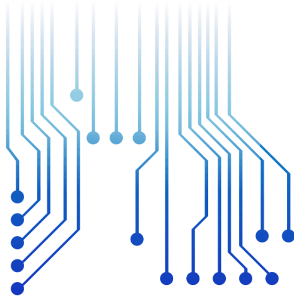
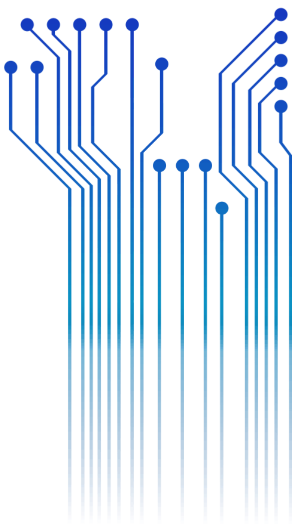


Curso de Graduação em Engenharia Elétrica

Gustavo Vilar de Farias

Trabalho de Conclusão de Curso

Desenvolvimento de um gerador de banco de registradores em hardware



Campina Grande

2024

GUSTAVO VILAR DE FARIAS

DESENVOLVIMENTO DE UM GERADOR DE BANCO DE REGISTRADORES EM HARDWARE

Trabalho de Conclusão de Curso submetido à Coordenação do Curso de 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.

Área de Concentração: Microeletrônica

Orientador:
Professor Gutemberg Gonçalves dos Santos Júnior, Dr.

Campina Grande
2024

GUSTAVO VILAR DE FARIAS

DESENVOLVIMENTO DE UM GERADOR DE BANCO DE REGISTRADORES EM HARDWARE

Trabalho de Conclusão de Curso submetido à Coordenação do Curso de 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.

Área de Concentração: Microeletrônica

Aprovado em 24 / 10 / 2024

Professor Marcos Ricardo Alcântara Morais
Universidade Federal de Campina Grande
Avaliador, UFCG

Professor Gutemberg Gonçalves dos Santos Júnior, Dr.
Universidade Federal de Campina Grande
Orientador, UFCG

AGRADECIMENTOS

A conclusão deste trabalho só foi possível graças ao apoio, à colaboração, influência e inspiração de muitas pessoas, a quem sou profundamente grato.

Em primeiro lugar, agradeço ao meu orientador, Gutemberg, pela orientação não só nesse trabalho, mas em toda a minha graduação. Sua dedicação e conhecimento foram essenciais para o desenvolvimento deste projeto. Agradeço também a todos os professores que foram importantes em minha jornada, contribuindo com conhecimento, incentivo e inspiração ao longo do caminho.

Aos meus colegas do laboratório x-men, vocês foram fundamentais para que este trabalho se concretizasse e sou muito grato pela parceria, momentos de descontrações e trocas.

Também agradeço aos meus amigos, que tornaram essa jornada acadêmica mais leve e divertida. A troca de experiências, as conversas e os momentos de descontração ajudaram a superar os desafios ao longo do caminho.

Aos amigos que fiz durante o intercâmbio, sou muito grato pelas vivências e aprendizados compartilhados. Vocês me proporcionaram uma experiência enriquecedora que levarei comigo para sempre.

Aos livros e jogos que joguei com amigos que foram uma válvula de escape durante todo este processo.

A minha família que sempre estiveram ao meu lado, especialmente aos meus pais e minha irmã, Milena, Angélica e Marcondes, agradeço por todo o apoio e incentivo ao longo da minha jornada. Vocês sempre acreditaram no meu potencial me apoiando e proporcionando a base necessária para seguir em frente e sempre buscar o melhor de mim.

Por fim, gostaria de agradecer a minha companheira, e aos amigos mais próximos que estiveram ao meu lado nos momentos mais desafiadores. O apoio emocional e a compreensão de vocês foram essenciais ao longo desses anos.

A todos vocês, meu sincero agradecimento.

"Nada termina, Adrian. Nada jamais termina"

Alan Moore

RESUMO

Este trabalho apresenta o desenvolvimento de uma ferramenta automatizada para a geração de bancos de registradores em hardware (SystemVerilog) utilizando Python. A ferramenta tem como objetivo reduzir o tempo de desenvolvimento de ASICS e minimizar erros manuais. O projeto implementa o AMBA APB como protocolo de comunicação, garantindo flexibilidade e adaptabilidade a diferentes arquiteturas. Os resultados mostram que a solução proposta é eficaz para melhorar a produtividade e a qualidade dos projetos de hardware, além de oferecer potencial para expansões futuras.

Palavras-chave: Bancos de Registradores, Geração Automatizada, Python, SystemVerilog

ABSTRACT

This work presents the development of an automated tool for generating register banks in hardware (SystemVerilog) using Python. The tool aims to reduce the development time of ASICs and minimize manual errors. The project implements AMBA APB as the communication protocol, ensuring flexibility and adaptability to different architectures. The results show that the proposed solution is effective in improving the productivity and quality of hardware designs, while also offering potential for future expansions.

