Centro de Engenharia Elétrica e Informática
Departamento de Engenharia Elétrica

João Pedro Melquiades Gomes

Relatório de Estágio Supervisionado
Laboratório de Sistemas Embarcados e Computação Pervasiva - Embedded

Campina Grande
Novembro de 2023

João Pedro Melquiades Gomes

Relatório de Estágio Supervisionado
Laboratório de Sistemas Embarcados e Computação Pervasiva - Embedded

Relatório de Estágio Supervisionado submetido à Coordenação do Curso de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciências no Domínio da Engenharia Elétrica.

Orientador:
Gutemberg Gonçalves dos Santos Júnior, Dr.

Campina Grande
Novembro de 2023

João Pedro Melquiades Gomes

Relatório de Estágio Supervisionado
Laboratório de Sistemas Embarcados e Computação Pervasiva - Embedded

Relatório de Estágio Supervisionado submetido à Coordenação do Curso de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciências no Domínio da Engenharia Elétrica.

Aprovado em 16 de novembro de 2023

Gutemberg Gonçalves dos Santos Júnior,
Dr.
UFCG

Marcos Ricardo Alcântara de Morais,
Dr.
UFCG

Campina Grande
Novembro de 2023

AGRADECIMENTOS

Agradeço a meus pais, João Carlos e Iranês, por terem me incentivado e garantido minha educação desde minha alfabetização, sempre lutando para que eu estivesse nas melhores escolas através de bolsas. Ao meu padrinho, Edgley, que agiu como intermédio durante o ensino médio para que fosse possível que eu estudasse em uma das melhores escolas de Campina Grande através de uma bolsa.

Agradeço a minha noiva, Karine, por estar sempre comigo, pela compreensão, pelas brigas quando eu ultrapassava os limites e dedicava muito tempo ao trabalho, pela motivação quando eu achei que tudo ia dar errado. A conquista da graduação eu dedico a ela.

Agradeço aos meus amigos, a todos que tive a oportunidade de conhecer durante a jornada da graduação. Aos amigos companheiros de RPG, Kilian, Vinícius, Filipe, Renato e Gabriel, que propiciaram várias doses de alegrias nos vários mundos que criamos juntos nas sessões de RPG. Aos amigos que me auxiliaram durante o curso, sendo estes os mesmos citados anteriormente, com adição de Pedro Henrique, amigo que fiz no primeiro período e me ajuda até hoje, Lara Sobral, minha "madrinha" de curso, e Ronildo, grande amigo desde o ensino médio.

Agradeço aos meus irmãos escoteiros, movimento que me auxiliou a evoluir como ser humano durante a graduação, e que espero nunca deixar de fazer parte. A todos os chefes do grupo escoteiro Dom Luís Gonzaga Fernandes, na pessoa do chefe Fábio Góes, diretor presidente atual. Ao distrito da Borborema, na pessoa do chefe Cláudio Carvalho, e à Região dos Escoteiros da Paraíba, na pessao do atual presidente chefe Peron. Todos contribuiram de alguma forma para meu crescimento como escoteiro e como cidadão, e a maturidade que tenho hoje ao final do curso devo também a eles. Além disso, agradeço também a todos os jovens os quais fui responsável durante minha trajetória no movimento, com os quais aprendi muita coisa enquanto também os ensinava.

Agradeço a todos os engenheiros que me apoiaram durante minha jornada como aluno $PD\&I$ no Virtus, por terem formado todas as competências que tenho hoje e que irei exercer no mercado de trabalho. Aos engenheiros Pedro, Matheus, Kelvin, Kaline, Hugo e Gabriel. O futuro engenheiro que serei é também graças a essas pessoas.

Agradeço aos vários professores que pude conhecer durante o curso. Aqui, cirto meu orientador de estágio Gutemberg Júnior, responsável por me fazer querer seguir na área de microeletrônica, o professor Marcos Morais, o qual admiro pela carga de conhecimento acumulada em uma só pessoa, o que me motiva a adquirir mais conhecimento, e ao meu orientador de TCC, Edmar Candeia, que sempre foi um exemplo de professor por

sua forma de ministrar as aulas e sua preocupação com o aluno. Que todos continuem sendo essa fonte de motivação para mais jovens como eu.

Por fim, agradeço também a minha pequena cadela Lilica, que entrou em minha vida durante a graduação e também auxiliou a manter minha cabeça no lugar com os momentos de descontração que ela propiciou, seja com seus beijos inesperados ou com os surtos de energia que sempre me faziam rir.

"Se você acha que é possível ter uma vida perfeita, viverá em eterna frustração. Altos e baixos, alegria e tristeza, entusiasmo e decepção são partes integrantes da nossa existência. Lute sempre para melhorar e alegre-se com suas conquistas.'Muitas pessoas devem a grandeza de suas vidas aos problemas que tiveram de vencer.' "

Baden-Powell

RESUMO

A grande maioria dos circuitos integrados existentes atualmente são compostos por partes analógicas e partes digitais, caracterizando o que é denominado de *design* AMS. Entretanto, não existe ainda um padrão definido sobre a melhor forma de se verificar tais *designs*, com cada empresa tendo sua própria solução para o problema. Com isso, foi feita durante esse trabalho uma investigação e implementação de um ambiente UVM que consiga lidar com circuitos mistos, com base principalmente nos trabalhos em desenvolvimento do grupo de trabalho UVM-AMS que busca desenvolver um padrão universal de verificação para tais circuitos. O ambiente foi concebido através de uma abordagem que se aproxima bastante do exposto pelo grupo e que consegue lidar com diferentes formatos de *designs* mistos, demonstrando sua funcionalidade e potencial aplicação na indústria.

Palavras-chave: AMS, Circuitos mistos, *SystemVerilog*, verificação, UVM.

ABSTRACT

The vast majority of currently existing integrated circuits are composed of analog and digital parts, featuring what is called AMS design. Therefore, there is still no defined standard on the best way to verify such designs, with each company having its own solution to the problem. With this, during this work, an UVM environment was investigated and implemented so it can deal with mixed circuits, based mainly on the work in development of the UVM-AMS working group that is developing a universal verification standard for such circuits. A environment was conceived through an approach that closely aligns with what the group presented and manages different formats of mixed designs, demonstrating its functionality and potential application in the industry.

Keywords: AMS, Mixed circuits, SystemVerilog, Verification, UVM

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

UFCG	Universidade Federal de Campina Grande
UVM	*Universal Verification Methodology*
RAK	*Rapid Adoption Kit*
EDIF	*Electronic Design Interchange Format*
RHEL	*Red Hat Enterprise Linux*
SO	Sistema Operacional
SPICE	*Simulation Program with Integrated Circuit Emphasis*
PGA	*Programmable gain amplifier*
DUT	*Design Under Test*

SUMÁRIO

1 INTRODUÇÃO

Este relatório objetiva a descrição das atividades desenvolvidas durante o período de Estágio Supervisionado realizado no Laboratório de Sistemas Embarcados e Computação Pervasiva (Embedded), tendo esse ocorrido durante o período do dia 7 de agosto de 2023 a 10 de novembro de 2023, totalizando uma carga horária de 274 horas.

O estágio em questão foi realizado com o intuito de apresentar técnicas de integração de designs AMS em ambientes de verificação UVM, apontando tomadas de decisões e dificuldades de tal integração. Neste contexto, foram atribuídas ao estagiário atividades que se referem ao aprendizado do fluxo de circuitos analógicos e AMS, além da verificação de dois *designs* de mesma funcionalidade, porém escritos com diferentes técnicas cuja integração com UVM teria de ser feita de maneiras distintas.

O principal desafio dessa integração é conectar os dados analógicos ao ambiente UVM, visto que *SystemVerilog* não possui suporte para algumas das estruturas de dados utilizadas em *designs* mistos. Para isso, utilizou-se dos estudos feitos até o presente momento do grupo UVM-AMS da *Accelera*, organização que criou a metologia UVM. Com isso, tentou-se implementar uma arquitetura semelhante ao que está sendo proposta por eles e que futuramente virá a se tornar um novo padrão de verificação. Além disso, por ser uma temática relativamente nova, sem nenhum padrão bem definido, as soluções propostas durante o estágio não necessariamente são as melhores, podendo ser alteradas à medida que o padrão for desenvolvido.

1.1 LABORATÓRIO DE SISTEMAS EMBARCADOS E COMPUTAÇÃO PERVASIVA - EMBEDDED

O Laboratório de Sistemas Embarcados e Computação Pervasiva - Embedded, é uma parte do Centro de Engenharia Elétrica e Informática (CEEI) na Universidade Federal de Campina Grande (UFCG), localizada em Campina Grande, Paraíba. Fundado em dezembro de 2005, o laboratório está situado em um prédio de 600 metros quadrados no campus da UFCG. Através da UFCG, o Embedded é credenciado no Comitê da Área de Tecnologia de Informação (CATI) para receber recursos da Lei de Informática, com o Parque Tecnológico da Paraíba também credenciado no CATI para apoio financeiro.

O laboratório tem como característica a participação de alunos de graduação e de engenheiros, que se formam e atuam nas áreas de: Internet das coisas, *Software-Hardware Co-Design* e microeletrônica. Com isso, a equipe do Laboratório Embedded conta com cerca de 60 colaboradores, incluindo alunos de doutorado, mestrado e graduação, dos

cursos de Engenharia Elétrica e Ciência da Computação.

1.2 OBJETIVOS

A atividade de estágio, em seu escopo geral, possui as seguintes atividades objetivadas:

- 1. Entender o fluxo de desenvolvimento de circuitos mistos, habituando-se ao desenvolvimento na ferramenta *Virtuoso*;
- 2. Investigar o que é feito na indústria atualmente quanto à verificação de circuitos mistos;
- 3. Desenvolver ambientes de verificação de circuitos mistos utilizando UVM;
- 4. Elencar as tomadas de decisão e dificuldades encontradas durante o desenvolvimento.

O estagiário utilizou de *Rapid Adoption Kits* - RAKs, acessados na plataforma de suporte da *Cadence* para o aprendizado básico de como circuitos mistos podem ser modelados. Depois disso, atuou na implementação dos ambientes de verificação utilizando a linguagem *Systemverilog* e algumas outras ferramentas da *Cadence*, como *Simvision*, *Verisium Debug*, *Xrun* e *Vmanager*.

1.3 ORGANIZAÇÃO DO TRABALHO

O presente relatório está estruturado em 4 capítulos, incluindo este introdutório, conforme a seguir:

No Capítulo 2 serão apresentadas as tecnologias utilizadas, discorrendo sobre detalhes teóricos relacionados.

No Capítulo 3 serão abordadas as atividades realizadas pelo estagiário, com detalhes das implementações.

Por fim, no Capítulo 4 apresenta-se a conclusão sobre o trabalho.

2 TECNOLOGIAS UTILIZADAS

Nesta seção será apresentado o ferramental utilizado para a realização das tarefas.

2.1 GIT

O Git é uma ferramenta de controle de versão de código-fonte amplamente utilizada na indústria de desenvolvimento de software. Foi criado por Linus Torvalds, o mesmo criador do sistema operacional Linux, em 2005 (SPINELLIS, 2012). O Git é projetado para ajudar equipes de desenvolvedores a colaborar no desenvolvimento de projetos de software, acompanhando as mudanças no código-fonte ao longo do tempo.

Suas principais funcionalidades incluem:

1. Controle de Versão: O *Git* permite que os desenvolvedores rastreiem e gerenciem todas as alterações no código-fonte, registrando quem fez as mudanças e quando elas foram feitas.

