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

Trabalho de Conclusão de Curso

Análise por Simulação de Perdas em Conversor Estático PWM Utilizando Módulo FPGA

Samuel de Melo Barros

Campina Grande - PB

Julho de 2019

Samuel de Melo Barros

Análise por Simulação de Perdas em Conversor Estático PWM Utilizando Módulo FPGA

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

Área de Concentração: Eficiência Energética

Universidade Federal de Campina Grande - UFCG

Centro de Engenharia Elétrica e Informática - CEEI

Departamento de Engenharia Elétrica - DEE

Coordenação de Graduação em Engenharia Elétrica - CGEE

Alexandre Cunha Oliveira, D.Sc.

(Orientador)

Campina Grande - PB

Julho de 2019

Samuel de Melo Barros

Análise por Simulação de Perdas em Conversor Estático PWM Utilizando Módulo FPGA

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

Aprovado em _____ /_____ /_____

Professor Avaliador
Universidade Federal de Campina Grande
Avaliador

Alexandre Cunha Oliveira
Universidade Federal de Campina Grande
Orientador

Campina Grande - PB
Julho de 2019

Este trabalho é dedicado aos meus pais, Sonaldo e Sandra, que sempre lutaram para que eu tivesse um futuro digno, e a todos os meus estimados amigos pela paciência e companheirismo ao longo de toda essa jornada.

Agradecimentos

Primeiramente a Deus, sobre todas as coisas, pois a Ele tudo devo e por tudo sou eternamente grato.

Aos meus pais, Sonaldo e Sandra, que sempre fizeram de um tudo para que nada me faltasse. Por todos os ensinamentos e todo apoio para que sempre seguisse meus sonhos. Não são vocês que devem se orgulhar de mim, mas eu de vocês.

A todos da minha família que sempre estiveram ao meu lado. Meus irmãos Savyo e Suiany, em que mesmo sem demonstrar todos os dias, nunca me faltaram com afeto e preocupação, e em ajudas incontáveis me buscando onde fosse preciso.

Aos meus amigos e colegas de curso, com quem dividi noites de estudos (Tinho, Japa, Jorgin, Júlio Mau, Raphinha, Mitor Ramos), risadas improváveis em momentos inoportunos (Marina, Monaliza, Ulisses, Matheus Guerra), que me deram força em meio às preocupações durante o curso, os quais nunca me faltaram. Eles estão nas caronas (Valmir, Dinho, Mylena BBB, Rapha V.), nas conversas descomprometidas durante o almoço (Vagne, Stayner, Giovanny, Giordano), nos rolês aleatórios (Alequis, Samuel Best, Camilinha, Alison, Bruninho, Pedro), nas fofocas diárias, nos jogos, na companhia, nas mensagens, nas ligações. Aos meus amigos de infância de toda uma vida e ao mesmo tempo àqueles que me trouxeram pra mais perto de Deus (Lucas, Janis, Thiaguin), e a amigos recentes que o IEEE me deu a oportunidade de conhecer (Nayara, Phelipe, Rodrigo, Amanda). Cada um teve seu papel, seu ajuste fundamental e foram eles que me ajudaram a acordar todos os dias, às vezes literalmente, e continuar a travar batalhas diárias.

Àqueles que me auxiliaram neste trabalho de uma forma ou de outra, seja na formatação (Myleninha rainha do português) ou bibliografias disponibilizadas, mas em especial à Max, um amigo improvável que nessa reta final do trabalho, não teria conseguido a tempo sem a sua ajuda.

Ao Professor Alexandre Cunha Oliveira, pela orientação deste Trabalho de Conclusão de Curso, por toda paciência, por ter me dado o apoio necessário, sanado minhas dúvidas, disponibilizado o laboratório e os equipamentos para meus experimentos, e todos os ensinamentos que me passou durante toda a graduação, seja dentro ou fora de sala de aula.

Enfim, agradeço a todos que de alguma forma, passaram pela minha vida e contribuíram para a construção de quem sou hoje.

"Tomou, então, Samuel uma pedra,
e a pôs entre Mizpá e Sem,
e chamou-lhe Ebenézer, e disse:
Até aqui nos ajudou o SENHOR."
(Biblia Sagrada,1 Samuel 7,12)

Resumo

O presente trabalho tem como propósito a análise de perdas de potência de chaveamento e condução em um conversor estático PWM por meio de um módulo FPGA. Serão estudados o modo de implementação da simulação por meio da linguagem de descrição de *hardware verilog*, bem como a comunicação serial periférica utilizada (SPI) com o DAC empregado. A perda por condução depende da corrente, em que o MOSFET está conduzindo, e de sua resistência. No entanto, as perdas devido ao chaveamento são difíceis de serem estimadas, devido a não linearidade presente nas capacitâncias parasitas do MOSFET, sendo proposta assim no trabalho uma alternativa para a estimação dessas perdas. A fim de comprovar o funcionamento da implementação realizada, foram analisados o comportamento das potências de perdas de chaveamento e condução, dos módulos de medição que iriam captar esses instantes de chaveamento e condução, e as tensões e as correntes associadas ao conversor por meio de simulações realizadas com auxílio do *software* QUARTUS©.

Palavras-chave: FPGA, comunicação SPI, Conversor DAC, Perdas de potência por condução, Perdas de potência por chaveamento.

Abstract

The present work has the purpose of the analysis of losses of power caused by switching and conduction in a static PWM converter through an FPGA module. The mode of implementation of the simulation will be studied by means of the description language of hardware verilog, as well as the peripheral serial communication used (SPI) with the DAC used. The loss by conduction depends on the current, in which the MOSFET is conducting, and its resistance. However, losses due to switching are difficult to estimate, due to the non-linearity present in the MOSFET parasitic capacitances, thus it is proposed in this work an alternative for the estimation of these losses. In order to prove the performance of the implemented implementation, the behavior of the switching and conduction loss powers, the measuring modules that would capture these switching and conduction moments, and the voltages and currents associated with the converter through simulations were analyzed with the help of software QUARTUS ©.

Keywords: FPGA, SPI communication, DAC converter, Conduction losses, Switching losses.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

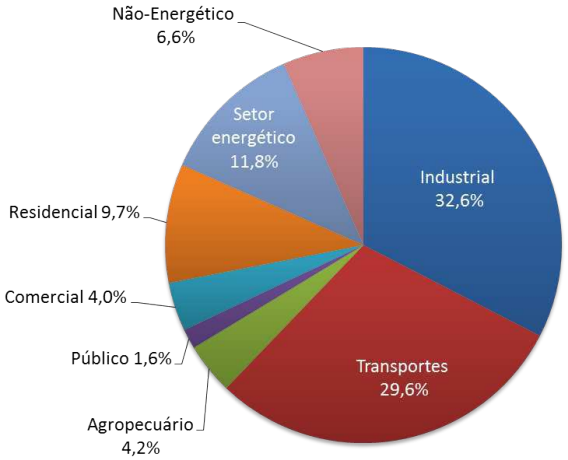
ABNT	Associação Brasileira de Normas Técnicas
ASIC	*Australian Securities and Investments Commission*
DAC	*Digital Analog Converter*
FPGA	*Field-Programmable Gate Array*
GPIO	*General Purpose Input/Output*
IGBT	*Insulated Gate Bipolar Transistor*
MOSFET	*Metal Oxide Semiconductor Field Effect Transistor*
LEIAM	Laboratório de Eletrônica Industrial e Acionamento de Máquinas
PWM	*Pulse Width Modulation*
SPI	*Serial Peripheral Interface*
TSMC	*Taiwan Semiconductor Manufacturing Company*
VHDL	*VHSIC Hardware Description Language*