Keywords: Register Banks, Automated Generation, Python, SystemVerilog

LISTA DE ILUSTRAÇÕES

Figura 1 - Exemplo de como um registrador é definido (PULP PLATFORM, 2016) 14
Figura 2 - Bloco SPI com interface APB (ALI et al., 2021).. 16
Figura 3 - Diagrama de Classes do Gerador.. 19
Figura 4 - Escrita e leitura com sinal externo sendo gerado.. 27

LISTA DE TABELAS

Tabela 1 - Sinais do protocolo AMBA APB 15
Tabela 2 - Exemplo de tabela CSV de configuração 17

Lista de Abreviaturas e Siglas

HDL	Linguagem de descrição de hardware

SUMÁRIO

Agradecimentos iv
Resumo vi
Abstract vii
Lista de Ilustrações viii
Lista de Tabelas ix
Lista de Abreviaturas e Siglas x
Sumário xi
1 Introdução 12
2 Embasamento Teórico 14
3 Metodologia 17
3.1 Estrutura do arquivo CSV 17
3.2 Arquitetura do Software 18
3.2.1 Input Parser 20
3.2.2 Bit e *Register* 20
3.2.3 IO Protocol 20
3.2.4 *Flip Flop* 21
3.2.5 *RB Generator* 21
3.3 Diretório 22
4 Resultados 23
4.1 Entradas, saídas e Parâmetros 23
4.2 Registradores 24
4.3 Lógica de Leitura 24
4.4 Lógica de Escrita 25
4.5 Definições 26
4.6 Testes 27
5 Conclusão 28
Referências 29
APÊNDICE A – register_bank.sv 30
APÊNDICE B – register_bank.svh 32

1 INTRODUÇÃO

Com o avanço da microeletrônica e o crescente uso de sistemas embarcados em aplicações de alta performance, a necessidade de projetar circuitos digitais eficientes, flexíveis e de fácil manutenção tornou-se uma prioridade na indústria de hardware. Entre os componentes fundamentais desses sistemas estão os bancos de registradores, utilizados para armazenar e gerenciar dados essenciais para o funcionamento de diversos tipos de dispositivos. Contudo, o desenvolvimento manual desses bancos de registradores em linguagens de descrição de hardware, como VHDL, Verilog e SystemVerilog, pode ser um processo repetitivo, propenso a erros e de difícil escalabilidade.

Neste contexto e devido a crescente complexidade dos projetos de hardware, o uso de ferramentas automatizadas para a geração de código surge como uma solução promissora para otimizar o processo de desenvolvimento e aumentar a confiabilidade dos sistemas, permitindo que os projetistas foquem em aspectos de maior valor agregado no design.

Tendo isso em vista, o principal objetivo deste trabalho é propor uma ferramenta que agilize a criação de bancos de registradores, proporcionando uma redução significativa no tempo de desenvolvimento e uma minimização dos erros manuais comuns no processo de codificação em linguagens de descrição de hardware. Além disso, a solução visa ser facilmente extensível e adaptável a diferentes arquiteturas, especificações de projetos e protocolos de comunicação.

Essa ferramenta, desenvolvida em Python, gera código de descrição de hardware (SystemVerilog) e é configurável partir de um arquivo CSV contendo informações e a especificação dos registros que devem ser gerados. A escolha de Python como linguagem de programação se deve à sua simplicidade e versatilidade, permitindo a criação de scripts de automação de forma fácil, já SystemVerilog foi escolhido como linguagem de descrição de hardware pela sua estrutura e ferramentas que facilitam a construção de código parametrizáveis.

No desenvolvimento do trabalho, Capítulo 2, é feita uma revisão bibliográfica procurando definir o que são bancos de registradores, explicitar protocolos de

comunicação, padrões adotados no projeto e trabalhos na área de automação de no design de hardware. Posteriormente, no Capítulo 3, é introduzida a metodologia utilizada, quando são apresentados a estrutura do projeto e os métodos utilizados para a geração do HDL. Em seguida, no Capítulo 4, são mostrados os resultados em termos de código e testes obtidos. O trabalho é finalizado no Capítulo 5, com o encaminhamento das conclusões e proposta de refinamentos para pesquisas similares posteriores.

2 EMBASAMENTO TEÓRICO

Os registradores são pequenas unidades de armazenamento temporárias de dados utilizados em chips por terem uma velocidade de acesso rápida em comparação com as memórias. De forma a facilitar e organizar seu acesso, eles podem ser agrupados em um bloco chamado de banco de registradores que fornecem uma interface rápida e eficiente de escrita e armazenamento de dados, sendo amplamente utilizados na configuração e acesso de sinais internos de chips e periféricos.

Nos periféricos, esses registradores são frequentemente empregados para verificar status e realizar configurações do bloco (PULP PLATFORM, 2016). Portanto, é crucial que essas informações estejam centralizadas e facilmente acessíveis em um local que possa ser acessado tanto internamente quanto externamente.

5.4.6 STATUS (Status Register)

Address: `0x1A10_5014`
Reset Value: `0x0000_0000`

`Bit 7` **RXA**: Acknowledge from sent data.

`Bit 6` **BUS**: Bus is busy.

`Bit 5` **AL**: Arbitration lost.

`Bit 4:2` **Reserved**: Set to 0.