2. Ramificação: Os desenvolvedores podem criar "ramificações" do código-fonte principal para trabalhar em novos recursos ou correções de *bugs* sem interferir no código principal. Isso facilita o desenvolvimento paralelo e a colaboração.

3. Fusão (*Merge*): Após a conclusão de uma tarefa em uma ramificação, as alterações podem ser "emergidas" de volta para o código principal, combinando o trabalho de diferentes colaboradores.

4. Histórico de *Commits*: O *Git* mantém um histórico detalhado de todos os *commits*, permitindo que os desenvolvedores revertam para versões anteriores do código, se necessário.

5. Distribuição: O *Git* é uma ferramenta distribuída, o que significa que cada desenvolvedor tem uma cópia completa do repositório, facilitando o trabalho offline e a colaboração entre equipes geograficamente dispersas.

6. Hospedagem na Nuvem: Serviços como o *GitHub* e o *GitLab* fornecem plataformas de hospedagem na nuvem para repositórios *Git*, o que simplifica o compartilhamento de código e a colaboração entre desenvolvedores.

O estagiário implementou suas atividades no seu repositório pessoal de forma privada, uma vez que os *designs* utilizados foram obtidos dentro da plataforma da *Cadence* e, por isso, são confidenciais. Entretanto, ao decorrer do relatório serão expostos os códigos do ambiente de verificação, além da macroarquitetura dos *designs* utilizados.

2.2 VIRTUOSO® ANALOG DESIGN ENVIRONMENT

O *Virtuoso® Analog Design Environment* é uma ferramenta de design poderosa que permite aos usuários criar e simular circuitos analógicos interativamente. Ele oferece uma interface de usuário consistente que facilita a transição entre diferentes ferramentas da Cadence, além de permitir a integração de ferramentas de terceiros. O ambiente de design é baseado em uma arquitetura aberta que suporta formatos padrão do setor, como EDIF e Virtuoso GDSII Stream.

Além disso, essa ferramenta oferece recursos avançados, como entrada de design analógico, hierarquia de design, anotação, simulação interativa, saída e análise de simulação e análise avançada. A simulação interativa permite que os usuários interrompam a simulação, verifiquem as tensões e correntes dos nós e continuem a simulação. A ferramenta também suporta a análise de parâmetros avançados, como ruído, distorção e sensibilidade. Com sua ampla gama de recursos e interface de usuário intuitiva, o Virtuoso Analog Design Environment é uma ferramenta essencial para designers de circuitos analógicos.

O estagiário utilizou essa ferramenta durante as primeiras fases do estágio, com o único objetivo de aprender mais sobre o fluxo de desenvolvimento de circuitos mistos. Posteriormente, todas as simulações foram feitas utilizando o *Xrun*.

2.3 XRUN

Essa ferramenta, também proprietária da *Cadence*, faz parte da instalação do *XCELIUM*, um conjunto de utilitários voltado para o desenvolvimento de circuitos integrados. Mais especificamente, o *Xrun* permite a compilação, elaboração e simulação de um *design*, além de aceitar uma grande variedade de formatos de arquivo como entrada. Dentre esses formatos: *Verilog*, *SystemVerilog*, VHDL, *Verilog AMS*, VHDL AMS, e *Specman e*, além de programas escritos em linguagens de programação mais genéricas, como C e C++.

2.4 SIMVISION

O *SimVision* é uma ferramenta de simulação da *Cadence* que oferece uma variedade de recursos para análise e depuração de *designs* de circuitos. Ele fornece uma interface gráfica intuitiva para visualizar e analisar resultados de simulação, além de suportar várias linguagens de descrição de hardware. Além disso, também é capaz de realizar análises avançadas, como análise de forma de onda e análise de potência, sendo útil tanto no desenvolvimento quanto na verificação dos circuitos. Além disso, o *SimVision* é altamente personalizável e oferece integração com outras ferramentas também da *Cadence*.

O estagiário utilizou essa ferramenta apenas durante o início da fase de simulação do primeiro *design*. Depois, ele passou a utilizar o *Verisium Debug* para depuração e análise das formas de onda.

2.5 VERISIUM DEBUG

O *Verisium Debug* é uma poderosa ferramenta de depuração integrada nativamente com várias plataformas de verificação e simulação da *Cadence*, incluindo o *Xcelium Logic Simulator*, a emulação *Palladium*, e a prototipagem *Protium*. Ele oferece recursos avançados de depuração, como rastreamento de *drivers*, visualização de formas de onda de alto desempenho e navegação eficiente na hierarquia de *design*. Além disso, essa ferramenta é capaz de realizar depuração em cenários de baixo consumo de energia com suporte a padrões como IEEE 1801/UPF e *X-propagation*. Ela também permite a depuração simultânea de RTL e *software* incorporado, facilitando análises rápidas de erros durante o *co-design* de *hardware* e *software*.

O estagiário passou a utilizar essa ferramenta para a depuração e simulação dos *testbenchs* implementados durante o estágio, visto que o *Verisium* apresenta melhorias em relação ao *SimVision* e logo irá ser aderido pela maior parte da indústria.

2.6 VERISIUM MANAGER

O *Verisium Manager* é uma poderosa ferramenta de gerenciamento de projetos de verificação de *hardware* que automatiza todas as etapas, desde o planejamento até o fechamento da verificação. Ele se destaca pela integração com a plataforma *Cadence Verisium AI-Driven Verification*, utilizando IA e *big data* para reduzir defeitos de silício e acelerar o tempo de lançamento no mercado. Além disso, o *Verisium Manager* oferece integrações com várias ferramentas de gerenciamento de especificações e requisitos, bem como métricas de cobertura geradas por outras ferramentas, proporcionando uma solução completa e personalizável para o planejamento da verificação. Ele também se conecta a ferramentas de gerenciamento de recursos e integração contínua, além de oferecer uma interface unificada baseada em navegação para relatórios e análises detalhadas, tornando o processo de verificação mais eficiente e transparente.

O estagiário utilizou do *Vmanager* durante o estágio apenas o recurso de lançamento de regressões e exportação de relatórios de simulação para a análise de um dos parâmetros do *testbench*.

3 RELATO DE ATIVIDADES

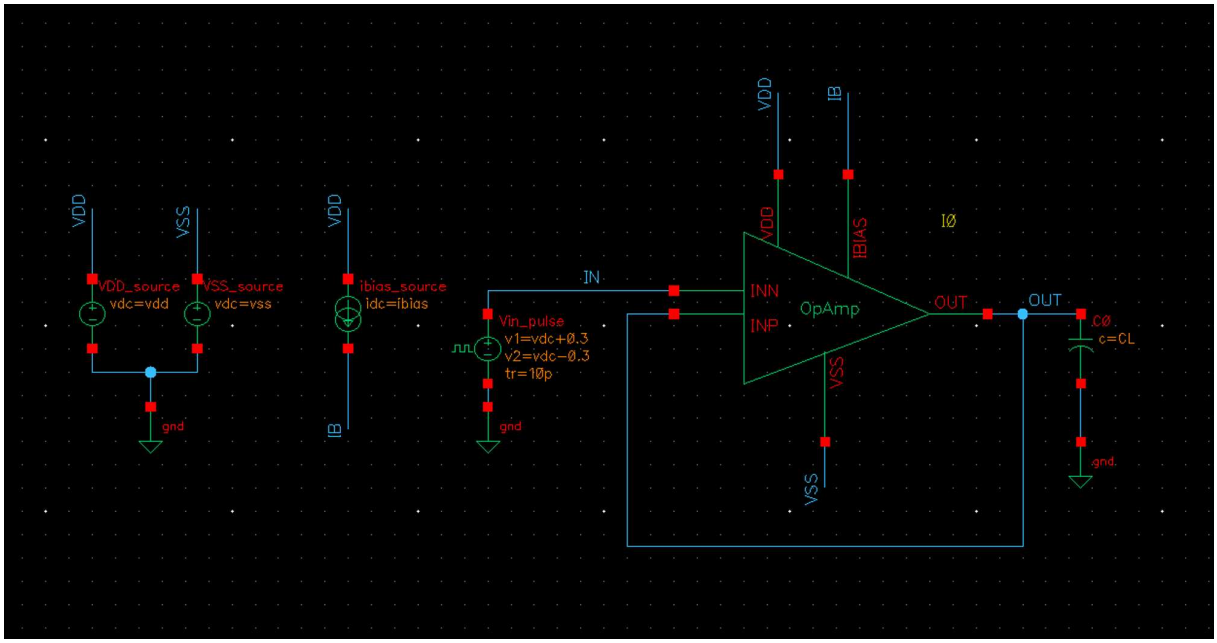
Nesta seção, estão descritas as atividades realizadas pelo estagiário durante a vigência do período correspondente no Laboratório de Sistemas Embarcados e Computação Pervasiva - Embedded.

3.1 RAMP UP NO FLUXO ANALÓGICO

A primeira parte do estágio foi composta por uma capacitação acerca do fluxo de desenvolvimento analógico. O objetivo final dessa etapa era se familiarizar com os possíveis formatos de arquivos utilizados no desenvolvimento AMS, ao mesmo tempo que entender como utilizar a ferramenta *Xrun* para simular tais *designs*.

Sendo assim, primeiramente o RAK *Basics of Analog Flow: A Design Oriented Approach* foi feito. Neste, a ferramenta *Virtuoso* é apresentada, e o estagiário implementou um amplificador operacional (*Opamp*), desde a fase de simulação até o início do *layout*. O RAK não foi finalizado pois a parte final de *layout* não era de interesse para o objetivo final do estágio. Na figura 1, é possível visualizar uma das vistas do *Virtuoso* utilizada para a simulação. Não é possível mostrar a implementação do amplificador porque a *Cadence* não permite a divulgação de seus materiais internos.

Figura 1 – Vista de simulação de transiente para o *Opamp*



Fonte: Autoria própria

Posteriormente, outro RAK foi feito, dessa vez o *Introduction to AMS Designer Simulation*. Este foi mais focado na simulação de circuitos AMS, tanto no *Virtuoso* quanto utilizando a ferramenta *Xrun*. Após esse RAK, o estagiário descobriu que o *Xrun* possui uma compatibilidade com os vários formatos de circuitos analógicos, desde arquivos *.vams* até arquivos SPECTRE (Um formato proprietário semelhante ao SPICE, porém com algumas diferenças de sintaxe).

Durante o estágio, os *designs* utilizados foram ambos escritos em *Verilog-AMS*. Porém, um deles utiliza apenas o formato *wreal*, estrutura de dados que é facilmente convertida em *real* e lida em *SystemVerilog*. O outro *design* utiliza de estrutura de dados *electrical*, e logo precisam de um arquivo de configuração analógica para serem simulados. Esse arquivo é passado para o *Xrun* com a *flag* `-analogcontrol file.scs`.

3.1.1 DIFICULDADES ENCONTRADAS

Durante o *ramp up*, as principais dificuldades enfrentadas foram relacionadas ao ferramental utilizado. O estagiário teve que instalar em sua máquina o sistema operacional (SO) *Red Hat*, mais especificamente o RHEL 7. O SO escolhido foi este devido ao fato de que o estagiário não possuia em mãos um disco rígido com espaço suficiente para armazenar a ISO do RHEL 8. Entretanto, as ferramentas utilizadas possuem suporte para a versão 7, logo o desenvolvimento dos trabalhos durante o estágio não foi afetado. Posteriormente, todas as ferramentas da *Cadence* necessárias foram instaladas. Entretanto, por se tratar de uma instalação de um SO recém instalado, algumas bibliotecas necessárias só foram instaladas após as ferramentas começarem a não funcionar da maneira correta. Sendo assim, uma parte do tempo de *ramp up* foi convertido em tempo para configuração do ambiente de trabalho. O principal erro foi a não instalação do *Xterm*, ferramenta necessária para configurar todas as ferramentas da *Cadence*. Após instalá-la, não houveram maiores dificuldades durante o aprendizado.