Sumário

1 Introdução

No cenário nacional, o setor industrial é responsável por quase 1 terço da energia elétrica brasileira consumida, seguido pelos setores de transportes, residencial, energético, não-energético, agropecuário, comercial e repartição pública, respectivamente, como pode ser visto na Figura 1. Esses dados fazem parte do relatório do Ministério de Minas e Energia, e nele destaca-se que os motores são responsáveis pelo consumo energético de cerca de 68%, na indústria, 18%, no setor público, 33% no setor comercial e 34% no setor residencial (EPE, 2007).

Figura 1 – Dados de Consumo Energético Apresentado por Setores.



Fonte: EPE (2007).

No que tange o universo dos consumidores: residencial, comercial, repartição pública e agropecuária, estão os usuários que fazem uso de motores monofásicos através de cargas como, geladeiras, freezers, condicionadores de ar, bombas, etc. Para esse tipo de carga há um potencial de economia energética nos setores público, comercial e para o setor residencial (EPE, 2007).

Nesse contexto da eficiência energética é proposto o estudo e análise da dissipação de potência em conversores, que consiste em analisar a potência dissipada nos semicondutores presentes em diversos conversores estáticos que operam sob modulação PWM.

O conhecimento adequado dessa potência dissipada pelos dispositivos semicondutores de potência do conversor é fundamental para escolha da chave, bem como, do seu dissipador de calor, quando necessário.

A potência dissipada pela chave pode ser dividida em: dissipação de potência por condução e dissipação de potência por chaveamento. A dissipação de potência devido ao

chaveamento, acontece durante a transição de desligado para ligado e vice-versa, enquanto que a dissipação de potência por condução ocorre no momento que a chave estiver atuando.

Neste trabalho, o esforço será concentrado na análise da dissipação de potência por chaveamento, pois seu cálculo não é trivial devido as não linearidades presentes, principalmente nas capacitâncias parasitas, o que leva a dificuldades em dimensioná-las de forma adequada.

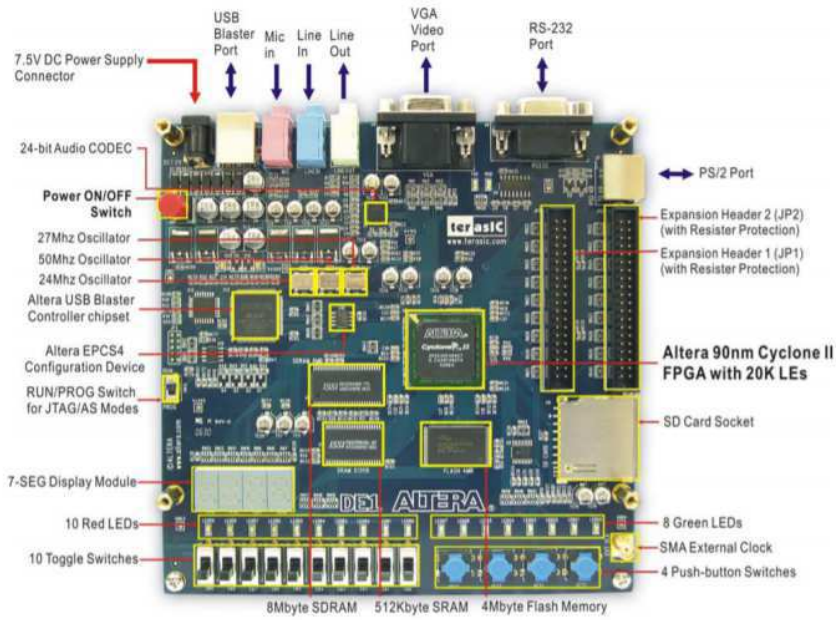
Dessa maneira, o uso de dispositivos com elevada capacidade de processamento digital, capazes de analisar e caracterizar de modo adequado esses valores de correntes, tensões e frequências de chaveamento, tornou atrativo a escolha da utilização de um módulo FPGA para esta investigação.

Seguindo o comercialmente bem sucedido dispositivo Cyclone® da primeira geração, os FPGAs Altera® Cyclone II ampliaram a faixa de densidade do FPGA de baixo custo a 68.416 elementos lógicos (LEs) e fornece até 622 E/S (entrada/saída) pinos utilizáveis e até 1,1 Mbits de memória incorporada.

FPGAs do Cyclone II são fabricado em wafers de 300 mm usando o processo dielétrico *low-k* de 90 nm da TSMC para garantir disponibilidade rápida e baixo custo. Minimizando a área do silício, os dispositivos Cyclone II podem suportar sistemas digitais complexos em um único chip a um custo que rivaliza com o de ASICs. Ao contrário de outros fornecedores FPGA que comprometem o consumo de energia e desempenho para baixo custo, a última geração da Altera de FPGAs de baixo custo - FPGAs Cyclone II, oferecem 60 % mais desempenho e metade do consumo de energia em relação aos FPGAs de 90 nm (ALTERA, 2007).

O conjunto de recursos otimizado e de baixo custo dos FPGAs Cyclone II, os tornam soluções ideais para uma ampla gama de consumidores automotivos, de comunicações, processamento de vídeo, teste e medição, e outras soluções para o mercado final.

Figura 2 – Placa de Desenvolvimento.



Fonte: Altera (2007).

Baseado nisso será adotada uma metodologia de projeto hierárquico que consiste essencialmente em dividir projetos em uma estrutura hierárquica de entidades de projeto. Cada uma das entidades possui uma função específica e uma interface de iteração com outras entidades. Assim, um projeto pode ser visto através de diferentes níveis de hierarquia, desde os blocos lógicos elementares até a visão geral no topo da hierarquia. Quando o fluxo de desenvolvimento de um projeto parte dos blocos elementares os quais são integrados em entidades de nível mais alto, é dito que se trata da metodologia *bottom-up* (TERASIC, 2014).

Diversas vantagens podem ser destacadas quando se emprega a metodologia hierárquica no desenvolvimento de um projeto. Uma delas é que o desenvolvimento do projeto pode ser realizado por partes, ou seja, cada entidade de projeto é compilado, analisado, simulado e testado separadamente. Assim, garante-se que o seu funcionamento esteja de acordo com as especificações do projeto e, eventualmente, erros, modificações e otimizações tornam-se tarefas mais simples de serem executadas. Adicionalmente, ao se analisar o projeto como um todo, simplifica a procura por eventuais erros que venham a ocorrer e, consequentemente, facilita o trabalho para a sua correção já que os módulos foram testados previamente.