`Bit 1` **TIP**: Transfer in progress.

`Bit 0` **IRQ**: Interrupt received.
This flag is always set when transmission has finished or bus arpitration was lostm, regardless of whether interrupts are enabled or not. This flag can possibly polled and is cleared by writing 1 to the IA command register.

FIGURA 1 - EXEMPLO DE COMO UM REGISTRADOR É DEFINIDO (PULP PLATFORM, 2016)

Na **Figura 1** vemos um exemplo de como esses registradores são definidos em datasheets, onde temos vários bits cada um com sua função e tamanho específico.

Geralmente, a comunicação nos bancos de registradores é realizada por protocolos de comunicação estruturados e conhecidos como APB (Advanced Peripheral

Bus), AXI (Advanced eXtensible Interface) e AHB (Advanced High-performance Bus), que garantem o desempenho e integridade do sistema. Para esse trabalho foi utilizado o APB como interface padrão, mas a adição de novos protocolos é suportada.

Esse protocolo AMBA APB foi desenvolvido pela Arm com o objetivo de ser uma interface simples, performática e de baixo custo para aplicações em periféricos, especificamente para programação e leitura de registros (ARM, 1999).

Os principais sinais utilizados nesse protocolo são descritos na tabela abaixo.

TABELA 1 - SINAIS DO PROTOCOLO AMBA APB

Nome	Fonte	Descrição
PCLK	Clock	Sinal de clock
PRESETn	Sistema	Sinal de reset
PADDR	Mestre	Endereço de onde será realizada a operação
PSELx	Mestre	Seleção de escravo, para sinalizar que o escravo foi selecionado e uma transferência de dado foi demandada
PENABLE	Mestre	Ativação, indica que uma transferência APB está acontecendo
PWRITE	Mestre	Direção da transação, sendo nível logico ativo para indicar uma escrita e nível lógico desativo para indicar uma leitura
PWDATA	Mestre	Dado a ser escrito no escravo
PREADY	Escravo	Indica que o escravo está pronto para receber uma transação
PSLVERR	Escravo	Indica que houve um erro na transferência de dados
PRDATA	Escravo	Dado do escravo lido

Esse protocolo é amplamente utilizado em diversos periféricos para realizar a leitura e escrita em seus registradores, com esse objetivo o bloco desenvolvido neste trabalho tem a função de atuar fornecendo os registros e a interface de comunicação externa e interna ao bloco. Na **Figura 2** podemos ver um bloco onde a interface APB é utilizada para se comunicar com os registros de um periférico e onde o nosso banco de registradores poderia ser utilizado na construção desse periférico.

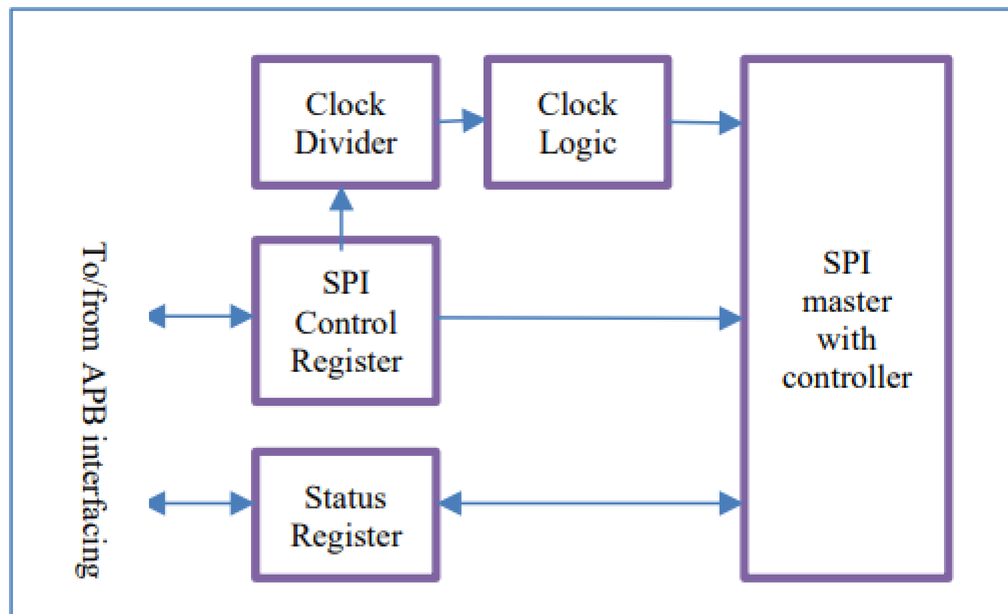


FIGURA 2 - BLOCO SPI COM INTERFACE APB (ALI ET AL., 2021).

Para garantir que o bloco gerado possa ser utilizado em vários tipos de periféricos e tenha uma alta configurabilidade é utilizado um arquivo de configuração em formato CSV (Comma-Separated Values) que consiste em valores separados por virgulas e é geralmente utilizado para representar tabelas, pois podem ser exportados diretamente de planilhas como Microsoft Excel, Google Sheets, etc. (FREELON, 2010) e são facilmente utilizados nos programas Python.