3.2 INVESTIGAÇÃO DO CENÁRIO ATUAL NO MUNDO

Durante a segunda etapa do estágio, buscou-se entender como está a situação de verificação de blocos mistos utilizando UVM. Primeiro, o estagiário investigou uma apresentação da *Accelera* de 2021 (UVM-AMS Working Group, 2021). Nesta, o grupo de trabalho UVM-AMS mostrou o progresso feito até então na criação de um padrão de verificação de circuitos mistos utilizando UVM. Assim, a visão que se tinha na época era que deveria existir um *Agent* exclusivo para realizar os estímulos analógicos, sendo esses estímulos passados para uma API que iria converter os valores para os formatos dos *designs* mistos. Com isso, existiria dentro do topo do *testbench* um ambiente específico para o DUT, no qual haveriam blocos responsáveis por converter sinais SPICE, sinais

Verilog-AMS e sinais gerados por modelagem de números reais. Até então, só havia um conjunto de regras que os componentes UVM seguiriam quando o padrão viesse a estar finalizado. Não havia nenhum exemplo de como seria feito.

Por outro lado, algumas empresas contam com uma solução proprietária para integrar os circuitos AMS a ambientes UVM. É o caso da *Scientific analog*, que utiliza uma ferramenta chamada *XMODEL* para integrar sinais de controle gerados em *Systemverilog* às interfaces analógicas do DUT (JAEHA; DANčAK, 2023). Além disso, a própria *Cadence* apresenta alguma soluções para esse tipo de verificação, como a apresenta durante a *DVCon* de 2015 (BRENNAN et al., 2015). Nessa época, anterior à criação do grupo de trabalho UVM-AMS, a solução proposta pela *Cadence* foi a utilização de *Gaskets*, que em uma tradução livre, seria algo como "juntas", ou "gaxeta". A ideia seria que o ambiente de verificação gerasse sinais de controle e essa estrutura convertesse os sinais para o DUT analógico.

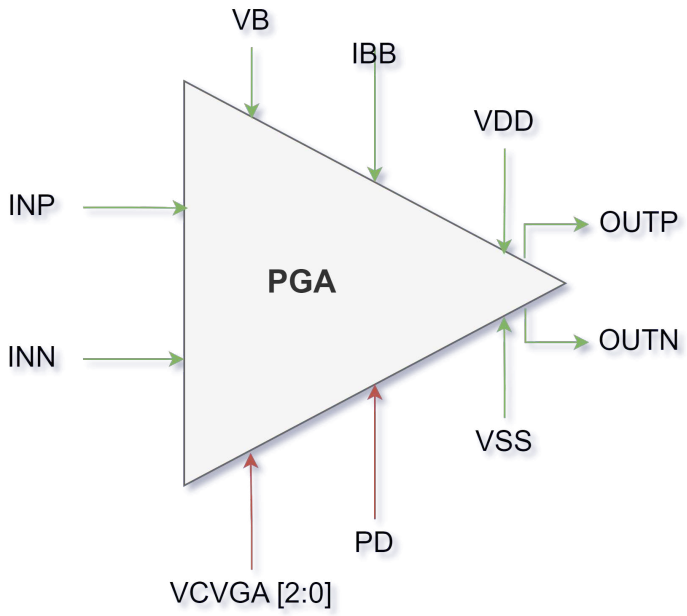
Somente em 2022, durante a *DVCon* daquele ano, o grupo de trabalho UVM-AMS publicou uma atualização do padrão em desenvolvimento (UVM-AMS Working Group, 2022). Esse *Webinar* foi utilizado pelo estagiário como base para a implementação dos ambientes de verificação nas etapas posteriores do estágio. Neste, o grupo de trabalho apresentou exemplos com trechos de códigos das principais unidades de conversão entre o mundo digital e analógico, o que ajudou o estagiário durante o desenvolvimento dos ambientes. Além disso, a ideia de que o *Agent* analógico geraria apenas os sinais de controle continua. O que muda é apenas a terminologia utilizada para os blocos de conversão. Desse modo, o *Agent* analógico não irá se comunicar com o DUT através de uma *interface*, como ocorre normalmente em um ambiente UVM, mas sim através de um *proxy*, enviando os sinais de controle para um novo bloco chamado *MS Bridge*. Este bloco instanciará o *Proxy* e uma outra estrutura, o *Analog Resource*, que será a unidade de conversão entre o digital e o analógico, sendo esse último ligado diretamente à interface analógica do DUT. Os sinais digitais, por outro lado, são manipulados como sempre, com a exceção de que a interface é instanciada também dentro do *MS Bridge*. Por fim, todos esses novos blocos são instanciados no *testbench* do ambiente de verificação.

Em síntese, percebe-se que por mais que as empresas possuam soluções próprias para lidarem com a verificação de circuitos mistos utilizando UVM, em todas a solução envolve a implementação de alguma ponte entre o mundo digital, no ambiente de verificação, e o mundo analógico, dentro do DUT. Sendo assim, o padrão em desenvolvimento não irá mudar essa abordagem, mas apenas irá oferecer ferramentas que garantirão a reusabilidade e e uma melhor integração entre esses dois mundos, mantendo as características já existentes da metodologia UVM.

3.3 IMPLEMENTAÇÃO DO PRIMEIRO AMBIENTE UVM

Após as duas primeiras fases do estágio, o estagiário iniciou o processo de implementação dos ambientes UVM. Para isso, utilizou como DUT de referência um PGA (do inglês: *Programmable Gain Amplifier*) oferecido em um dos *Training Bytes* da *Cadence*. Esse PGA foi escrito em *Verilog-AMS* utilizando apenas sinais no formato *wreal*, e os detalhes de implementação desse *design* não podem ser mostrados. Entretanto, o cabeçalho do módulo foi divulgado no *Webinar* de atualização do padrão UVM-AMS (UVM-AMS Working Group, 2022), logo não há problemas em mostrar a macroarquitetura desse *design*. Sendo assim, na figura 2 podem ser vistos todos os sinais que compõem a interface desse módulo, assim como uma descrição de cada um pode ser lida na tabela 1.

Figura 2 – Macroarquitetura do PGA



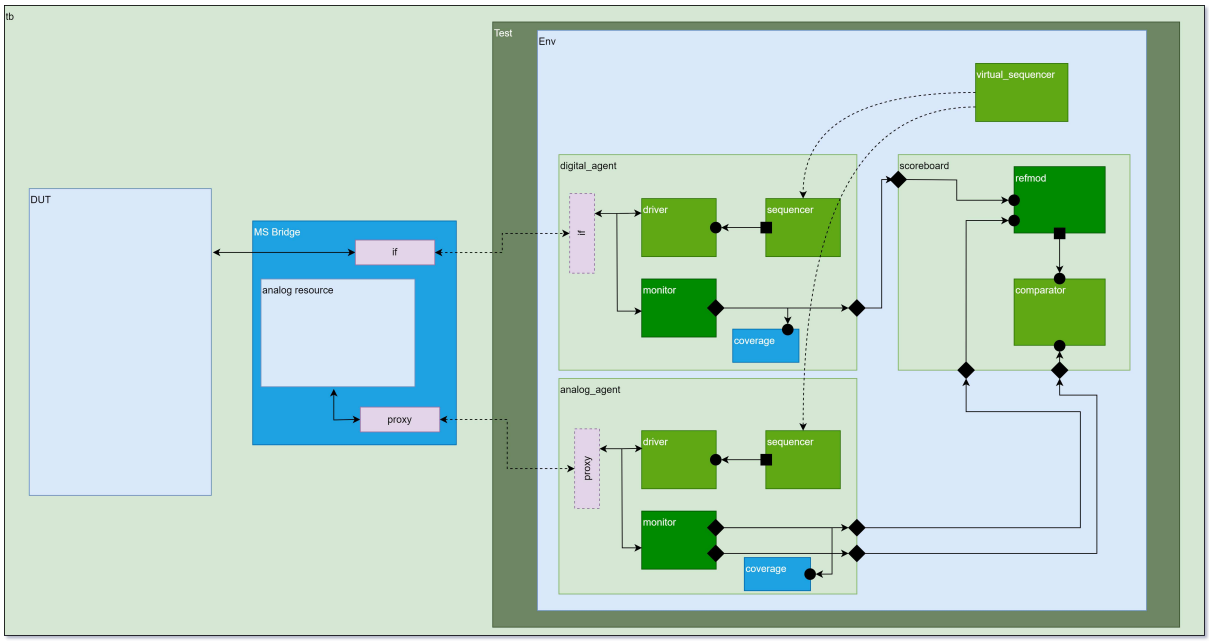
Fonte: Autoria própria

Tabela 1 – Descrição dos sinais do PGA

Sinal	Domínio	Descrição
VB	Analógico	Tensão de polarização.
IBB	Analógico	Corrente de polarização de entrada.
VDD	Analógico	Tensão de alimentação positiva.
VSS	Analógico	Tensão de alimentação negativa.
INP	Analógico	Entrada positiva do amplificador.
INN	Analógico	Entrada negativa do amplificador.
OUTP	Analógico	Saída diferencial positiva.
OUTN	Analógico	Saída diferencial negativa.
VCVGA	Digital	Controle do ganho. Sinal de 3 bits.
PD	Digital	Sinal de desligamento. Nível alto indica desligado.

Desse modo, o ambiente de verificação mostrado na figura 3 foi desenvolvido. As próximas seções irão detalhar cada componente desse ambiente.

Figura 3 – Diagrama UVM para a verificação do PGA



Fonte: Autoria própria

3.3.1 *AGENT* DIGITAL

Essa seção irá detalhar a implementação dos componentes do *agent* digital, ou seja, o *agent* responsável por lidar com os sinais **VCVGA** e **PD**.