1.1 Objetivos

O objetivo do trabalho é desenvolver uma plataforma digital que a partir de sinais de gatilho, corrente e tensão sobre a chave de potência, seja capaz de analisar e discriminar as perdas tanto de condução como de chaveamento que ocorrem em um conversor, bem como discutir os aspectos gerais relacionados à potência dissipada nas chaves. Isto será feito por meio de um hardware implementado numa placa FPGA de modo a caracterizar essas perdas.

Será realizado também a aquisição de sinais de correntes e tensões simuladas que seriam provenientes de um conversor, de modo a captar e interpretar esses sinais a partir do módulo FPGA. Em seguida, será feito o cálculo da potência por meio do processamento desses sinais.

1.2 Organização do Trabalho

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

O **Capítulo 2** apresenta o estudo da hierarquia utilizada na placa FPGA e como será configurada com uma descrição de hardware de módulos que implementem funcionalidades do sistema de análise de perdas em um conversor. Resultados das simulações computacionais no *software* QUARTUS © são apresentados ao final para comprovar a validação da técnica.

O **Capítulo 3** aborda o estudo da interface de comunicação SPI (*Serial Peripheral Interface*) utilizada para realizar a transição de dados entre a placa FPGA e o conversor digital/analógico utilizado.

No **capítulo 4** é feita a análise realizada das perdas por condução e chaveamento na chave de um conversor e a interpretação da conversão realizada pelo conversor DAC utilizado. Também serão mostrados resultados das simulações computacionais no *software* QUARTUS © mostrando o funcionamento da comunicação no sistema.

O **Capítulo 5** apresenta as conclusões do trabalho em um formato do resumo dos resultados alcançados e propostas futuras visando complementar as atividades desenvolvidas.

2 Hierarquia Utilizada

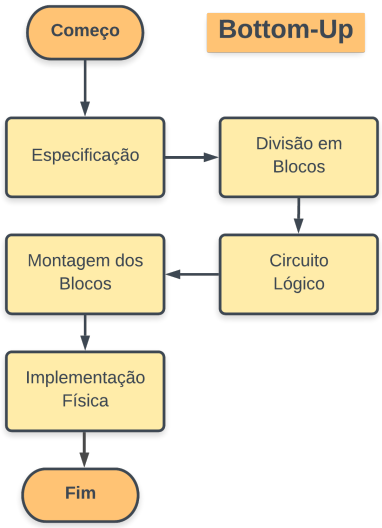
Este capítulo é voltado para o estudo da hierarquia utilizada na placa FPGA e como será configurada com uma descrição de hardware de módulos.

2.1 Metodologia *Bottom-Up*

Como já foi mencionado anteriormente, quando o fluxo de desenvolvimento de um projeto parte dos blocos elementares os quais são, então, integrados em entidades de nível mais alto, é dito que se trata da metodologia *bottom-up* (TERASIC, 2014).

Este capítulo aborda a metodologia utilizada que impõe a seguinte estrutura: diversas entidades auxiliares que devem ser agregadas para constituir uma entidade de projeto principal. Essas entidades são colocadas em uma espécie de árvore, o que forma uma hierarquia. Não respeitar essa hierarquia de entidades resulta em erros durante a compilação do projeto, pois a compilação é executada de forma ascendente, ou seja, a entidade no topo será a última lida no processo de compilação, como pode ser visto na Figura 3.

Figura 3 – Metodologia *Bottom-up*.



Fonte: Autoria Própria.

O problema básico a ser resolvido é a implementação de uma arquitetura eficiente, para execução desse algoritmo ao invés de compilá-lo para sua execução em uma CPU. A tarefa que faz a tradução de um algoritmo para uma arquitetura de hardware eficiente em

um dispositivo de Lógica Programável é denominada Síntese. A síntese cria uma arquitetura com células lógicas que executam as operações de algoritmos, sem a necessidade de se gerar e decodificar instruções (Costa, 2008).

Uma das grandes vantagens da utilização de FPGAs é a possibilidade de se definir vários blocos de hardware, que operam em paralelo, aumentando a capacidade computacional do sistema.

À medida que os projetos ficam mais complexos, as descrições em nível de portas lógicas tornam-se inviáveis, fazendo com que seja necessário descrever os projetos em modos mais abstratos. As linguagens de descrição de hardware também conhecidas como HDL (do inglês, *Hardware Description Language*) foram desenvolvidas para auxiliar os projetistas a documentarem projetos e simularem grandes sistemas.

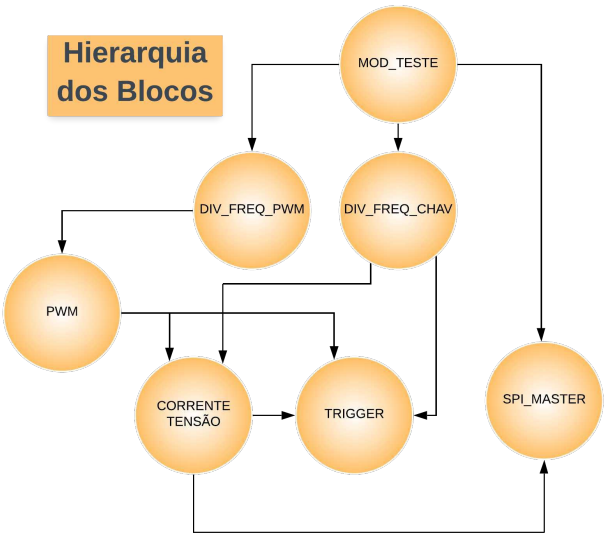
Em uma FPGA, por exemplo, para que cada bloco lógico e suas interconexões sejam configuradas são necessárias algumas centenas de bits. Cada bit de configuração define o estado de uma célula de memória, que controla uma função "look-up table", ou seleciona uma entrada de um multiplexador, ou define o estado de uma chave de interconexão. Existem diversas linguagens de descrição de hardware disponíveis, sendo as mais comumente utilizadas: VHDL (do inglês, *Very High Speed Integrated Circuit Hardware Description Language*) e *Verilog* (Costa, 2008).

As linguagens de descrição de hardware HDL são utilizadas para descrever o comportamento de um sistema digital de variadas formas, inclusive equações lógicas, tabelas verdades e diagramas de estado que utilizam declarações como a linguagem C.

2.2 Hierarquia dos Blocos

Utilizando da metodologia *bottom-up* assim como mencionada anteriormente, a hierarquia que compõe o sistema está disposta na Figura 4. Iniciando do bloco principal *Mod_Teste*, abaixo dele existem outros 6 blocos elementares que cada um possuem um papel essencial ao funcionamento do programa.

Figura 4 – Hierarquia dos Blocos.



Fonte: Autoria Própria.