Com isso, vemos que o desenvolvimento de hardware utilizando Python é uma solução prática para aumentar a parametrização e confiabilidade do design (JIANG et al., 2020).

3 METODOLOGIA

Nesse capítulo iremos abordar a organização do trabalho desenvolvido e os métodos utilizados. Começando pela estrutura do arquivo de entrada no formato CSV, que é utilizado para configurar o gerador desenvolvido em Python que tem sua arquitetura também descrita nesse capítulo e por fim é explicitado a organização das pastas nesse projeto.

3.1 ESTRUTURA DO ARQUIVO CSV

O arquivo CSV representa uma tabela contendo informações sobre os registradores que devem ser gerados e é utilizado como entrada do gerador. A **Tabela 2** é um exemplo de como deve ser esse arquivo.

TABELA 2 - EXEMPLO DE TABELA CSV DE CONFIGURAÇÃO

Register Address	Register Name	Bit Name	Position Bit	Access	From Controller	To Controller	Functional Description
0	addr	TBA	31	R W	0	1	Type of address
		reserved	30:7	R	0	0	
		SLVADDR	6:0	R W	0	1	Slave address
4	status	REC	2	R	1	0	Byte received
		TRA	1	R WC	0	0	Byte transmited
		NAK	0	R WC	0	0	Nack response

As colunas apresentadas nessa tabela representam:

- **Register Address:** Endereço do registrador e deve ser o mesmo para o mesmo registrador
- **Registar Name:** Nome do registrador e deve ser o mesmo para o mesmo registrador
- **Bit Name:** Nome do bit ou conjunto de bits a ser especificado

- **Position Bit:** Posição que o bit se encontra no registrador, podendo ser uma posição qualquer entre 0 e o tamanho do registrador ou um intervalo separado por ":" e começando pela posição do bit mais significativo e terminando pela posição do bit menos significativo
- **Access:** Indica o tipo de acesso para aquele determinado bit ou conjunto de bits podendo ser W para escrita, R para leitura e WC para write-clear que são bits onde devemos escrever nível lógico 1 para poder resetar o seu valor para 0
- **From Controller:** 1 para indicar que o valor desse bit virá também do controlador externo ao bloco
- **To Controller:** 1 para indicar que o valor desse bit deverá ser colocar como uma saída do bloco
- **Functional Description:** Uma descrição funcional do que o bit representa, seu uso é opcional e é inserido como um comentário no código gerado.

Os registradores e bits da **Tabela 2** foram colocados juntos e em sequência apenas para facilitar a visualização, mas isso não é um requisito de entrada.

3.2 ARQUITETURA DO SOFTWARE

O software do gerador tem sua estrutura de classes representado na **Figura 3**, temos uma classe principal "RB generator" que se relaciona com a classe "Input Parser" para tratamento do arquivo de entrada, com a classe "IO Protocol" para gerir o protocolo de comunicação utilizado e com as classes de "Register" e "FlipFlop" que representam os registradores e flipFlops utilizados para geração do HDL. O detalhamento de cada classe é apresentado nas seções abaixo.