3.3.1.1 TRANSAÇÃO

O arquivo de transação, nomeado `pga_d_packet.svh` não apresentam nenhuma particularidade. Apresenta apenas dois sinais aleatórios, um para o **VCVGA** e outro para o **PD**, além de inserir esses sinais na *factory*. Além disso, uma *constraint* foi implementada para que o sinal de **PD** fosse sempre zero por padrão, fazendo o bloco sempre estar ligado caso a sequência não informasse o contrário. Abaixo é possível ver o código implementado para esse componente.

```
class pga_d_packet extends uvm_sequence_item;

    rand bit [2:0] vcvga;
    rand bit pd;

    `uvm_object_utils_begin(pga_d_packet)
        `uvm_field_int(vcvga, UVM_ALL_ON)
        `uvm_field_int(pd, UVM_ALL_ON)
    `uvm_object_utils_end

    constraint pd_always_on{
        pd == 1'b0;
    }


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

endclass : pga_d_packet
```

3.3.1.2 *DRIVER*

O arquivo do *driver* foi nomeado `pga_d_driver.svh`, e a única diferença em relação a grande parte dos *drivers* para circuitos digitais é que nesse caso não existe um *handshaking* ou sinal de controle que sinalize um momento de estímulo do DUT. Sendo assim, uma variável `ts` foi implementada. Essa variável é passada por linha de comando para o ambiente em tempo de simulação, e representa o tempo em *ns* que o *driver* esperará antes de estimular novamente o DUT.

```
class pga_d_driver extends uvm_driver#(pga_d_packet);
    `uvm_component_utils(pga_d_driver)

    pga_d_vif vif;
    real ts;

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

    extern function void build_phase(uvm_phase phase);

    extern task main_phase(uvm_phase phase);

endclass : pga_d_driver

function void pga_d_driver::build_phase(uvm_phase phase);
    super.build_phase(phase);
endfunction : build_phase


task pga_d_driver::main_phase(uvm_phase phase);
    vif.PD <= 1'b0;
```

```
24      #10 // Tempo arbitrario para iniciar os estimulos
25      forever begin
26          seq_item_port.get_next_item(req);
27          begin_tr(req, "digital_driver");
28          vif.VCVGA <= req.vcvga;
29          vif.PD <= req.pd;
30          #(ts * 1ns)
31          end_tr(req);
32          seq_item_port.item_done();
33      end
34  endtask : main_phase
```

3.3.1.3 *MONITOR*

O *monitor* foi implementado em um arquivo nomeado `pga_d_monitor.svh`. Esse componente, diferente do *driver*, não precisou de nenhum tempo de referência para observar o DUT. Assim, por escolha do estagiário, o *monitor* toma como referência a mudança no sinal de entrada **VCVGA** para escrever uma transação na sua porta de saída. Com isso, toda vez que o *driver* envia um sinal de controle para o DUT, esse componente observa e escreve esse sinal em uma transação, enviando-a para o restante do ambiente. O código completo do *monitor* digital pode ser visto abaixo:

```
1   class pga_d_monitor extends uvm_monitor;
2       `uvm_component_utils(pga_d_monitor)
3
4       pga_d_packet req_pkt;
5       uvm_analysis_port#(pga_d_packet) req_port;
6
7       pga_d_vif vif;
8
9       function new(string name="pga_d_monitor", uvm_component parent);
10          super.new(.name(name), .parent(parent));
11          req_port = new("req_port", this);
12      endfunction
13
14      extern function void build_phase(uvm_phase phase);
15
16      extern task main_phase(uvm_phase phase);
17  endclass : pga_d_monitor
18
19  function void pga_d_monitor::build_phase(uvm_phase phase);
20      super.build_phase(phase);
21      req_pkt = pga_d_packet::type_id::create(.name("req_pkt"), .contxt(
            get_full_name()));
22  endfunction : build_phase
23
24  task pga_d_monitor::main_phase(uvm_phase phase);
25      forever begin
26          @(vif.VCVGA);
27          begin_tr(req_pkt, "digital_monitor");
28          req_pkt.vcvga = vif.VCVGA;
29          req_pkt.pd = vif.PD;
30          req_port.write(req_pkt);
```

```
31          end_tr(req_pkt);
32      end
33  endtask : main_phase
```

3.3.1.4 COBERTURA

A cobertura dos sinais digitais foi implementada utilizando um *Subscriber*, componente UVM que possui uma porta *analysis_export* intrínseca, facilitando sua conexão com o *monitor*. Desse modo, toda vez que o *monitor* escreve uma transação em sua porta de saída, o *subscriber* executa a função `sample()`, que atualiza a cobertura. O código completo desse componente pode ser visto abaixo:

```
1  class pga_d_coverage extends uvm_subscriber #(pga_d_packet);
2      `uvm_component_utils(pga_d_coverage)
3
4      pga_d_packet d_pkt;
5
6      covergroup digital_gain_cg;
7          option.per_instance = 1;
8          option.at_least = 20;
9
10         gain : coverpoint d_pkt.vcvga;
11     endgroup : digital_gain_cg
12
13     function new(string name="pga_d_coverage", uvm_component parent);
14         super.new(.name(name), .parent(parent));
15         digital_gain_cg = new();
16     endfunction : new
17
18     extern virtual function void write(pga_d_packet t);
19
20 endclass : pga_d_coverage
21
22 function void pga_d_coverage::write(pga_d_packet t);
23     d_pkt = pga_d_packet::type_id::create(.name("d_pkt"), .contxt(get_full_name(
           )));
24     d_pkt.copy(t);
25     digital_gain_cg.sample();
26 endfunction :write
```

Os valores de *hits* e *bins* foram escolhidos sem nenhuma preocupação com a eficiência da verificação, visto que o foco do estágio foi integrar o ambiente UVM a um DUT misto, e não verificar de fato o PGA em questão.

3.3.1.5 *INTERFACE*

A interface de comunicação com o DUT apresenta os mesmos dois sinais implementados na transação. Dessa forma, é um arquivo com poucas linhas cujo código pode ser visualizado abaixo:

```
1  interface pga_d_if;
```

```
2       logic [2:0] VCVGA;
3       logic PD;
4
5       modport mst(output VCVGA, PD);
6       modport slv(input VCVGA, PD);
7
8   endinterface : pga_d_if
```

3.3.1.6 *AGENT*

Por fim, o estagiário implementou o *Agent* digital para finalizar a implementação dessa primeira parte do ambiente. O código do *sequencer* não será mostrado pois ele é apenas uma instanciação do *sequencer* padrão que a metodologia UVM oferece. No *agent*, todos os componentes citados até então são instanciados, criados e adicionados à *factory*. Além disso, as portas de cada componente são conectadas devidamente durante a *connect_phase*, enquanto que a interface virtual é conectada por *config_db* à interface no *tesbench*.

```
1   class pga_d_agent extends uvm_agent;
2       `uvm_component_utils(pga_d_agent)
3
4       uvm_analysis_port#(pga_d_packet) d_agt_port;
5
6       pga_d_driver pga_drv;
7       pga_d_monitor pga_mon;
8       pga_d_sequencer pga_sqr;
9       pga_d_coverage pga_d_cov;
10
11      pga_d_vif vif;
12
13      real ts_drv;
14
15      function new(string name="pga_d_agent", uvm_component parent);
16          super.new(.name(name), .parent(parent));
17          d_agt_port = new("d_agt_port", this);
18      endfunction : new
19
20      extern function void build_phase(uvm_phase phase);
21
22      extern function void connect_phase(uvm_phase phase);
23
24  endclass : pga_d_agent
25
26  function void pga_d_agent::build_phase(uvm_phase phase);
27      super.build_phase(phase);
28      assert(uvm_config_db#(pga_d_vif)::get(this, "", "d_vif", vif))
29          else `uvm_fatal(get_type_name(), "Failed to get digital virtual
                interface")
30
31      pga_drv = pga_d_driver::type_id::create(.name("pga_drv"), .parent(this));
32      pga_mon = pga_d_monitor::type_id::create(.name("pga_mon"), .parent(this));
33      pga_d_cov = pga_d_coverage::type_id::create(.name("pga_d_cov"), .parent(this
            ));
```

```
34
35      pga_sqr = pga_d_sequencer::type_id::create(.name("pga_sqr"), .parent(this));
36
37      pga_drv.vif = vif;
38      pga_mon.vif = vif;
39
40      pga_drv.ts = ts_drv;
41  endfunction : build_phase
42
43  function void pga_d_agent::connect_phase(uvm_phase phase);
44      super.connect_phase(phase);
45      pga_drv.seq_item_port.connect(pga_sqr.seq_item_export);
46      pga_mon.req_port.connect(d_agt_port);
47      pga_mon.req_port.connect(pga_d_cov.analysis_export);
48  endfunction : connect_phase
```

3.3.2 *AGENT* ANALÓGICO

Implementado o *agent* digital, o estagiário implementou então o equivalente analógico, responsável pelos estímulos dos sinais de controle que irão ser enviados pelo *proxy* para o módulo *MS Bridge*. Desse modo, essa seção irá percorrer os mesmos componentes feitos para o *agent* digital, porém agora lidando com os sinais analógicos.

3.3.2.1 TRANSAÇÃO

A transação do sinais analógicos foi um pouco mais extensa. Como sinais de controle para a entrada do DUT, de maneira arbitrária escolheu-se um sinal senoidal para estímulo. Sendo assim, quatro variáveis foram definidas: **frequência**, **nível DC**, **amplitude** e **fase**. Esses quatro sinais definem uma forma de onda senoidal. Além deles, também foram definidas as variáveis de controle de alimentação, que tem como principal objetivo fornecer os valores de tensão e corrente que mantém o DUT funcionando.

Ademais, algumas *constraints* foram implementadas que mantém esses sinais de controle de alimentação em limites que mantém o DUT ligado. Essas *constraints* não podem ser visualizadas no código por detalharem o funcionamento interno do *design*.

Por fim, a grande diferença dessa transação está na função `do_compare()`. Essa função sobrescreve a comparação padrão de transações fornecida pelo UVM. Com isso, uma tolerância foi definida, e a comparação é feita com base no valor aproximado das transações, uma vez que a amplitude é real e é inviável comparar com exatidão dois valores reais. Nesse caso, optou-se por comparar com base em uma tolerância em casas decimais, sendo 4 o valor padrão.

```
1  class pga_a_packet extends uvm_sequence_item;
2
3      // General waveform parameters
4      rand real frequency;
5      rand real bias;
```

```
6       rand real amplitude;
7       rand real phase;
8
9       // Power control
10      rand real vdd;
11      rand real vss;
12      rand real bias_voltage;
13      rand real bias_current;
14
15      // Tolerance (In decimal houses)
16      int TOLERANCE = 4;
17
18      `uvm_object_utils_begin(pga_a_packet)
19          `uvm_field_real(frequency, UVM_NOCOMPARE)
20          `uvm_field_real(bias, UVM_NOCOMPARE)
21          `uvm_field_real(amplitude, UVM_NOCOMPARE)
22          `uvm_field_real(phase, UVM_NOCOMPARE)
23          `uvm_field_real(vdd, UVM_NOCOMPARE)
24          `uvm_field_real(vss, UVM_NOCOMPARE)
25          `uvm_field_real(bias_voltage, UVM_NOCOMPARE)
26          `uvm_field_real(bias_current, UVM_NOCOMPARE)
27          `uvm_field_real(frequency, UVM_NOCOMPARE)
28      `uvm_object_utils_end
29
30
31      // CONSTRAINTS DEFINITIONS - CLASSIFIED INFORMATION
32
33      function new(string name="pga_a_packet");
34          super.new(.name(name));
35      endfunction
36
37      extern virtual function bit do_compare(uvm_object rhs, uvm_comparer comparer
            );
38
39  endclass : pga_a_packet
40
41  function bit pga_a_packet::do_compare(uvm_object rhs, uvm_comparer comparer);
42      bit res;
43      real epsilon = $pow(10, -TOLERANCE);
44      pga_a_packet _obj;
45
46      $cast(_obj, rhs);
47
48      res = super.do_compare(_obj, comparer) &&
49      ((amplitude - _obj.amplitude) < epsilon) &&
50      ((amplitude - _obj.amplitude) > -epsilon);
51
52      return res;
53  endfunction : do_compare
```