- **MOD_TESTE**: responsável por inicializar a placa AlteraDE2 com todas as atribuições de pinos, bem como instanciar todos os blocos elementares que foram criados separadamente.
- **DIV_FREQ_PWM**: responsável por realizar o divisor da frequência de 50 MHz que já é intrínseco a placa, para 20 kHz. Esta nova frequência será utilizada no bloco que irá gerar o sinal PWM.
- **DIV_FREQ_CHAV**: responsável por realizar o divisor da frequência de 50 MHz que já é intrínseco a placa, para 1 MHz. Esta nova frequência será utilizada nos blocos de *trigger* e corrente_tensao.
- **SPI_MASTER**: tem por função realizar a comunicação serial da placa FPGA com os periféricos. Essa comunicação será mais detalhada no próximo capítulo.
- **PWM**: objetiva realizar a criação de um sinal PWM, variando seu *duty cycle* a partir da comutação das chaves de SW[7-0].
- **CORRENTE_TENSAO**: esse bloco foi criado para que pudesse ser feita a simulação de valores de corrente e tensão que seriam provenientes de um conversor elementar, do tipo *Buck*, por exemplo. Nesse bloco, os valores de corrente gerados serão incrementados na borda de subida do PWM até alcançar um valor de regime de 20 A por meio de um contador, e começar a decair no instante da borda de descida do PWM. Nesse mesmo instante, o valor da tensão começará a crescer até um valor de regime de 50V, e começar a decair no instante da borda de subida do PWM, repetindo o processo.

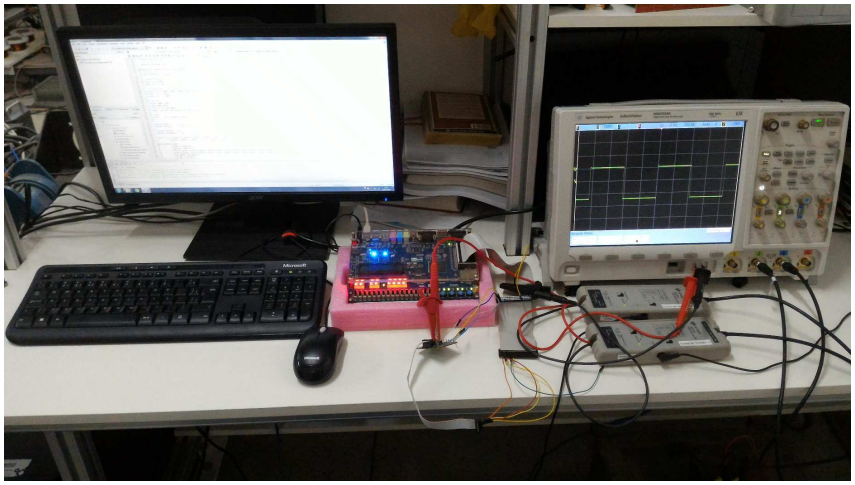
- **TRIGGER**: o bloco de *trigger* tem o papel de identificar os momentos de transição das correntes/tensões em função da comutação das chaves, comandadas pelo sinal PWM, sendo ele capaz de detectar os momentos que os valores de corrente e tensão estiverem em transição. Essa detecção é feita por meio de condicionais que realizam continuamente a verificação dos valores de corrente e tensão gerados à uma taxa de 50 MHz, e a partir desses testes tomar a decisão de permanecer em nível lógico alto (indicando que o chaveamento ainda está ocorrendo) ou decair para nível lógico baixo até que se perceba novamente que os valores de corrente e tensão voltaram a ser chaveados. A variável que representa o chaveamento ativo é a *trig*, e ela possuirá nível lógico alto ativo sempre que os valores de corrente e tensão forem diferentes dos seus valores de regime, ou seja, quando os valores corrente/tensão forem diferentes de 0A/50V ou 20A/2V.

É importante ressaltar, que nesse bloco ainda existe a variável *flag*, sendo ela capaz de caracterizar se o sistema está em estado de condução ou não. Essa variável possuirá nível lógico 1 caso a corrente seja superior à 1A e a tensão menor que 23V. A razão dessas escolhas, se deve ao fato de tentar aproximar os valores reais de condução próximos de corrente de 0.1A e de tensão de 2.3V devido às limitações de implementação de valores de ponto flutuante no *software* utilizado.

2.3 Equipamentos Utilizados

Além da bancada de teste da Figura 5 que foi disponibilizada pelo professor orientador Alexandre para realização do trabalho, uma série de equipamentos do laboratório de eletrônica industrial e acionamentos de máquinas (LEIAM) foram disponibilizados, tais como:

Figura 5 – Bancada de Teste.



Fonte: Autoria Própria.

- Um osciloscópio para que fosse possível a visualização das formas de onda e análise dos dados que estavam sendo gerados, na Figura 6.

Figura 6 – Osciloscópio.

Fonte: Keysight Technologies (2019).

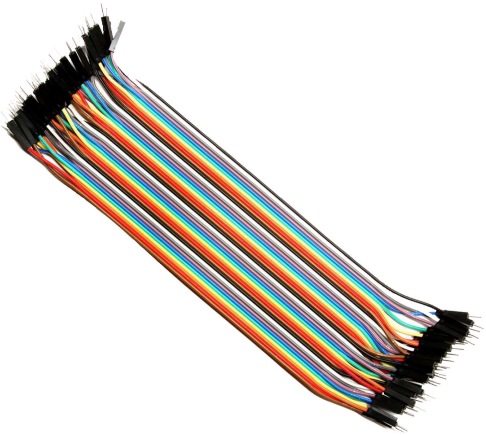
- Pontas de tensão para realizar as aquisições dos sinais de tensão, corrente e potências provenientes dos dispositivos de saída, na Figura 7.

Figura 7 – Ponta de Tensão.

Fonte: eSpot (2019).

- *Jumpers* para as conexões dos elementos da placa FPGA e do conversor D/A que será utilizado, na Figura 8.

Figura 8 – Jumpers.



Fonte: Pinclipart (2019).

- Cabo *flat* responsável pela comunicação com os pinos GPIO da placa FPGA, na Figura 9.

Figura 9 – Cabo *Flat*.

Fonte: USINAINFO (2019).

3 Interface de Comunicação SPI

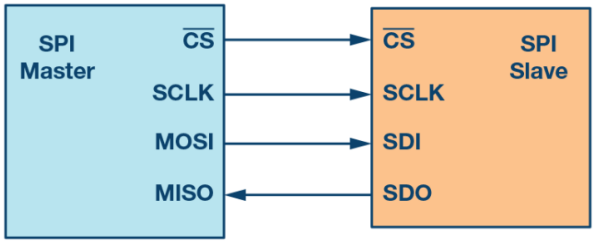
Este capítulo é voltado para a análise da comunicação SPI realizada entre a placa FPGA e o conversor D/A utilizado.

A SPI, do inglês *Serial Peripheral Interface*, é uma das interfaces mais utilizadas entre o microcontrolador e os CIs periféricos, como sensores, ADCs (do inglês, *Analog Digital Converter*), DACs (do inglês, *Digital Analog Converter*), registradores de deslocamento, SRAM e outros.

A SPI é uma forma de comunicação serial, ou seja, é uma transferência de dados binária de forma serial. Pode ser síncrona (com o auxílio de um sinal de alta frequência chamado *clock* que varia entre 0 e 1) ou assíncrono (sem a utilização do sinal de *clock*) (Total Phase, 2019).