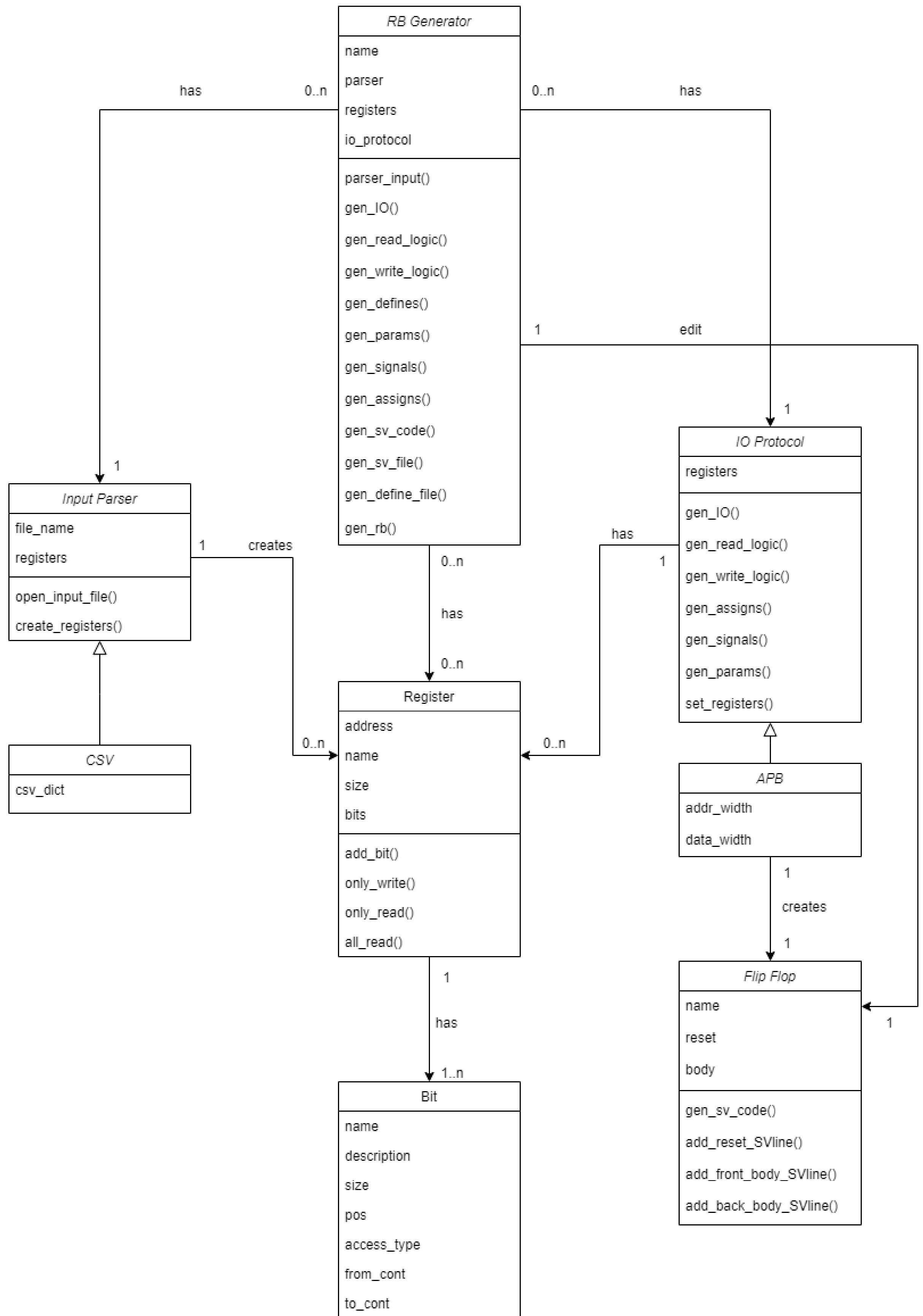


FIGURA 3 - DIAGRAMA DE CLASSES DO GERADOR

3.2.1 *Input Parser*

Essa classe é uma classe abstrata responsável por gerir a forma como o arquivo de entrada é lido, contendo apenas o nome do arquivo que deve ser aberto e funções para abrir esse arquivo e gerar os registradores.

Essa classe é feita de forma que possamos adicionar qualquer método de entrada para o gerador, pois o interesse dessa classe é a geração dos registradores representados pela classe *Register.*

Para adicionarmos nosso método de entrada precisamos implementar uma classe filha da classe *Input Parser,* é o que é feito pela classe *CSV* que implementa as funções abstratas de abertura do arquivo e de geração dos registradores a partir de um arquivo CSV de entrada.

3.2.2 *Bit* e *Register*

As classes *Bit* e *Register* contém a representação do bit e do registrador como mostrado na **Tabela 2**, trazendo todas os atributos apresentados na tabela. Além disso, cada bit é associado a um registrador através de um vetor de *Bit* na classe *Register* ordenado pela posição dos bits.

A classe *Register* também possui funções utilitárias que são utilizadas na geração do HDL, como: *only_write* utilizada para sabermos se todos os bits do registrador são apenas para escrita; o*nly_read* retorna verdadeiro se todos os bits do registrados são apenas para leitura; e *all_read* que retorna verdadeiro se todos os bits do registrador podem ser lidos.

3.2.3 *IO Protocol*

Tem a função de gerar, a partir de um protocolo de comunicação, a interface de escrita e a leitura dos registradores. Para que exista a possibilidade de expansão para diferentes protocolos de comunicação, essa classe é abstrata e deve ser herdada pela classe que realmente implementa o protocolo como a classe *APB*.

As funções dessa classe são as seguintes:

- *gen_io:* para gerar o código das entradas e saídas do protocolo;

- *gen_read_logic:* gera o código responsável pela lógica de leitura dos registradores;
- *gen_write_logic:* gera o código responsável pela lógica de escrita dos registradores;
- *gen_assigns:* gera os *assigns* necessários para o protocolo de comunicação;
- *gen_signals:* gera os sinais necessários para o protocolo de comunicação;
- *gen_params:* gera os parâmetros do HDL que dizem respeito ao protocolo;
- *set_registers:* recebe os registradores que são gerados do *input parser.*

A classe *APB* implementa essas funções e também adiciona os atributos: add_width e data_width, utilizados para configurar a quantidade de bits utilizado para o endereço e para o dado respectivamente.

3.2.4 *FLIPFLOP*

Essa classe é utilizada para definir um bloco sequencial no HDL. Como não podemos ter 2 blocos sequenciais diferentes que alterem a mesma variável, esse bloco deve ser compartilhado pelas classes do *IO Protocol,* que escreve nos registros de acordo com o protocolo de comunicação, e pelo *RB Generator,* que escreve nos registros através de variáveis externas ao bloco.

Assim, a classe *FlipFlop* é utilizada para gerar o código HDL de uma lógica sequencial.

3.2.5 *RB GENERATOR*

É a classe principal do gerador responsável por gerar o código em HDL e os respectivos *defines*, para isso essa classe possui um *Input Parser* para ler o arquivo de entrada e transformar em objetos do tipo *Register* ou *Bit*. Possui também um *IO Protocol* para gerar o código responsável por fazer a comunicação externa utilizando um protocolo.