3.3.2.2 *DRIVER*

Analogamente ao *driver* digital, este também precisou de uma variável de controle para lidar com o tempo entre estímulos. Essa variável também é controlada por linha de

comando em tempo de simulação. Os pontos de interesse que merecem ser mencionados para esse componente é a instanciação do *proxy* ao invés da *interface* virtual. Além disso, também é possível observar a chamada de duas funções específicas desse novo componente: `set_wave_parameters()` e `set_power_parameters()`. Essas funções são responsáveis por enviar os sinais de controle do *proxy* para o *analog resource*. Dessa forma, estimulando o DUT com uma nova configuração.

```
1  class pga_a_driver extends uvm_driver#(pga_a_packet);
2      `uvm_component_utils(pga_a_driver)
3
4      pga_bridge_proxy proxy;
5      real ts;
6
7      function new(string name="pga_a_driver", uvm_component parent);
8          super.new(.name(name), .parent(parent));
9      endfunction : new
10
11     extern function void build_phase(uvm_phase phase);
12
13     extern task main_phase(uvm_phase phase);
14
15 endclass : pga_a_driver
16
17 function void pga_a_driver::build_phase(uvm_phase phase);
18     super.build_phase(phase);
19
20 endfunction : build_phase
21
22 task pga_a_driver::main_phase(uvm_phase phase);
23     #10
24     forever begin
25         seq_item_port.get_next_item(req);
26         begin_tr(req, "analog_driver");
27         // Power supply
28         proxy.vdd = req.vdd;
29         proxy.vss = req.vss;
30         proxy.ibb = req.bias_current;
31         proxy.vb = req.bias_voltage;
32         proxy.set_power_parameters();
33
34         // Waveform
35         proxy.wave_type = req.wave_type;
36         proxy.ampl_in = req.amplitude;
37         proxy.freq = req.frequency;
38         proxy.pha = req.phase;
39         proxy.duty = req.duty_cycle;
40         proxy.bias = req.bias;
41         proxy.set_wave_parameters();
42
43         #(ts * 1ns)
44         end_tr(req);
45         seq_item_port.item_done();
46     end
47
48 endtask : main_phase
```

3.3.2.3 *MONITOR*

Por escolha do estagiário, optou-se implementar um único *monitor* para observar os sinais de entrada e saída analógicos. Em um cenário de verificação real, seria preferível separar em dois para que o código ficasse modularizado, porém para fins de simplicidade, essa abordagem foi feita. Diferentemente do *monitor* digital, esse componente não tem como realizar as escritas de transação com base na variação de alguma entrada analógica, visto que estas variam constantemente. Logo, uma variável de controle também foi implementada para controlar esse tempo de amostragem. Sendo assim, o *monitor* captura a cada intervalo de amostragem tanto as entradas analógicas, enviando-as para o restante do ambiente, quanto as saídas analógicas (nesse caso, somente a amplitude de saída e as tensões de alimentação são importantes para a comparação), enviando-as também para o restante do ambiente. Além disso, nota-se que antes de atribuir os valores do *proxy* aos devidos campos da transação, as funções `get_wave_parameters()` e `get_power_parameters()` são chamadas. Essas funções tem como objetivo atribuir os valores atuais do DUT aos sinais do *proxy*.

Por fim, é importante explicitar que a escolha de dividir o tempo de amostragem colocando metade no início do procedimento e metade no final é apenas feita para uma melhor visualização da transação nas ferramentas de visualização de forma de onda, não impactando em nada na funcionalidade do *monitor*. Isso foi feito para evitar que houvesse uma sobreposição de transações na visualização devido ao fato de que a finalização de uma gravação de transação estava ocorrendo ao mesmo tempo que a próxima começava.

```
1  class pga_a_monitor extends uvm_monitor;
2      `uvm_component_utils(pga_a_monitor)
3
4      pga_a_packet rsp_pkt, req_pkt;
5      uvm_analysis_port#(pga_a_packet) rsp_port;
6      uvm_analysis_port#(pga_a_packet) req_port;
7      pga_bridge_proxy proxy;
8
9      real ts;
10
11     function new(string name="pga_a_monitor", uvm_component parent);
12         super.new(.name(name), .parent(parent));
13         rsp_port = new("rsp_port", this);
14         req_port = new("req_port", this);
15     endfunction
16
17     extern function void build_phase(uvm_phase phase);
18
19     extern task main_phase(uvm_phase phase);
20
21     extern task collect_analog_in();
22
23     extern task collect_analog_out();
24 endclass : pga_a_monitor
25
```

```
26
27 function void pga_a_monitor::build_phase(uvm_phase phase);
28     super.build_phase(phase);
29     rsp_pkt = pga_a_packet::type_id::create(.name("rsp_pkt"), .contxt(
           get_full_name()));
30     req_pkt = pga_a_packet::type_id::create(.name("req_pkt"), .contxt(
           get_full_name()));
31 endfunction : build_phase
32
33 task pga_a_monitor::main_phase(uvm_phase phase);
34     #10
35     forever begin
36         fork
37             collect_analog_in();
38             collect_analog_out();
39         join
40     end
41 endtask : main_phase
42
43 task pga_a_monitor::collect_analog_in();
44     #(ts * 0.5ns)
45     begin_tr(rsp_pkt, "analog_monitor_out");
46     proxy.get_wave_parameters();
47     rsp_pkt.amplitude = proxy.ampl_out;
48     rsp_port.write(rsp_pkt);
49     #(ts * 0.5ns)
50     end_tr(rsp_pkt);
51 endtask : collect_analog_in
52
53 task pga_a_monitor::collect_analog_out();
54     #(ts * 0.5ns)
55     begin_tr(req_pkt, "analog_monitor_in");
56     proxy.get_wave_parameters();
57     req_pkt.amplitude = proxy.ampl_in;
58     req_pkt.vdd = proxy.vdd;
59     req_pkt.vss = proxy.vss;
60     req_port.write(req_pkt);
61     #(ts * 0.5ns)
62     end_tr(req_pkt);
63 endtask : collect_analog_out
```

3.3.2.4 *PROXY*

Este foi o primeiro componente implementado que tem uma funcionalidade diretamente atrelada aos estímulos dos sinais analógicos. Entretanto, na escrita dessa classe, o estagiário apenas definiu os atributos e criou duas funções abstratas puras. Além disso, é possível ver que a própria classe é também uma classe virtual. Isso ocorre porque a conversão dos sinais *real* para *wreal* e vice-versa é feita dentro da *MS Bridge*. É lá que as funções declaradas aqui serão definidas atribuindo diretamente os valores do *proxy* aos valores do *analog resource*. Essa escolha de implementação foi feita por conta da impossibilidade de acessar os valores do *Analog Resource* diretamente desse ponto do ambiente. Assim, é possível aproveitar das funcionalidades de *Downcast* que a linguagem de *Sys-*

temverilog oferece. Desse modo, no *Analog Resource* essa classe é redefinida e as funções são implementadas de fato, e ao passar o *proxy* por *config_db* para o *driver* e *monitor* analógicos, o *proxy* genérico instanciado nesses componentes passa a se comportar como o *proxy* filho definido anteriormente, e a chamada das funções invoca de fato as funções redefinidas na classe filha.

```
virtual class pga_bridge_proxy;
    real ampl_in;
    real ampl_out;
    real freq;
    real pha;
    real bias;

    real vdd;
    real vss;
    real ibb;
    real vb;

    // Push functions
    pure virtual function void set_wave_parameters();
    pure virtual function void set_power_parameters();

    // Pull functions
    pure virtual function void get_wave_parameters();
    pure virtual function void get_power_parameters();

endclass : pga_bridge_proxy
```

3.3.2.5 COBERTURA

A cobertura foi feita de forma análoga ao que foi implementado na parte digital do ambiente. Entretanto, a única diferença está na criação dos *coverpoints*. As ferramentas da *Cadence* permitem que valores reais sejam cobertos desde que seja informado o valor do passo a ser considerado entre um *bin* e outro. Dessa forma, não houve necessidade de uma conversão dos valores reais observados pelo *monitor* analógico em valores digitais para que fossem cobertos. Entretanto, essa conversão seria necessária caso uma ferramenta que não suporta tal abordagem fosse utilizada.

Desse modo, na implementação da cobertura, o estagiário optou por utilizar como exemplo o sinal de frequência da entrada para cobertura, não sendo essa decisão voltada à verificação do *design*, e sim para fins didáticos.

```
class pga_a_coverage extends uvm_subscriber #(pga_a_packet);
    `uvm_component_utils(pga_a_coverage)

    pga_a_packet a_pkt;

    covergroup analog_cg;
        option.per_instance = 1;
        option.at_least = 20;
        frequency : coverpoint a_pkt.frequency{
```

```
            type_option.real_interval = 1e3;
            bins range[10] = {[5e6:10e6]};
        }
    endgroup : analog_cg

    function new(string name="pga_a_coverage", uvm_component parent);
        super.new(.name(name), .parent(parent));
        analog_cg = new();
    endfunction : new

    extern virtual function void write(pga_a_packet t);
endclass : pga_a_coverage

function void pga_a_coverage::write(pga_a_packet t);
    a_pkt = pga_a_packet::type_id::create(.name("a_pkt"), .contxt(get_full_name(
        )));
    a_pkt.copy(t);
    analog_cg.sample();
endfunction :write
```

3.3.2.6 *AGENT*

Por fim, o *agent* analógico foi implementado. Este *agent* não tem nenhuma funcionalidade nova em comparação com o analógico, somente que ao invés de instanciar e conectar a *interface* virtual, este realiza esses processos com o *proxy*.

```
class pga_a_agent extends uvm_agent;
    `uvm_component_utils(pga_a_agent)

    uvm_analysis_port#(pga_a_packet) a_agt_port_out;
    uvm_analysis_port#(pga_a_packet) a_agt_port_in;

    pga_a_driver pga_drv;
    pga_a_monitor pga_mon;
    pga_a_sequencer pga_sqr;
    pga_a_coverage pga_a_cov;

    pga_bridge_proxy proxy;
    real ts_drv, ts_mon;

    function new(string name="pga_a_agent", uvm_component parent);
        super.new(.name(name), .parent(parent));
        a_agt_port_in = new("a_agt_port_in", this);
        a_agt_port_out = new("a_agt_port_out", this);
    endfunction : new

    extern function void build_phase(uvm_phase phase);

    extern function void connect_phase(uvm_phase phase);
endclass : pga_a_agent

function void pga_a_agent::build_phase(uvm_phase phase);
    super.build_phase(phase);

    assert(uvm_config_db#(pga_bridge_proxy)::get(this, "", "proxy", proxy))
```

```
30      else `uvm_fatal(get_type_name(), "Failed to get analog proxy")
31
32      pga_drv = pga_a_driver::type_id::create(.name("pga_drv"), .parent(this));
33      pga_mon = pga_a_monitor::type_id::create(.name("pga_mon"), .parent(this));
34
35      pga_sqr = pga_a_sequencer::type_id::create(.name("pga_sqr"), .parent(this));
36      pga_a_cov = pga_a_coverage::type_id::create(.name("pga_a_cov"), .parent(this
            ));
37
38      pga_drv.proxy = proxy;
39      pga_mon.proxy = proxy;
40
41      pga_drv.ts = ts_drv;
42      pga_mon.ts = ts_mon;
43  endfunction : build_phase
44
45  function void pga_a_agent::connect_phase(uvm_phase phase);
46      super.connect_phase(phase);
47
48      pga_drv.seq_item_port.connect(pga_sqr.seq_item_export);
49      pga_mon.req_port.connect(a_agt_port_in);
50      pga_mon.rsp_port.connect(a_agt_port_out);
51      pga_mon.req_port.connect(pga_a_cov.analysis_export);
52  endfunction : connect_phase
```

3.3.3 *MS BRIDGE*

Após criar os dois *agents*, o estagiário implementou o *MS Bridge*. Para isso, dois arquivos foram implementados: o *Analog Resource*, escrito em *.vams*, responsável por estimular o DUT, e o *MS Bridge* em si, repsonsável por redefinir o *proxy*, instanciá-lo, instanciar a interface digital e o *Analog Resource*.