Permitindo assim a comunicação do mestre com diversos periféricos, a SPI possui quatro fios, como pode ser visto na Figura 10:

Figura 10 – Configuração da SPI com Mestre (*Master*) e Escravo (*Slave).*



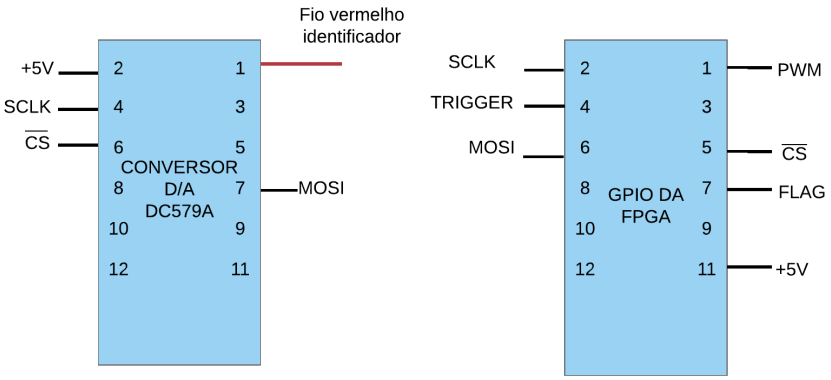
Fonte: Total Phase (2019).

- **MOSI** (*Master Output, Slave Input*): Saída de dados do mestre, também conhecido como SDI (*Serial Data In*) do lado do escravo;
- **MISO** (*Master Input, Slave Output*): Entrada de dados do mestre, também conhecido como SDO (*Serial Data Out*) do lado do escravo;
- **SCLK** (*Serial Clock*): É uma linha de controle direcionada pelo mestre e regula o fluxo de *bits* de dados;
- **CS** (*Chip Select*): Seleção do escravo. São linhas de controle que permitem que os escravos sejam ligados/desligados com o controle do *hardware*.

O mestre no nosso caso é o FPGA, sendo o dispositivo que fornece o sinal de *clock* e determina o estado do CS que será enviado para um dispositivo escravo. O escravo será

aquele que irá receber os dados de *clock* e do CS do seu mestre, que no caso é o conversor D/A, representado no esquema de ligação da Figura 11.

Figura 11 – Esquema de Conexão dos Pinos GPIO e do Conversor DC579A.

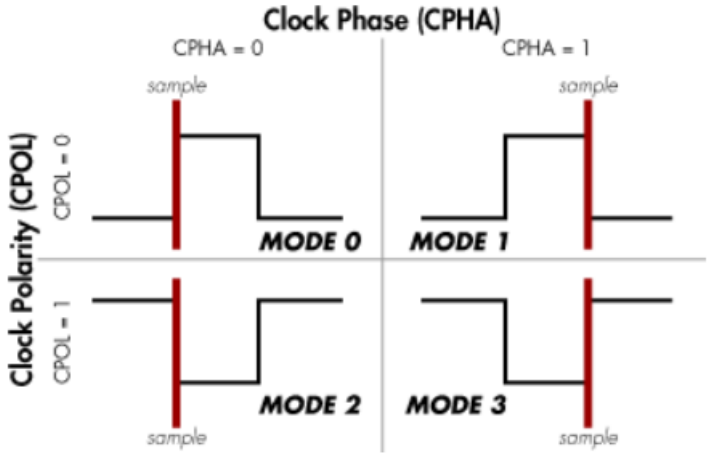


Fonte: Autoria Própria.

Nesta forma de comunicação, todos os escravos compartilham os fios MOSI, MISO e SCK, mas cada um possui seu CS individual junto ao mestre. Essa é uma desvantagem da comunicação SPI, quanto maior o número de escravos, maior será o número de portas digitais utilizadas para representar os sinais CS.

Quando o mestre quer iniciar a transmissão, ele muda o nível lógico do CS do escravo que ele deseja se comunicar e, a depender do modo de operação SPI, ele começa a enviar ou receber os dados bit a bit enquanto gera o clock de sincronia. Os modos SPI são mostrados na Figura 12:

Figura 12 – Modos de Operação SPI.

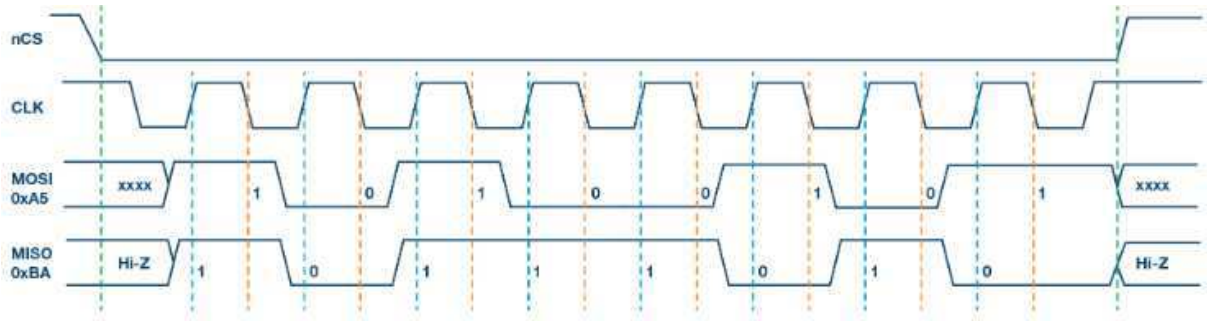


Fonte: Total Phase (2019).

No desenvolvimento deste trabalho foi utilizado o modo SPI 2, representado na Figura 13. Neste modo, a polaridade do *clock* (CPOL) é 1, o que indica que o estado

inativo do sinal de clock é alto. A fase do *clock* (CPHA) neste modo é 0, indicando que a amostra de dado é adquirido na borda de descida (representada pela linha pontilhada laranja) e o dado é alterado na borda de subida (representada pela linha pontilhada azul) do sinal de *clock*. O ínicio e o final da transmissão de dados é representado pela linha pontilhada verde (AnalogDialogue, 2019).

Figura 13 – Modo 2 de Operação SPI.



Fonte: AnalogDialogue (2019).

Dentre as vantagens temos que o SPI é um protocolo de comunicação muito simples. Não possui um protocolo específico de alto nível, o que significa que quase não há sobrecarga. Os dados podem ser alterados em taxas muito altas em *full duplex*, que podem comunicar entre si em ambas direções. Isso torna muito simples e eficiente em um cenário de um único escravo e único mestre.

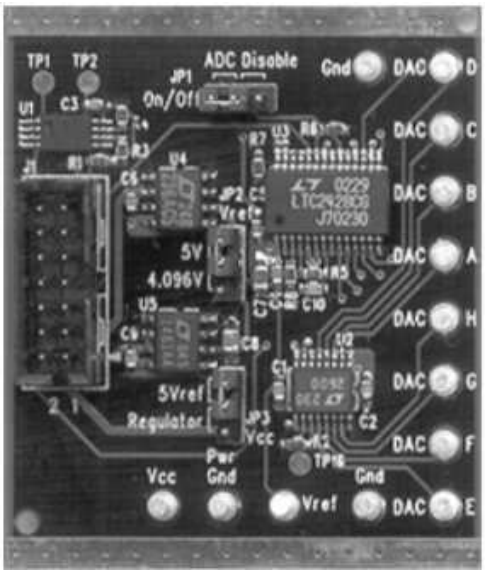
Como desvantagem, sabendo que cada escravo precisa de seu próprio CS, o número de rastreamentos necessários é $n + 3$, onde n é o número de dispositivos SPI. Isso significa maior complexidade da placa quando o número de escravos é aumentado (Total Phase, 2019).