O papel dessa classe é integrar o código gerado pelo *IO Protocol* ao mesmo tempo que gera o código responsável pelos sinais para comunicação com o controlador. Para isso utiliza as seguintes funções:

- *parser_input:* utilizado para gerar os registradores a partir do arquivo de entrada. Utiliza a classe *Input Parser* para isso;
- *gen_IO:* gera os sinais de entrada e saída do módulo, adiciona os sinais do controlador aos sinais do protocolo;
- *gen_read_logic:* gera a lógica de leitura do bloco com base na lógica de leitura do protocolo;
- *gen_write_logic:* gera a lógica de escrita de bloco, integrando a escrita dos sinais que vem do controlador com as escritas vindas do protocolo de comunicação. Para essa integração, usa-se um objeto *FlipFlop* pois a escrita dos dados é sequencial;
- *gen_defines:* gera os *defines* do módulo. Em geral são os endereços dos registradores;
- *gen_params:* gera os parâmetros do bloco;
- *gen_signals:* gera os sinais do bloco. Neste ponto é realizada a definição dos registradores;
- *gen_assigns:* gera os *assigns* necessários para a comunicação do bloco;
- *gen_sv_code:* gera o código em SytemVerilog de cada parte e os integra em um único módulo;
- *gen_define_file:* gera o arquivo de definições, onde teremos todos os *defines* necessários;
- *gen_sv_file:* gera o arquivo em SystemVerilog do bloco;
- *gen_rb:* função principal que irá gerar todos os arquivos do banco de registros.

3.3 DIRETÓRIO

O diretório do projeto é composto de 3 pastas:

- ***input*:** pasta onde são colocados os arquivos *.csv* de configuração;
- ***lib*:** é onde se encontra os arquivos em python do gerador apresentados na seção anterior;
- ***output*:** pasta onde será gerado os resultados que consiste no arquivo **.sv** com o código do gerador e o arquivo *.svh* que engloba os *defines*.

4 RESULTADOS

Nessa seção iremos abordar o código HDL gerado, apresentando sua arquitetura e código. Para geração dos códigos desse capitulo é utilizado a **Tabela 2** como arquivo de entrada, outras tabelas de entrada com mais configurações e o código do gerador pode ser encontrado no github do projeto.

4.1 ENTRADAS, SAÍDAS E PARÂMETROS

A classe *IO Protocol* define os valores dos parâmetros e gera também as entradas e saídas relacionadas ao protocolo. Adicionalmente, temos os sinais de entrada e saída externos como mostrado no código gerado abaixo.

```
module register_bank #(
// APB params
APB_ADDR_WIDTH = 12,
APB_DATA_WIDTH = 32
)
(
// APB IO
input  logic                      HCLK,
input  logic                      HRESETn,
input  logic [APB_ADDR_WIDTH-1:0] i_PADDR,
input  logic [APB_DATA_WIDTH-1:0] i_PWDATA,
input  logic                      i_PWRITE,
input  logic                      i_PSEL,
input  logic                      i_PENABLE,

output logic                      o_PREADY,
output logic [APB_DATA_WIDTH-1:0] o_PRDATA,

// Controller IO
input  logic                      i_status_rec,

output logic                      o_addr_tba,
output logic                [6:0] o_addr_slvaddr
);
```

Aqui os sinais de entrada possuem o prefixo *i_* e os sinais de saída possuem o prefixo *o_* para que fique claro a direção do sinal em todo o resto do código.

4.2 REGISTRADORES

A estrutura de dados *struct* é utilizada para a definição dos registradores por permitirem um encapsulamento dos bits, além de uma leitura e visualização melhor dos sinais, adotando-se uma ordem de bit mais significativo primeiro a partir de uma ordenação dos bits feito no código do gerador. Abaixo temos um exemplo de como esses registradores são gerados.

```
struct packed {
    logic         TBA;  //Type of address
    logic [23:0] reserved;
    logic [6:0] SLVADDR;  //Slave address
  } r_addr;

  struct packed {
    logic         REC;  //Byte received
    logic         TRA;  //Byte transmited
    logic         NAK;  //Nack response
  } r_status;
```

A *struct* deve ser *packed* para que esse código seja sintetizável.

4.3 LÓGICA DE LEITURA

A lógica de leitura é toda combinacional, tanto para os sinais que vão para o controlador quanto para os sinais do protocolo. O endereço é decodificado em um a*lways_comb* e como o bloco está sempre pronto para receber transações, o valor de *o_PREADY* é sempre nível lógico ativo. Os sinais que irão para o controlador são definidos como *assigns* diretos, como mostrado no código abaixo.

```
always_comb begin
    o_PRDATA = 'h0;

    case (i_PADDR)
```

```
      `ADDRESS_ADDR:
        o_PRDATA[31:0] = r_addr;
      `ADDRESS_STATUS:
        o_PRDATA[2:0] = r_status;
      default:
        o_PRDATA = 'h0;
    endcase
  end

  // To controller
  assign o_addr_tba = r_addr.TBA;
  assign o_addr_slvaddr = r_addr.SLVADDR;

  // To APB
  assign o_PREADY  = 1'b1;
```

O valor de *o_PRDATA* é colocado em '0' no início do a*lways_comb* para que não ocorra a geração de *latch*. É tomado o cuidado para que não seja lido sinais que são apenas escrita e registradores onde nenhum bit pode ser lido onde esses não são nem mesmo colocados dentro do *case*.