Desse modo, primeiramente foi implementado o *Analog resource*, também chamado de *core*. Este é um módulo em *.vams* que possui todos os sinais necessários para o estímulo do DUT. É possível observar que a interface desse módulo é idêntica à interface do DUT. Isso foi feito para seguir o padrão mencionado pelo grupo UVM-AMS. Entretanto, o estagiário não precisou conectar esses valores a nenhum outro fio durante a instanciação o *Analog resource*, apenas no *tesbench* foi atribuído esses valores diretamente ao DUT.

```
1   `timescale 1s/1ps
2
3   module pga_bridge_core(OUTP, OUTN, INP, INN, VCVGA, VB, IBB, VDD, VSS, PD);
4
5       output OUTP, OUTN, INP, INN;
6       input [2:0] VCVGA;
7       input VB, IBB;
8       inout VDD, VSS;
9       input PD;
10
11      wreal OUTP, OUTN, INP, INN, VB, IBB, VDD, VSS;
12      wreal vsin;
13
```

```
14      real a_in;
15      real bias_in;
16      real freq_in;
17      real pha_in;
18      real duty_in;
19
20      real VDD_in;
21      real VSS_in;
22      real IBB_in;
23      real VB_in;
24
25      wreal ampl_in;
26      wreal ampl_out;
27
28      wreal VDD_out;
29      wreal VSS_out;
30      wreal IBB_out;
31      wreal VB_out;
32
33      real VINdif;
34
35      initial begin
36          freq_in = 10e6;
37          a_in = 0;
38          pha_in = 0;
39          bias_in = 0;
40      end
41
42      always begin
43          #100p
44          VINdif = a_in*sin(2*3.14159265*freq_in*$abstime + pha_in);
45      end
46
47      // Atribuicao dos valores calculados as entradas analogicas
48      assign INN = bias_in - VINdif/2;
49      assign INP = bias_in + VINdif/2;
50
51      assign VDD = VDD_in;
52      assign VSS = VSS_in;
53      assign VB = VB_in;
54      assign IBB = IBB_in;
55
56      // Atribuicao da amplitude que sera vista pelo monitor analogico
57      assign ampl_out = (OUTP - OUTN);
58      assign ampl_in = (INP - INN);
59
60  endmodule
```

Finalmente, no *MS bridge*, é possível visualizar a redefinição do *proxy*, executando assim a conversão entre os valores oriundos do ambiente de verificação e os valores utilizados pelo *Analog resource*.

```
1   module pga_bridge();
2       import pga_pkg::*;
3       pga_d_if d_if();
4
```

```
5       class pga_proxy extends pga_bridge_proxy;
6
7            // Push functions
8           function void set_wave_parameters();
9               core.a_in = ampl_in;
10              core.freq_in = freq;
11              core.pha_in = pha;
12              core.bias_in = bias;
13          endfunction : set_wave_parameters
14
15          function void set_power_parameters();
16              core.VDD_in = vdd;
17              core.VSS_in = vss;
18              core.IBB_in = ibb;
19              core.VB_in = vb;
20          endfunction : set_power_parameters
21
22           // Pull functions
23          function void get_wave_parameters();
24              ampl_in = core.ampl_in;
25              ampl_out = core.ampl_out;
26              freq_in = core.freq;
27          endfunction : set_wave_parameters
28
29          function void get_power_parameters();
30              vdd = core.VDD_out;
31              vss = core.VSS_out;
32              ibb = core.IBB_out;
33              vb = core.VB_out;
34          endfunction : set_power_parameters
35
36      endclass : pga_proxy
37
38      pga_proxy proxy = new();
39
40      pga_bridge_core core();
41
42  endmodule : pga_bridge
```

3.3.4 DEMAIS COMPONENTES

O restante do ambiente de verificação não apresenta nenhuma particularidade que valha a pena ser mencionada. Os códigos-fonte do ambiente de verificação podem ser vistos no repositório pessoal do estagiário no *Github* (GOMES,).

3.3.5 SIMULAÇÃO

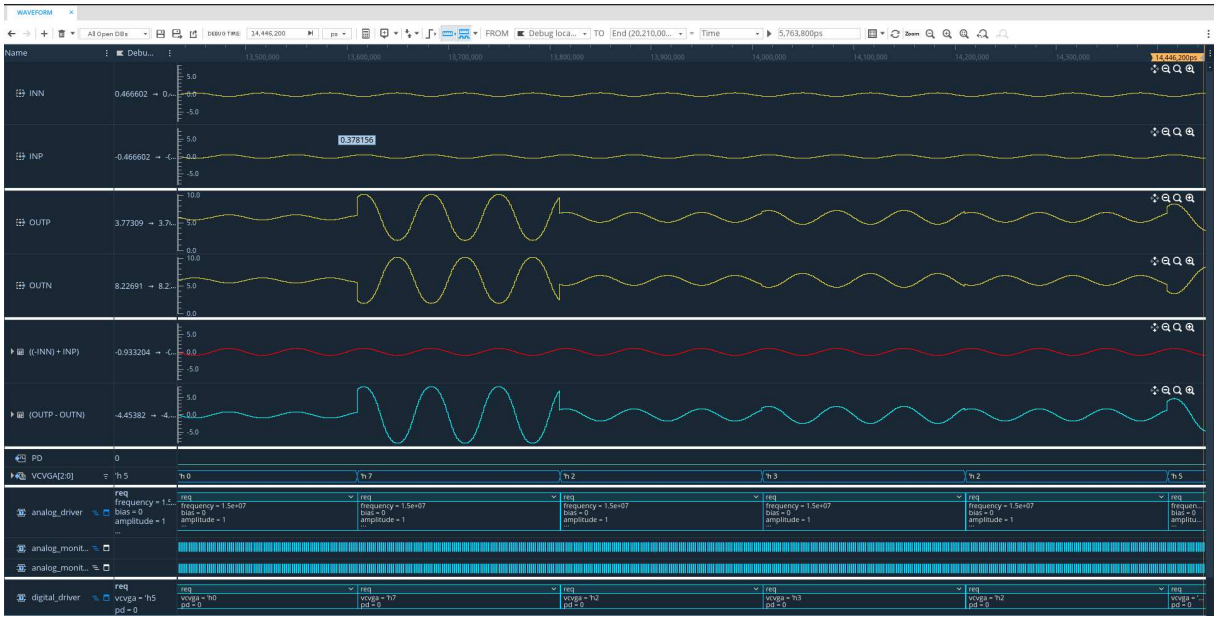
Sendo assim, finalizada a implementação do ambiente de verificação, simulações foram feitas cujos resultados podem ser observados nas figuras 4 e 5. Esta mostra uma visualização da forma de onda das entradas do DUT sendo estimuladas e as saídas sendo

geradas, enquanto aquela mostra o *log* do *Xrun* sinalizando que a simulação foi finalizada sem *missmatches*.

Figura 4 – *Log* do *Xrun* - Simulação do PGA

Fonte: Autoria própria

Figura 5 – Formas de ondas - PGA - *Verisium*



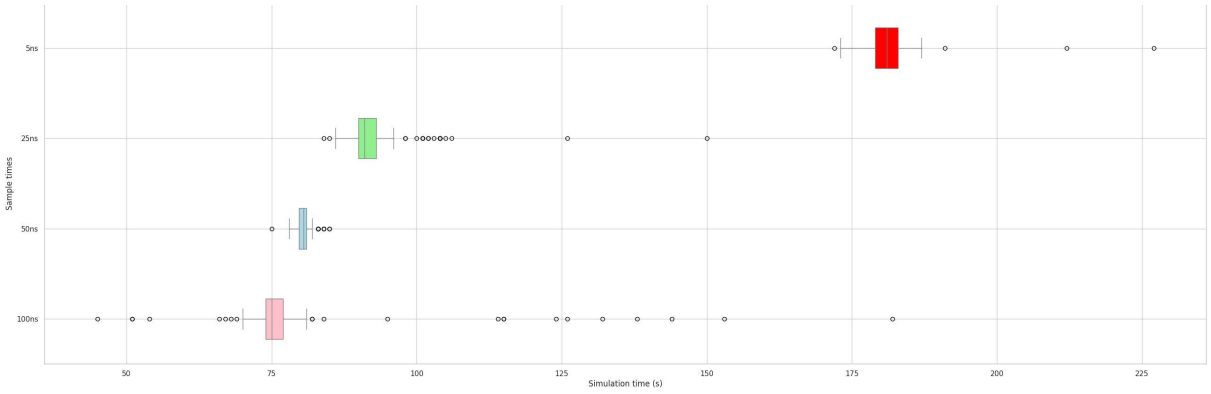
Fonte: Autoria própria

3.3.6 ANÁLISE PARA DIFERENTES TEMPOS DE AMOSTRAGEM

Além das simulações convencionais, o estagiário implementou também um arquivo de regressão *.vsif*, executando centenas de testes com variados tempos de amostragens para o *monitor* analógico. Isso basicamente controla a frequência com que as transações são

comparadas, e influencia diretamente no tempo de simulação. Desse modo, um *script* em *tcl* foi feito para manipular executar comandos do *Vmanager* que geram *reports* de tempo de simulação para cada teste, escrevendo os tempos para cada configuração de amostragem em um arquivo de texto distinto. Posteriormente, um programa em *Python* foi escrito para manipular esses arquivos e gerar um gráfico *BoxPlot* mostrando a diferença entre os tempos de simulações. Essa diferença pode ser vista na figura 6. Note que o tempo médio aumenta consideravelmente para valores menores do intervalo de amostragem. Desse modo, em um cenário de verificação real, seria necessária uma abordagem mais cuidadosa da equipe de verificação para decidir qual seria a frequência ideal de comparação para o DUT em questão, a fim de não escolherem uma taxa de comparação muito alta sem necessidade ou muito baixa e isso impactar na eficiência da verificação do módulo.

Figura 6 – Diferenças entre o tempo de simulação para intervalos de amostragem distintos



Fonte: Autoria própria

Por fim, para esse *Design* específico, não foram encontradas muitas dificuldades que bloqueassem o desenvolvimento. A única escolha que fez com que o ritmo fosse um pouco reduzido foi a mudança de ferramentas, deixando o *Simvision* e aderindo ao *Verisium* para a visualização das formas de ondas. Essa escolha foi feita para aproveitar o período de estágio e aprender a lidar com uma ferramenta mais atual que está começando a ser utilizada no mercado, além do *Verisium* ser mais fluido e apresentar uma interface de usuário mais amigável. A implementação dos componentes de manipulação dos sinais analógicos não apresentaram dificuldades, uma vez que exisitam exemplos tanto no *Webinar* já mencionado (UVM-AMS Working Group, 2022), quanto em *training bytes* da *Cadence* que ensinavam como implementar *testbenchs* rápidos para a simulação do PGA.

3.4 IMPLEMENTAÇÃO DO SEGUNDO AMBIENTE DE VERIFICAÇÃO

Finalizadas as análises para o PGA cujos dados eram apenas no formato *wreal*, o estagiário então criou um novo ambiente de verificação a partir do já implementado, e

o modificou para que ele agora fosse conectado ao *design* de um PGA de interface mais simples, extraído de um *Training Byte* da plataforma de suporte da *Cadence*. Este contém apenas dois sinais analógicos de entrada: **INN** e **INP**, e um sinal de saída, **OUT**. Esse *design* por sua vez apresenta um controle de ganho **GAIN** de 8 *bits*, permitindo assim uma variação de 256 níveis de ganho. Além disso, esse DUT é sensível à frequência de entrada, com sua saída dependendo do valor de frequência. Por fim, o pga em questão também apresenta sinais no formato *electrical*, o que tornou a conversão entre analógico e digital um pouco mais complicada.

