to indicate that manager is ready to receive the response from ...
      the subordinate
160         if(M.WREADY)                                  // when AWREADY ...
      is asserted by the subordinate, then go to DATA state else stay ...
      in ADDR state
161           next_state_write = DATA;
162         else
163           next_state_write = ADDR;
164       end
165
166
167
168   DATA: begin
169         M.AWVALID = 1'b0;                              // deassert ...
      AWVALID indicating that address has been transferred from ...
      manager to subordinate
170         M.WVALID  = 1'b0;                              // deassert ...
```

```
    WVALID indicating that data has been transferred from manager to...
     subordinate
171       if(M.BVALID)                                    // when BVALID ...
    is asserted by the subordinate, then go to RESP state else stay ...
    in DATA state
172         next_state_write = RESP;
173       else
174         next_state_write = DATA;
175     end
176
177
178   RESP: begin
179       M.BREADY = 1'b0;                                  // deassert ...
    BREADY indicating that write response transaction is over and go...
     to IDLE state
180       next_state_write = IDLE;
181     end
182
183   endcase
184 end     // next state and output logic for write operation
185
186 endmodule: axi4_lite_manager
```

- Subordinate:

```
1 //...
    ////////////////////////////////////////////////////////////////////////
2 // axi4_lite_subordinate.sv - Subordinate module for AXI4 Lite Bus
3 //
4 // Description:
5 // ------------
6 // Contains a subordinate module which is driven by the manager for...
     the read and write operations.
7 // Implemented two FSMs for read and write operation and each one ...
    has four states to
8 // traverse through the sequence of steps to complete the ...
    transaction.
9 //...
    ////////////////////////////////////////////////////////////////////////
10
11 // import the global definitions of AXI4 Lite bus
12 import axi4_lite_Defs::*;
13
14
```

```
15 module axi4_lite_subordinate(
16         axi4_lite_if.subordinate_if   S                                  ...
                          // interface as a subordinate
17    );
18
19
20 // declare the state variables for the FSM
21 //typedef enum logic [1:0] {IDLE,ADDR,DATA,RESP} state;                  ...
                         // four states for read and write operation
22 state current_state_read, next_state_read, current_state_write, ...
    next_state_write;    // current and next state variables for ...
    read and write operation
23
24 // declare the internal variables
25 logic [Data_Width-1:0] readdata, writedata;                              ...
                       // read data and write data variables
26
27 // declare a memory array of size 4096
28 logic [4095:0][31:0]mem;                                                 ...
                             // memory of 4096 locations each of 32 ...
    bit wide
29
30
31 //read from memory
32 always_comb
33 begin
34   if(S.ARVALID == 1'b1)                  // when ARVALID is asserted ...
    by subordinate, assign the data in ARADDR location of the memory...
     to readdata variable which will be later given to manager
35         readdata = mem[S.ARADDR];
36 end
37
38 // write data into memory
39 always@(posedge S.ACLK)
40 begin
41   if(S.WVALID == 1'b1)                   // when WVALID is asserted ...
    by subordinate, assign the writedata sent from manager to the ...
    AWADDR location of the memory
42   begin
43     mem[S.AWADDR] = writedata;
44   end
45 end
46
47
48
49 ////////////////////////////////////////////////////////////
50 // FSM's to implement read and write operations of the subordinate
```

```
51 ////////////////////////////////////////////////////////////
52
53 /////////////////FSM to implement read operation////////////
54
55
56 // state register for read operation
57 always_ff@(posedge S.ACLK, negedge S.ARESETN)
58 begin
59   if(S.ARESETN == 0)                                    // active low...
     reset
60         current_state_read <= IDLE;                    // go to IDLE ...
    state
61   else
62     current_state_read <= next_state_read;       // else go to next ...
    state
63 end   // state register for read operation
64
65
66 // next state and output logic for read operation
67 always_comb
68 begin
69   //     S.ARREADY = 1'b0;
70   //     S.RVALID  = 1'b0;
71   //     S.RDATA = 'z;
72   unique case(current_state_read)
73
74   IDLE: begin                                           // initialize ...
    the signals of the read channels to zero
75       S.ARREADY = 1'b0;
76       S.RVALID  = 1'b0;
77       if (S.ARVALID && S.RREADY)           // when ARVALID and ...
    ARREADY are asserted by the manager, then go to ADDR state else ...
    stay in IDLE state
78       begin
79         next_state_read = ADDR;
80       end
81       else
82       begin
83         next_state_read = IDLE;
84       end
85        end
86
87   ADDR:    begin
88       S.ARREADY = 1'b1;                     // assert ARREADY to ...
    indicate that subordinate is ready to receive the address(ARADDR...
    ) from manager and go to DATA state
89       next_state_read = DATA;
```

```
90      end
91
92    DATA:    begin
93        S.ARREADY = 1'b0;                    // deassert ARREADY ...
      indicating that address has been transferred from manager to ...
      subordinate
94        S.RDATA   = readdata;                // assign the data from ...
      the given location of the memory to RDATA signal of the read ...
      data channel
95        S.RVALID  = 1'b1;                    // assert RVALID to ...
      indicate that RDATA is valid
96        if(S.RREADY)                         // when RREADY is ...
      asserted by manager, then go to RESP state else stay in DATA ...
      state
97          next_state_read = RESP;
98        else
99          next_state_read = DATA;
100     end
101
102   RESP:    begin
103       S.RVALID = 1'b0;                      // deassert RVALID ...
      indicating that data has been transferred from subordinate to ...
      manager and go to IDLE state
104       next_state_read = IDLE;
105     end
106
107   endcase
108
109 end   // next state and output logic for read operation
110
111
112
113 ////////////////////////////FSM to implement write operation...
      ////////////////////////////////////
114
115
116 // state register for write operation
117 always_ff@(posedge S.ACLK, negedge S.ARESETN)
118 begin
119   if(S.ARESETN == 0)                                  // active ...
      reset low
120     current_state_write <= IDLE;              // go to IDLE state
121   else
122     current_state_write <= next_state_write;  // else go to next ...
      state
123 end     // state register for write operation
124
```