4.4 LÓGICA DE ESCRITA

A lógica de escrita é feita inteiramente dentro de um bloco sequencial para que assim seja gerada a estrutura de memória que define os registradores. Abaixo temos um exemplo de como é gerado essa lógica.

```
always_ff @(posedge HCLK, negedge HRESETn) begin
    if(!HRESETn) begin
      r_addr.TBA <= 'h0;
      r_addr.reserved <= 'h0;
      r_addr.SLVADDR <= 'h0;

      r_status.REC <= 'h0;
      r_status.TRA <= 'h0;
      r_status.NAK <= 'h0;
    end
    else begin
      if (i_PSEL && i_PENABLE && i_PWRITE) begin
        case (i_PADDR)
          `ADDRESS_ADDR: begin
            r_addr.TBA <= i_PWDATA[31];
```

```
          r_addr.SLVADDR <= i_PWDATA[6:0];
        end
        `ADDRESS_STATUS: begin
          r_status.TRA <= i_PWDATA[1] ? 1'b0:r_status.TRA;
          r_status.NAK <= i_PWDATA[0] ? 1'b0:r_status.NAK;
        end
      endcase
    end

    r_status.REC <= i_status_rec;
  end
end
```

Nesse bloco temos o *reset* onde os sinais são resetados quando o valor de HRESETn vai para nível lógico baixo. Temos também a escrita dos registradores que vem do protocolo APB, onde é feito uma checagem nos sinais *i_PSEL, i_PENABLE* e *i_PWRITE* que indicam se está acontecendo uma transação de escrita.

Se está acontecendo uma escrita por meio do protocolo APB, o endereço é decodificado e acontece a escrita nos bits que são permitidos a escrita. Para bits que são *write-clear* a escrita de nível lógico 1 faz com que eles sejam resetados para 0, e a escrita de nível lógico 0 faz com que eles mantenham seu valor anterior.

Os sinais que vem do controlador são colocados após a lógica do APB pois esses têm prioridade de escrita, eles são divididos em 2 categorias: sinais que são *write-clear* só são escritos quando teem nível lógico 1, sinais que não são *write-clear* recebem diretamente o valor que veem do controlador externo.

Qualquer lógica mais complexa de escrita de sinais que venha do controlador deve ser implementada após a geração do bloco, como por exemplo um esquema de *ready-valid*.

4.5 DEFINIÇÕES

Os endereços dos registradores são tratados como definições em um arquivo separado que deve ser incluído junto do arquivo principal, seu código exemplo é apresentado abaixo.

```
`define ADDRESS_ADDR   12'd0
`define ADDRESS_STATUS  12'd4
```

4.6 TESTES

Para o teste foi feito um grande arquivo *.csv* que implementa várias combinações de configurações, com o código gerado foi desenvolvido um testbench básico que escreve e lê nos registradores verificando se as limitações propostas no arquivo *.csv* estão sendo seguidas, como por exemplo a impossibilidade de escrever em registros que são somente leitura.

O código gerado passou por todos os testes básico, um exemplo de escrita em um registro e leitura do mesmo é mostrado abaixo, nesse mesmo exemplo podemos observar o sinal externo *o_tba* sendo acionado.

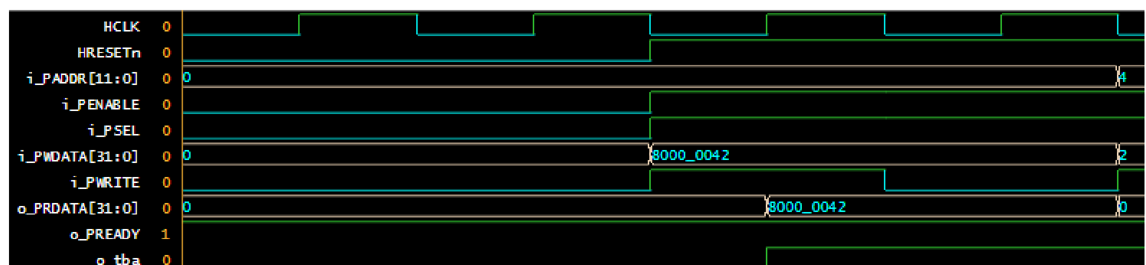


FIGURA 4 - ESCRITA E LEITURA COM SINAL EXTERNO SENDO GERADO

5 CONCLUSÃO

Os resultados obtidos neste trabalho comprovam a eficácia da ferramenta desenvolvida para a geração automatizada de bancos de registradores em hardware, permitindo um desenvolvimento rápido e preciso. Com isso, foi possível reduzir significativamente o esforço manual, aumentando a produtividade e garantindo maior padronização e qualidade nos projetos de hardware.