Por outro lado, a arquitetura do ambiente de verificação continou exatamente a mesma, sendo necessárias alterações apenas em alguns arquivos que lidam com alguns arquivos em específico. Para o *agent* digital, apenas foram trocados os campos da transação, removendo o **VCVGA** e o **PD**, inserindo o novo campo **GAIN**. Essa alteração foi feita em todos os componentes desse *agent*, e como foram alterações de apenas uma ou duas linhas por arquivo, não serão mostrados os códigos para que o relatório em questão não fique muito extenso desnecessariamente.

Já no *agent* analógico, as mudanças feitas foram maiores, e valem a pena serem detalhadas.

3.4.1 *TRANSAÇÃO* DO *AGENT* ANALÓGICO

O PGA em questão possui uma modelagem mais complexa, e de caráter não ideal. Sendo assim, a comparação entre transações precisou ser reescrita, com a tolerância não mais representando casas decimais, mas sim um valor percentual em relação ao valor ideal de amplitude calculado pelo modelo de referência. Essa comparação se aproxima mais do que é exigido nas especificações de circuitos como esse, sendo sempre explicitada a tolerância do circuito nos *datasheets*.

```
1  class pga_a_packet extends uvm_sequence_item;
2
3      // General waveform parameters
4      rand real frequency;
5      rand real bias;
6      rand real amplitude;
7      rand real phase;
8
9      // Tolerance (In percentage)
10     int TOLERANCE = 10;
11
12     `uvm_object_utils_begin(pga_a_packet)
13         `uvm_field_real(frequency, UVM_NOCOMPARE)
14         `uvm_field_real(bias, UVM_NOCOMPARE)
15         `uvm_field_real(amplitude, UVM_NOCOMPARE)
16         `uvm_field_real(phase, UVM_NOCOMPARE)
17     `uvm_object_utils_end
18
19     constraint small_wave_values {
```

```
20          amplitude > -3;
21          amplitude < 3;
22          bias > -3 ;
23          bias < 3;
24          phase >= 0;
25          phase <= 2*3.14159265;
26      }
27
28      constraint high_frequencies {
29          frequency < 500e6;
30          frequency > 0.5e6;
31      }
32
33      function new(string name="pga_a_packet");
34          super.new(.name(name));
35      endfunction
36
37      extern virtual function bit do_compare(uvm_object rhs, uvm_comparer comparer
            );
38  endclass : pga_a_packet
39
40
41  function bit pga_a_packet::do_compare(uvm_object rhs, uvm_comparer comparer);
42      bit res;
43      real error;
44      pga_a_packet _obj;
45
46      $cast(_obj, rhs);
47
48      error = 100*(amplitude - _obj.amplitude)/_obj.amplitude;
49
50      res = super.do_compare(_obj, comparer) &&
51      (error < TOLERANCE) &&
52      (-error < TOLERANCE);
53
54      return res;
55  endfunction : do_compare
```

3.4.2 *PROXY* DO *AGENT* ANALÓGICO

Devido ao *design* apresentar estruturas de dados *electrical*, foi necessária a declaração no *proxy* de duas funções auxiliares que seriam responsáveis por obter as tensões de cada nó do circuito dentro do *Analog Resource*. Isso foi necessário uma vez que a atribuição direta `proxy.ampl_out = core.OUT` não ser possível.

```
1  virtual class pga_bridge_proxy;
2      real ampl_in;
3      real ampl_out;
4      real freq;
5      real pha;
6      real bias;
7
8      // Push functions
9      pure virtual function void set_wave_parameters();
```

```
10
11      // Pull functions
12      pure virtual function void get_wave_parameters();
13
14      // Auxiliar functions
15      pure virtual function real get_inn_voltage();
16      pure virtual function real get_out_voltage();
17  endclass : pga_bridge_proxy
```

3.4.3 *MS BRIDGE*

Nesse caso, o *Analog Resource* foi implementado em um arquivo menor, uma vez que a interface do PGA também é menor do que a do *design* anterior. É importante notar a diferença entre o estímulo da tensão de entrada para os dois *designs*. No primeiro, este foi feito dentro de um bloco `always`. Já aqui, um bloco `analog` foi utilizado, com atribuição da equação da forma de onda não diretamente à entrada **INP**, mas a tensão desse nó `V(INP)`. Além disso, a função `$bound_step` define um limite no tamanho do próximo passo de tempo utilizado pelo núcleo de simulação contínua. Ela não especifica qual será o próximo passo, mas sim o quão grande ele pode ser. No caso do código em questão, o tempo máximo entre cada passo é de 1/20 do período do sinal de entrada.

Além disso, também é necessário mencionar as funções auxiliares:

- `get_inn_voltage(input dummy)`;
- `get_out_voltage(input dummy)`.

Em ambas, o argumento *dummy* está presente apenas porque a linguagem *Verilog-AMS* não permite a criação de funções sem argumentos. Essas funções são as responsáveis por converter a diferença de tensão de um nó em um valor *real* que poderá ser lido pelo *proxy*.

```
1   `timescale 1s/1ps
2   `include "disciplines.vams"
3   `include "constants.vams"
4
5   module pga_bridge_core(INP, INN, GAIN, OUT);
6
7       output OUT, INP, INN;
8       input [7:0] GAIN;
9
10      electrical INN, INP, OUT;
11
12      real a_in;
13      real bias_in;
14      real freq_in;
15      real pha_in;
16
17      analog begin
18          V(INP) <+ a_in * sin(2*`M_TWO_PI*freq_in*$abstime);
```

```
19          V(INN) <+ 0;
20          $bound_step(1/(freq_in*20));
21      end
22
23      initial begin
24          freq_in = 10e6;
25          a_in = 0;
26          pha_in = 0;
27          bias_in = 0;
28      end
29
30      function real get_inn_voltage(input dummy);
31          get_inn_voltage = V(INP, INN);
32      endfunction
33
34      function real get_out_voltage(input dummy);
35          get_out_voltage = V(OUT);
36      endfunction
37  endmodule
```

Por fim, a *MS Bridge* foi implementada de forma análoga ao *design anterior*, com a única exceção da definição das duas funções auxiliares mencionadas anteriormente, que irão ser responsáveis por obter do *Analog Resource* as tensões de entrada e saída do DUT.

3.4.4 SIMULAÇÃO

Com as alterações no ambiente feitas, o estagiário então iniciou o processo de simulação para validar as decisões tomadas. Primeiramente, um cenário no qual o *design* funcionaria foi simulado, com uma frequência mais baixa ($1MHz$) e um ganho máximo de $100V/V$. Na figura 7 pode ser visto o *log* do *Xrun* informando que o teste obteve sucesso, enquanto que na figura 8 é possível observar a forma de onda. Para esse *design*, duas novas variáveis foram adicionadas à visualização: A tensão absoluta na saída removendo o nível DC e o ganho absoluto em V/V naquele instante de tempo.

Posteriormente, o estagiário alterou as sequências de modo que o *design* passasse a distorcer a tensão de entrada na saída devido ao não cumprimento de especificações. Com isso, para uma frequência de $100MHz$ e um ganho máximo de $100V/V$, os testes começam a falhar por ultrapassarem a tolerância de 10% definida na transação. Na figura 9 é possível observar o *log* do *Xrun* mostrando o teste falhando, enquanto que na figura 10 é possível visualizar as formas de onda. É notável a distorção do sinal para os maiores valores de ganho, apresentando um *overshoot* nos momentos de alteração de ganho, além de um atraso no sinal de saída em relação ao de entrada.

Dessa forma, alterando o ganho máximo para um valor de $1V/V$, o *design* volta a funcionar dentro das especificações para uma frequência alta assim, e novamente os testes passam. Na figura 11 é possível observar novamente o *log* de simulação acusando o sucesso do teste. Além disso, é possível ver que não há mais *overshoot* ou distorção do

Figura 7 – *Log* do *Xrun* - Simulação do 2^o PGA - $1MHz$

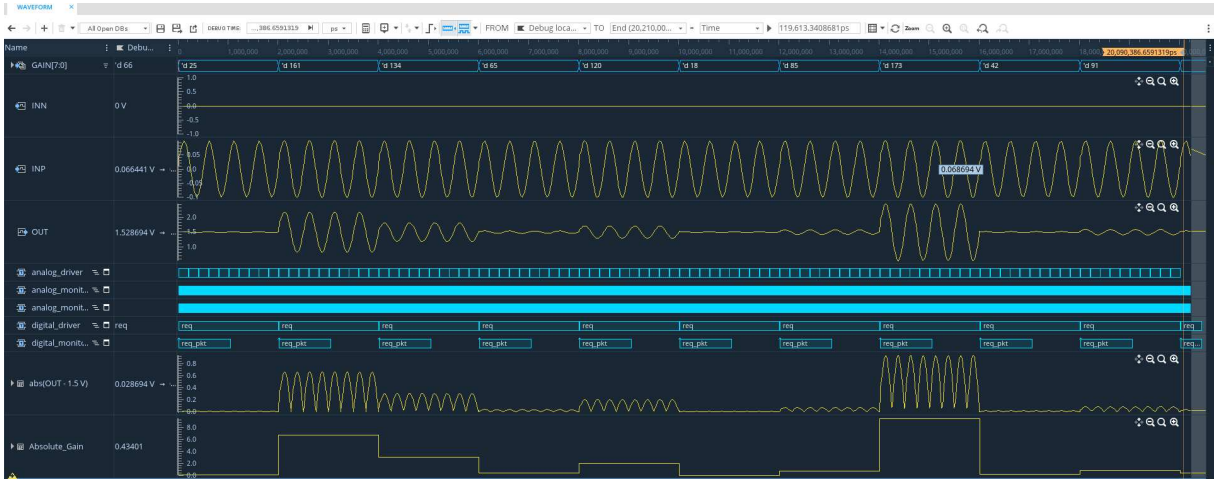
```
UVM_INFO ../src/env/pga_comparator.svh(82) @ 20210000: uvm_test_top.env.pga_scb.pga_comp [pga_comparator]
 - Total Match =       10100
 - Total Missmatch =          0

--- UVM Report catcher Summary ---

Number of demoted UVM_FATAL reports  :    0
Number of demoted UVM_ERROR reports  :    0
Number of demoted UVM_WARNING reports:    0
Number of caught UVM_FATAL reports   :    0
Number of caught UVM_ERROR reports   :    0
Number of caught UVM_WARNING reports :    0
```

Fonte: Autoria própria

Figura 8 – Formas de ondas - 2^o PGA - $1MHz$



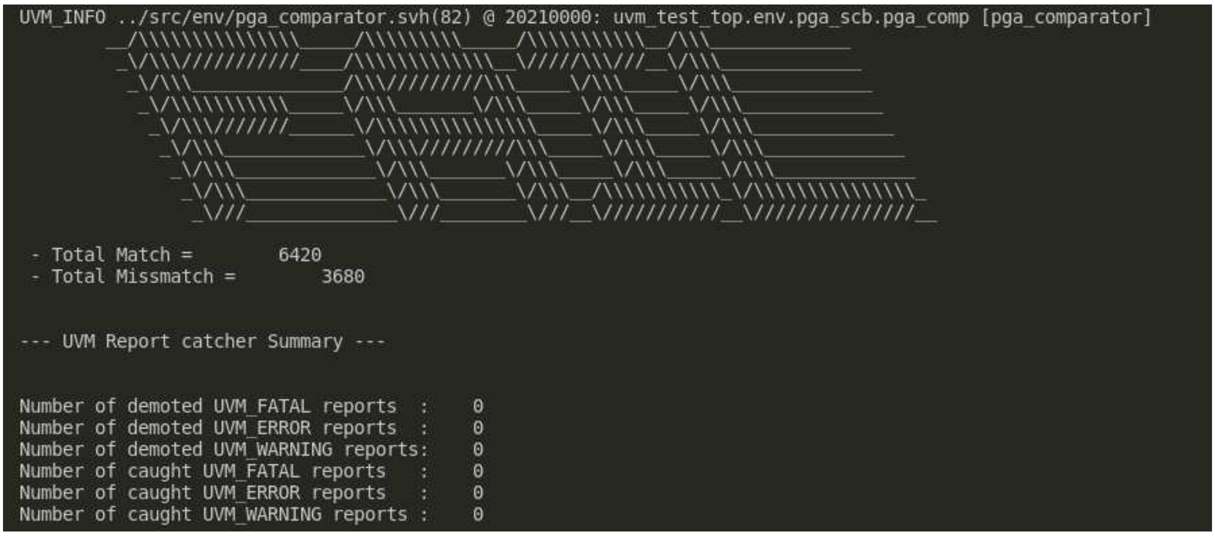
Fonte: Autoria própria

sinal para os maiores valores de ganho da simulação na figura 12.