3.1 Conversor LTC2600 e DC579A

Foi utilizado o *demo circuit 579A* que implementa uma placa de demonstração de uso do conversor D/A LTC2600. O LTC2600 mostrado na Figura 14, possui uma gama de registradores de configuração e de armazenamento de dados, todos de tamanho fixo de 16 bits. Uma vez conectado o módulo FPGA, antes do início da escrita do código foi necessário estudar como ler e escrever em seus registradores. O processo encontra-se a seguir:

Inicialmente, ao certificar-se que a chave de *reset* está inativa no nível lógico 1 (baixo ativo), o nível lógico do CS muda de 1 para 0, selecionando o LTC2600 e iniciando uma comunicação SPI descrita no tópico anterior. Quando esta entrada desce para nível

Figura 14 – Conversor LTC2600 e DC579A.

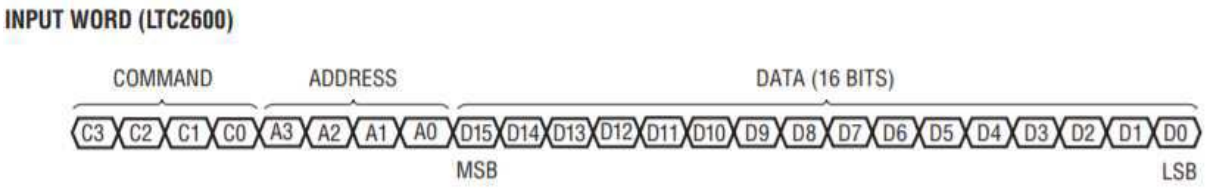


Fonte: Linear Technology (2004).

baixo, ela age como um sinal de seleção de chip, ligando o SDI e o SCK, habilitando o registro de deslocamento de entrada.

Os dados serão transferidos nas 24 bordas subsequentes do SCK como representado na Figura 15. O comando (*command*) a ser realizado é representado pelo conjunto de 4 bits que vai de C3-C0, e é a primeira parte da palavra a ser carregada. Os comandos possíveis de serem realizados são apresentados na Tabela 1.

Figura 15 – Palavra de Entrada.



Fonte: Linear Technology (2003).

Logo em seguida, os próximos 4 bits a serem enviados para o DAC serão o endereço da palavra. Esse endereço (*address*) corresponde à qual dos DACs (A, B, C, D, E, F G, H) deverão ter na sua saída a palavra convertida. Esse conjunto de 4 bits vai de A3-A0. Os endereços possíveis de serem realizados são apresentados na Tabela 2.

Finalmente, os 16 bits restantes que vão de D15-D0 representados do bit mais significativo (MSB) até o menos significativo (LSB), será a informação (e.g. corrente, tensão e potência) a ser enviada pelo mestre em formato binário, para posteriormente ser convertido pelo DAC e interpretado como sinal analógico no osciloscópio.

Tabela 1 – Comandos do LTC2600.

Comando				
C3	**C2**	**C1**	**C0**	
0	0	0	0	Escreve no registrador de entrada n
0	0	0	1	Atualiza o registrador DAC n
0	0	1	0	Escreve no registrador n e atualiza em todos registradores DACs
0	0	1	1	Escreve e atualiza o registrador n
0	1	0	0	Desliga registrador n
1	1	1	1	Sem operação

Fonte: Linear Technology (2003)

Tabela 2 – Endereços do LTC2600.

Endereço (n)				
A3	**A2**	**A1**	**A0**	
0	0	0	0	DAC A
0	0	0	1	DAC B
0	0	1	0	DAC C
0	0	1	1	DAC D
0	1	0	0	DAC E
0	1	0	1	DAC F
0	1	1	0	DAC G
0	1	1	1	DAC H
1	1	1	1	All DACs

Fonte: Linear Technology (2003)

Sabendo que o CS é "baixo ativo", os dados só podem ser transferidos para o dispositivo quando o sinal CS estiver no nível lógico 0. A borda de subida do CS termina a transferência de dados e faz com que o dispositivo pare de executar a ação especificada na palavra de entrada de 24 bits (Linear Technology, 2003).

4 Análise de Perdas

Este capítulo é voltado para a análise de perdas por condução e chaveamento na chave de um conversor estático e a interpretação da conversão dos sinais realizada pelo conversor D/A utilizado.

Esses cálculos das perdas na chave, se mostram difíceis de serem estimadas devido à natureza não linear do semicondutor, sendo proposta assim no trabalho uma alternativa para a estimação dessas perdas.

A importância do cálculo preciso das perdas leva à escolha da chave adequada para o projeto de conversores, bem como do seu dissipador de calor, quando necessário (AHMED, 2000).

É de suma importância que o projetista nunca subdimensione o cálculo das perdas, pois isso acarretaria em uma escolha da chave com menor capacidade de dissipação de energia reduzindo sua vida útil. Quando os cálculos são superdimensionados, leva-se à escolha de uma chave de custo bem mais elevado que o necessário (GRAOVAC; PURSCHEL; KIEP, 2006).

Dessa forma, com o auxílio do conversor D/A utilizado, foi feito um teste em bancada para que pudessem ser interpretados os diversos sinais e avaliados os seus comportamentos. Além disso, as medições realizadas tinham por objetivo validar se o conversor D/A realmente está exibindo as formas de onda esperadas dos sinais recebidos.

4.1 Sinais de PWM

Temos nas Figuras 16 - 17, os diversos sinais de PWM. O sinal de PWM é representado pela curva vermelha. Ao longo do projeto foram adotadas diversas combinações de chaves para compor o sinal PWM, variando o seu *duty cycle* (ciclo de trabalho) de modo a verificar essa influência no comportamento dos outros sinais, tais como corrente e tensão. Lembrando que o *duty cycle* de um sistema corresponde à fração do tempo total em que este se encontra em estado ativo, como visto na Equação 4.1. (FLUKE, 2019)

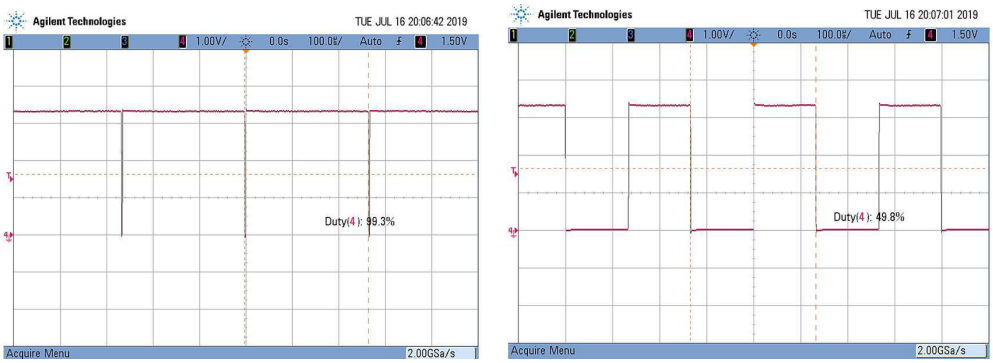
$$D = \frac{\tau}{T} \tag{4.1}$$

- D é o ciclo de trabalho;
- τ é o intervalo de tempo no qual a função é não-nula, dentro de cada período;

- T é o periodo da função.