Como aprimoramentos futuros, propõe-se a adição de uma coluna no arquivo .csv para configurar o valor de reset de cada bit, o que permitirá maior flexibilidade e controle no uso deles. Além disso, planeja-se a implementação de outros protocolos de comunicação e formas alternativas de entrada de dados, como XML e JSON, aumentando ainda mais a aplicabilidade da ferramenta em diferentes cenários e sistemas. Outro ponto importante será a geração automática de testbenchs, permitindo que os testes de validação sejam feitos de forma mais eficiente e prática, contribuindo para a robustez do sistema.

Com isso, teremos a ferramenta em constante evolução, se tornando cada vez mais útil e confiável em diversos tipos de projetos.

REFERÊNCIAS

 ARM. *AMBA Specification (Rev 2.0)*. Cambridge, UK: ARM Ltd., 1999. Disponível em: https://developer.arm.com/documentation/ihi0024/a/. Acesso em: 17 out. 2024.

 FREELON, D. ReCal: intercoder reliability calculation as a web service. *International Journal of Internet Science*, v. 5, n. 1, p. 20-33, 2010. Disponível em: https://dfreelon.org/publications/2010_ReCal_Intercoder_reliability_calculation_as_a_web_service.pdf. Acesso em: 17 out. 2024.

 JIANG, H.; LOCKHART, G.; ZHAO, B.; BATTEN, C. PyMTL3: A Python framework for open-source hardware modeling, generation, simulation, and verification. *IEEE Micro*, v. 40, n. 4, p. 58-66, 2020. Disponível em: https://www.csl.cornell.edu/~cbatten/pdfs/jiang-pymtl3-ieeemicro2020.pdf. Acesso em: 17 out. 2024.

 PULP PLATFORM. *PULPino: Datasheet*. 2016. Disponível em: https://pulp-platform.org/docs/pulpino_datasheet.pdf. Acesso em: 17 out. 2024.

 ALI, M. H.; YUNUS, M. A. M.; RAZALI, S. M.; et al. SPI Controller Using APB Interface for a RISC-V Based Microcontroller. *Journal of Engineering and Emerging Sciences Research (JEESR)*, v. 19, n. 1, p. 1-10, 2021. Disponível em: https://jeesr.uitm.edu.my/v1/ieesr/vol.19/article11.pdf. Acesso em: 17 out. 2024.

APÊNDICE A – REGISTER_BANK.SV

```
module register_bank #(
  // APB params
  APB_ADDR_WIDTH = 12,
  APB_DATA_WIDTH = 32
)
(
  // APB IO
  input  logic                          HCLK,
  input  logic                          HRESETn,
  input  logic [APB_ADDR_WIDTH-1:0] i_PADDR,
  input  logic [APB_DATA_WIDTH-1:0] i_PWDATA,
  input  logic                          i_PWRITE,
  input  logic                          i_PSEL,
  input  logic                          i_PENABLE,

  output logic [APB_DATA_WIDTH-1:0] o_PRDATA,

  // Controller IO
  input  logic                          i_status_rec,

  output logic                          o_addr_tba,
  output logic                    [6:0] o_addr_slvaddr
);

  // Register definitions
  struct packed {
    logic        TBA;  // Type of address
    logic [23:0] reserved;
    logic [6:0] SLVADDR;  // Slave address
  } r_addr;

  struct packed {
    logic        REC;  // Byte received
    logic        TRA;  // Byte transmited
    logic        NAK;  // Nack response
  } r_status;


  // Write FlipFlop
  always_ff @(posedge HCLK, negedge HRESETn) begin
    if(!HRESETn) begin
      r_addr.TBA <= 'h0;
      r_addr.reserved <= 'h0;
      r_addr.SLVADDR <= 'h0;

      r_status.REC <= 'h0;
      r_status.TRA <= 'h0;
      r_status.NAK <= 'h0;
    end
    else begin
      if (i_PSEL && i_PENABLE && i_PWRITE) begin
        case (i_PADDR)
          `ADDRESS_ADDR: begin
            r_addr.TBA <= i_PWDATA[31];
            r_addr.SLVADDR <= i_PWDATA[6:0];
          end
          `ADDRESS_STATUS: begin
            r_status.TRA <= i_PWDATA[1] ? 1'b0:r_status.TRA;
            r_status.NAK <= i_PWDATA[0] ? 1'b0:r_status.NAK;
          end
        endcase
      end
```

```
      r_status.REC <= i_status_rec;
    end
  end

  // Read logic
  always_comb begin
    o_PRDATA = 'h0;

    case (i_PADDR)
      `ADDRESS_ADDR:
        o_PRDATA[31:0] = r_addr;
      `ADDRESS_STATUS:
        o_PRDATA[2:0] = r_status;
      default:
        o_PRDATA = 'h0;
    endcase
  end

  // To controller
  assign o_addr_tba = r_addr.TBA;
  assign o_addr_slvaddr = r_addr.SLVADDR;

endmodule
```

APÊNDICE B – REGISTER_BANK.SVH

```
`define ADDRESS_ADDR  12'h0
`define ADDRESS_STATUS  12'h4
```