3.4.5 DESAFIOS ENCONTRADOS

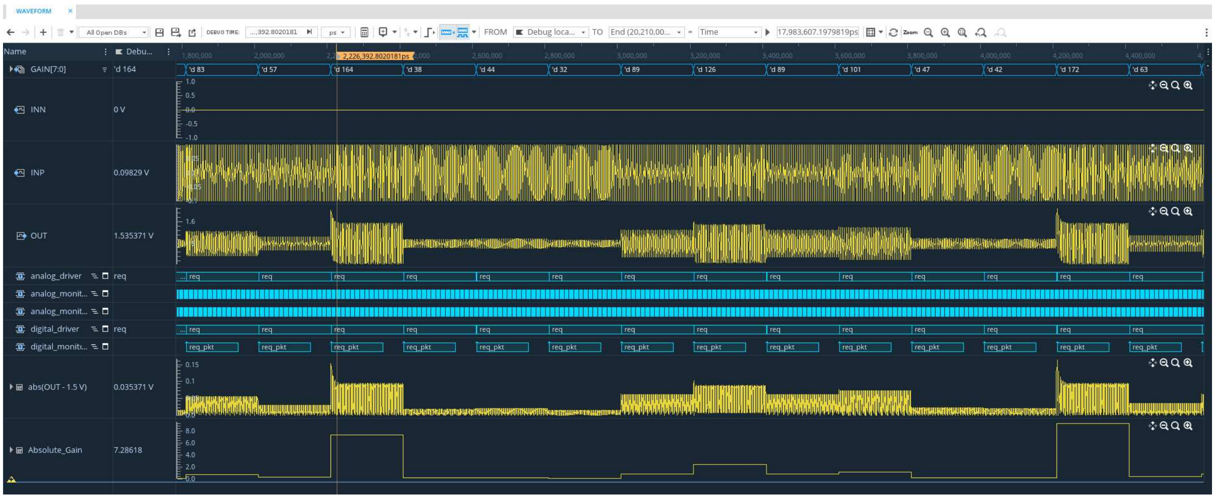
Um dos principais obstáculos para essa parte final do estágio foi a construção do arquivo de configuração analógica *.scs*. O estagiário precisou revisitar os RAKs feitos na primeira parte do estágio, além de investigar em outros *Training bytes* da *Cadence* mais exemplos de modelagem de circuitos em *.vams* para entender como deveria ser estruturado esse arquivo. Nesse ponto, um possível ponto de melhoria no ambiente de verificação é possível de ser observado. Dentro do arquivo de configuração, é necessário informar um tempo de simulação para o circuito analógico. Esse tempo é diferente do tempo

Figura 9 – *Log* do *Xrun* - Simulação do 2^o PGA - 100MHz - Falha



Fonte: Autoria própria

Figura 10 – Formas de ondas - 2^o PGA - 100MHz - Falha



Fonte: Autoria própria

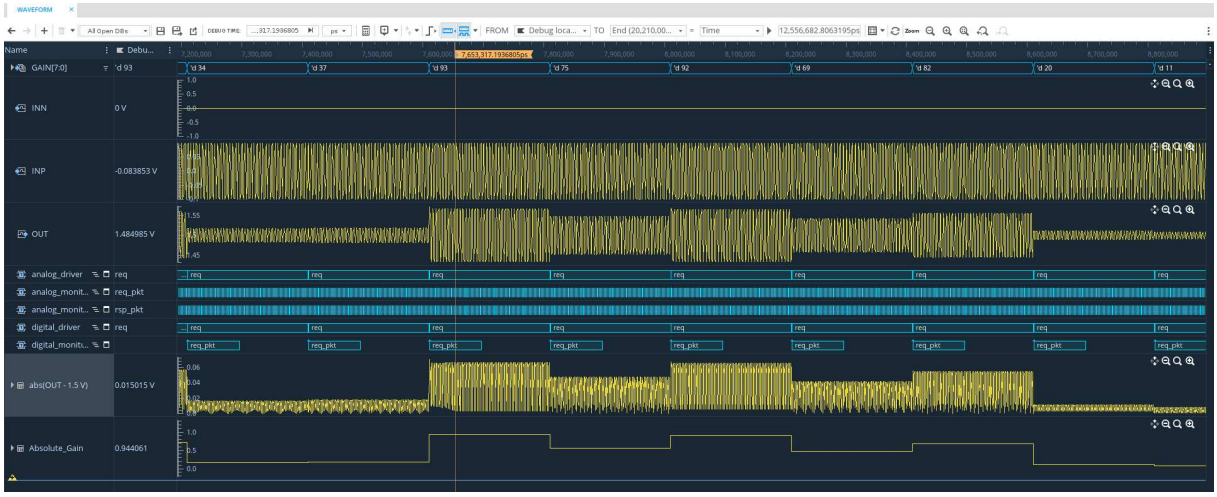
de simulação configurado no ambiente UVM através do número de transações a serem lançadas no teste. Caso o tempo do UVM seja menor que o tempo definido no arquivo de controle, a simulação ocorre normalmente até o fim. Caso contrário, o teste encerra antes da finalização de todas as fases do UVM. Sendo assim, seria interessante a implementação de uma lógica que sincronizasse esses dois tempos, a fim de evitar problemas de simulação durante as regressões, por exemplo.

Além disso, a comparação das amplitudes do modelo de referência com as amplitudes do DUT também foi um problema. O estagiário inicialmente utilizou do mesmo raciocínio empregado no primeiro PGA, comparando tomando como base as casas decimais para atribuir uma tolerância. Entretanto, nenhum teste teve sucesso. Depois, ao implementar a tolerância com base no erro percentual entre os modelos, os testes

Figura 11 – *Log* do *Xrun* - Simulação do 2º PGA - $100MHz$ - Sucesso

Fonte: Autoria própria

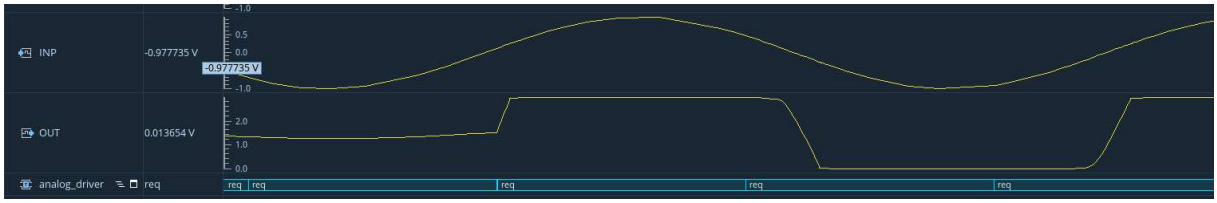
Figura 12 – Formas de ondas - 2º PGA - $100MHz$ - Sucesso



Fonte: Autoria própria

começaram a passar para os casos em que o *design* não satura a saída. Nos casos de saturação, a implementação do PGA suaviza os efeitos de modo que não é apenas um gatilho $OUT = min(V_{Sat}, VOUT)$. Além disso, não existe nenhuma documentação que forneça em detalhes como essa saturação ocorre, e isso continuou causando *mismatches* na simulação. Um exemplo de como a saída se comporta nesses casos pode ser visto na figura 13.

Figura 13 – Saturação da saída - 2º PGA



Fonte: Autoria própria

Desse modo, o modelo de referência para o segundo PGA não ficou robusto o suficiente para comparar com eficiência as amplitudes. Entretanto, isso não interfere no desenvolvimento do estágio, uma vez que o objetivo principal de toda a implementação era apenas integrar diferentes circuitos a um ambiente de verificação UVM.

4 CONCLUSÕES

Neste relatório, foram detalhadas as atividades realizadas pelo aluno João Pedro Melquiades Gomes durante o seu período de estágio no Laboratório de Sistemas Embarcados e Computação Pervasiva. Ao longo dessa experiência, o estagiário teve a oportunidade de ser estimulado, por meio de aprendizado e desafios, a aprimorar habilidades técnicas e desenvolver maturidade na resolução de problemas.

Além disso, foi possível definir alguns métodos de integração de circuitos mistos com ambientes de verificação UVM, com exemplos funcionais do ambiente funcionando e o DUT sendo estimulado/monitorado. Com isso, é possível concluir que é plausível o uso da metodologia de verificação que já é universal para circuitos mistos assim como para circuitos digitais, e uma vez que *designs* AMS são mais comuns na realidade, unir esses dois domínios em uma única metodologia de verificação como UVM faz com que o processo se torne cada vez mais conciso e eficiente, visto que um único ambiente tem a capacidade de estimular o DUT de forma completa.

Por fim, é importante ressaltar que as escolhas tomadas pelo estagiário durante o desenvolvimento desse trabalho não são necessariamentes as melhores decisões. Existe um grupo de trabalho da *Accelera* responsável por realizar pesquisas e desenvolver um padrão universal para a verificação de circuitos mistos com UVM, e possivelmente, quando esse padrão sair, algumas abordagens serão melhores escolhidas em relação ao que foi exposto no último *Webinar* em 2022. Porém, a ideia de construção de um módulo de conversão dos domínios analógicos e digital se mantém em todas as técnicas utilizadas pelas empresas atualmente, o que torna possível estimar que a solução final adotada pelo padrão não seja algo muito diferente do que já foi mostrado.

REFERÊNCIAS

BRENNAN, J. et al. The how to's of advanced mixed-signal verification. In: *Design and Verification Conference and Exhibition*. [S.l.: s.n.], 2015. 19

GOMES, J. P. M. *PGA Verification*. Disponível em: <https://github.com/JoaoPi314/pga_verification>. 35

JAEHA, K.; DANčAK, C. *Harnessing the Power of UVM for AMS Verification with XMODEL*. 2023. <https://www.scianalog.com/webinars/w20230302/>. 19

SPINELLIS, D. Git. *IEEE software*, IEEE, v. 29, n. 3, p. 100–101, 2012. 14

UVM-AMS Working Group. *Workshop: UVM-AMS: A UVM-Based Analog Verification Standard*. 2021. <https://www.accellera.org/resources/videos/uvm-ams-workshop-2021>. 18

UVM-AMS Working Group. *UVM-AMS: An Update on the Accellera UVM-AMS Standard*. 2022. <https://vimeo.com/720255794>. 19, 20, 37