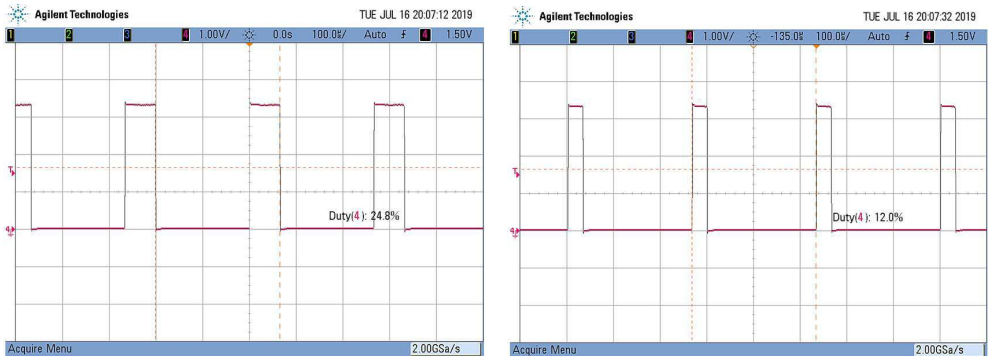
A configuração que foi adotada mais comumente foi aquela que possuía o *duty cycle* de 50%. Essa escolha foi feita em virtude dessa configuração nos fornecer melhores visualizações dos resultados, quando comparado à utilização das outras configurações possíveis.

Figura 16 – Sinais de PWM *Duty Cycle* de 100% à Esquerda e 50% à Direita.



Fonte: Autoria Própria.

Figura 17 – Sinais de PWM *Duty Cycle* de 25% à Esquerda e 12.5% à Direita.



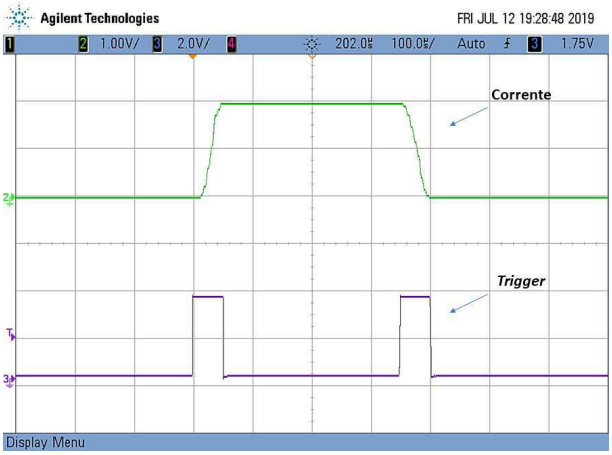
Fonte: Autoria Própria.

4.2 Sinais de Corrente e *Trigger*

Temos a seguir na Figura 18, os sinais de corrente e *trigger*. O sinal de *trigger* é responsável por indicar os instantes em que a chave de potência ainda está em transição de ligado para desligado ou vice-versa. O critério para caracterizar se o chaveamento ainda está ocorrendo, parte da premissa que enquanto não forem atingidos os valores de regime de corrente e tensão, o *trigger* deve permanecer em nível alto ativo.

Na Figura 18, o *trigger* é representado pela curva azul, enquanto que a corrente do conversor é representada pela curva verde. Como pode ser observado, no instante que a

Figura 18 – Sinais de Corrente e *Trigger*.



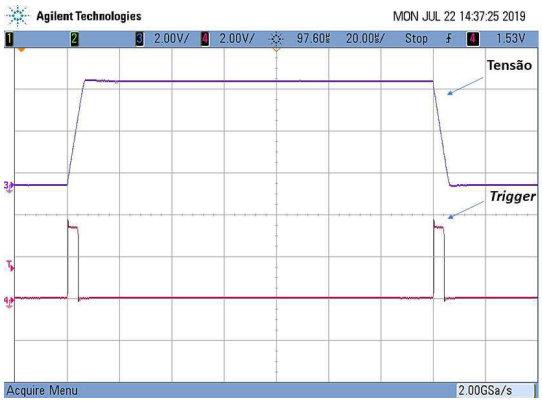
Fonte: Autoria Própria.

corrente inicia a transição do valor de 0A para 20A, o *trigger* permanece nível alto ativo até que o seu valor de regime seja atingido. Da mesma maneira, no instante que a corrente inicia a transição do valor de 20A para 0A, o *trigger* voltará a permanecer nível alto ativo até que o seu novo valor de regime seja atingido.

4.3 Sinais de Tensão e *Trigger*

Temos na Figura 19, os sinais de tensão e *trigger*. Como já foi mencionado, o sinal de *trigger* será capaz de captar os instantes que a chave de potência ainda está em transição de ligado para desligado ou vice-versa. Assim, enquanto não forem atingidos os valores de regime de corrente e tensão, o *trigger* deve permanecer em nível alto ativo.

Figura 19 – Sinais de Tensão e *Trigger*.



Fonte: Autoria Própria.

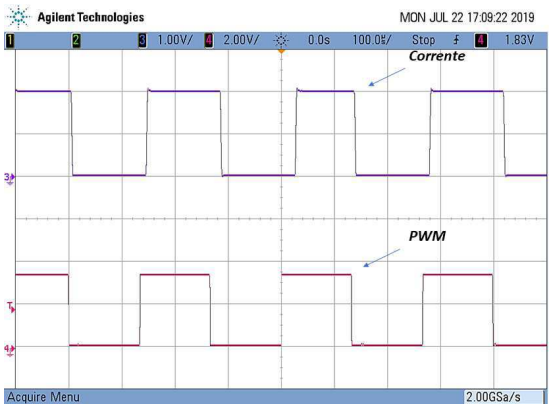
Na Figura 19, o *trigger* é representado pela curva vermelha, enquanto que a tensão do conversor é representada pela curva azul. Como pode ser observado, no instante que a

tensão inicia a transição do valor de 2V para 50V, o *trigger* permanece nível alto ativo até que o seu valor de regime seja atingido. Da mesma maneira, no instante que a tensão inicia a transição do valor de 50V para 2V, o *trigger* voltará a permanecer nível alto ativo até que o seu novo valor de regime seja atingido.