```
125
126 // next state and output logic for write operation
127 always_comb
128 begin
129
130 //  S.AWREADY = 1'b0;
131 //  S.WREADY  = 1'b0;
132 //  S.BVALID  = 1'b0;
133 //  writedata = 'z;
134   unique case(current_state_write)
135
136   IDLE: begin                                          // initialize ...
    the signals of the write channels to zero
137       S.AWREADY = 1'b0;
138       S.WREADY  = 1'b0;
139       S.BVALID  = 1'b0;
140       if (S.AWVALID && S.WVALID)              // when AWVALID and ...
    WVALID are asserted by the manager, then go to ADDR state else ...
    stay in IDLE state
141         next_state_write = ADDR;
142       else
143         next_state_write = IDLE;
144    end
145
146
147   ADDR: begin
148       S.AWREADY = 1'b1;                       // assert AWREADY to ...
    indicate that subordinate is ready to receive the write address(...
    AWADDR) from manager
149       S.WREADY  = 1'b1;                       // assert WREADY to ...
    indicate that subordinate is ready to receive that write data(...
    WDATA)
150       writedata = S.WDATA;                    // assign the write ...
    data received from manager to the write data internal variable ...
    and go to DATA state
151       next_state_write = DATA;
152    end
153
154   DATA: begin
155       S.AWREADY = 1'b0;                        // deassert AWREADY ...
    indicating that address has been transferred from manager to ...
    subordinate
156       S.WREADY  = 1'b0;                        // deassert WREADY ...
    indicating that data has been transferred from manager to ...
    subordinate
157       S.BVALID  = 1'b1;                        // assert BVALID to ...
    indicate that write response is valid
```

```
158       if(S.BREADY)                                  // when BREADY is ...
    asserted by manager, then go to RESP state else stay in DATA ...
    state
159         next_state_write = RESP;
160       else
161         next_state_write = DATA;
162     end
163
164
165   RESP:    begin
166       S.BVALID = 1'b0;                              // deassert BVALID ...
    indicating that write response has been transferred from ...
    subordinate to manager and go to IDLE state
167       next_state_write = IDLE;
168     end
169
170   endcase
171
172 end   // next state and output logic for write operation
173
174 endmodule: axi4_lite_subordinate
```

- Testbench:

```
1  import axi4_lite_Defs::*;
2  module axi4_lite_top_test();
3
4      // parameters
5      parameter Addr_Width = 32;  // Address width
6      parameter Data_Width = 32;  // Data width
7
8      // Signal of clock and reset
9      logic clk;
10     logic rstn;
11
12     // Control signals
13     logic rd_en;
14     logic wr_en;
15
16     // Address and data signals
17     logic [Addr_Width-1:0] Read_Address;
18     logic [Addr_Width-1:0] Write_Address;
19     logic [Data_Width-1:0] Write_Data;
20     logic [Data_Width-1:0] Read_Data;
21
22     // Module Instance axi4_lite_top
```

```
23      axi4_lite_top dut (
24          .clk(clk),
25          .rstn(rstn),
26          .rd_en(rd_en),
27          .wr_en(wr_en),
28          .Read_Address(Read_Address),
29          .Write_Address(Write_Address),
30          .Write_Data(Write_Data)
31      );
32
33      // Initialization of signals
34      initial begin
35          clk = 0;
36          rstn = 0;
37          rd_en = 0;
38          wr_en = 0;
39          Read_Address = 0;
40          Write_Address = 0;
41          Write_Data = 0;
42          Read_Data = 0;
43
44          // Reset
45          #10 rstn = 1;
46
47          // Send a read request
48          #20 rd_en = 1;
49          #30 Read_Address = 16'h1234;
50          #40 rd_en = 0;
51
52          // Submit a writing request
53          #50 wr_en = 1;
54          #60 Write_Address = 16'h5678;
55          #70 Write_Data = 32'hABCDE;
56          #80 wr_en = 0;
57
58          // Wait a while for the writing to be processed
59          #90;
60
61          // Send a read request to verify that the data was written ...
    correctly
62          #100 rd_en = 1;
63          #110 Read_Address = 16'h5678;
64          #120 rd_en = 0;
65
66          // Wait a while for the reading to be processed
67          #130;
68
```

```
69          // Check whether the read data matches the written data
70          if (Read_Data !== Write_Data) begin
71              $display("Error: The data read does not match the data ...
    written.");
72          end else begin
73              $display("Success: The data was written and read ...
    correctly.");
74          end
75
76          // End the simulation
77          #140 $finish;
78      end
79
80      // Simulate behavior of module axi4_lite_top
81      always @(posedge clk) begin
82          if (rd_en) begin
83              // Simulate reading data from the specified address
84              case (Read_Address)
85                  // Simulate reading address 16'h1234
86                  16'h1234: Read_Data <= 32'h11223344;
87                  // Simulate reading address 16'h5678
88                  16'h5678: Read_Data <= Write_Data;
89                  default: Read_Data <= 0; // By default return 0 for ...
    unspecified addresses
90              endcase
91          end
92      end
93
94      // Clock with a period of 10 time units
95      always #5 clk = ~clk;
96
97  endmodule
```