4.4 Sinais de Corrente e PWM

Temos na Figura 20, os sinais de corrente e PWM. O sinal de PWM é representado pela curva vermelha, enquanto que a corrente do conversor é representada pela curva azul. Pode ser constatado que na borda de subida do PWM (transição de 0 para 1), a chave será fechada e o valor de corrente começa a crescer até atingir o seu valor de regime de 20A . De modo análogo, na borda de descida do PWM (transição de 1 para 0), a chave será aberta e o valor de corrente começa a diminuir até atingir o seu novo valor de regime de 0A. Esse resultado era esperado visto que no instante que a chave abre, espera-se que a corrente que ali circulava comece a decair até cessar por completo.

Figura 20 – Sinais de Corrente e PWM.



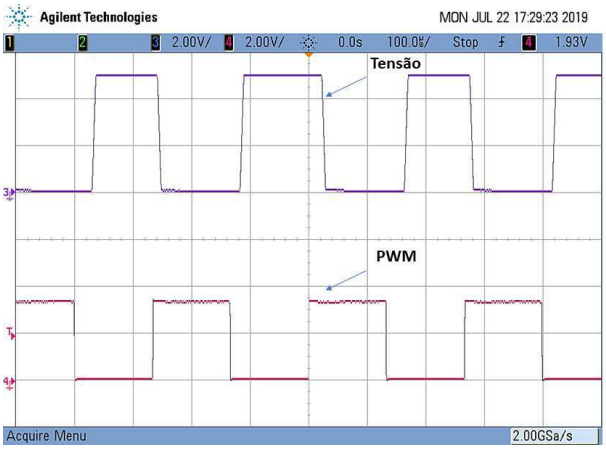
Fonte: Autoria Própria.

4.5 Sinais de Tensão e PWM

Temos na Figura 21, os sinais de tensão e PWM. O sinal de PWM é representado pela curva vermelha, enquanto que a tensão do conversor é representada pela curva azul. Pode ser constatado que na borda de subida do PWM (transição de 0 para 1), a chave será fechada e o valor de tensão começa a diminuir até atingir o seu valor de regime de 2V. De modo análogo, na borda de descida do PWM (transição de 1 para 0), a chave será aberta e o valor de tensão começa a crescer até atingir o seu novo valor de regime de 50V. Esse resultado era esperado, visto que no instante que a chave fecha, almeja-se que a

tensão que ali existia comece a decair até cessar por completo pelo fato de representar a tensão em um curto circuito.

Figura 21 – Sinais de Tensão e PWM.

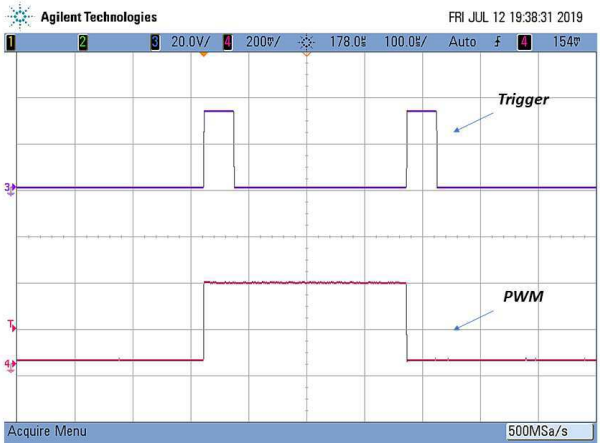


Fonte: Autoria Própria.

4.6 Sinais de *Trigger* e PWM

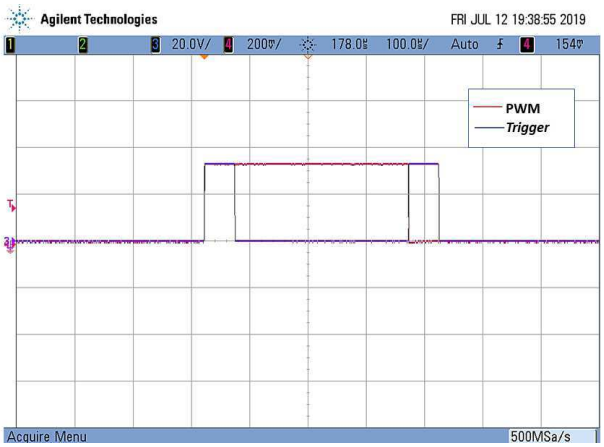
Temos na Figura 22, os sinais de *trigger* e PWM. O sinal de PWM é representado pela curva vermelha, enquanto que o *trigger* é representado pela curva azul. Podemos constatar que assim que ocorrem as transições do sinal de PWM, simultaneamente o sinal de *trigger* é ativado, caracterizando que nesse mesmo instante os valores de tensão e corrente começaram a crescer/decair.

Figura 22 – Sinais de *Trigger* e PWM.



Fonte: Autoria Própria.

Figura 23 – Sinais de *Trigger* e PWM Sobrepostos.



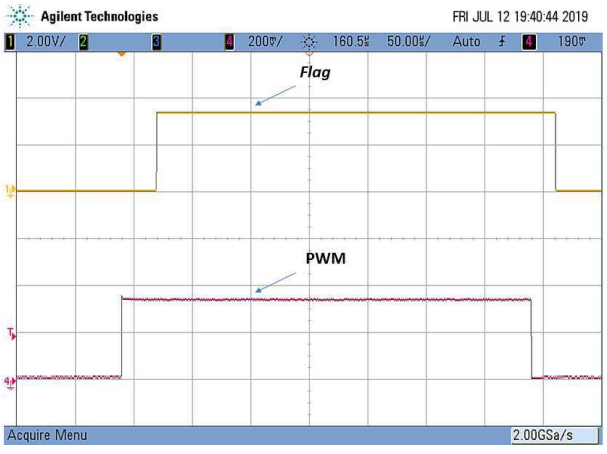
Fonte: Autoria Própria.

4.7 Sinais de *Flag* e PWM

Temos na Figura 24, os sinais de *pwm* e *flag*. O sinal de *flag* é representado pela curva amarela, enquanto que o *pwm* é representado pela curva vermelha.

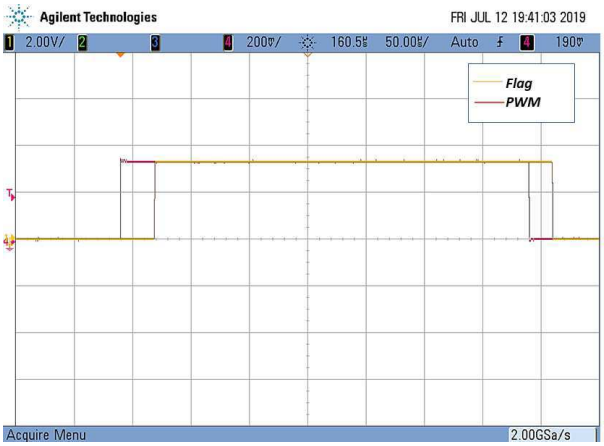
O sinal de *flag* também é responsável por atuar como uma espécie de *sniffer*, sendo ele capaz de captar os instantes que a chave de potência ainda está em estado de condução. O critério para caracterizar se a condução ainda está ocorrendo, está relacionado ao fato de que a partir de que a corrente for superior à 1A e a tensão inferior à 23V simultaneamente, o sinal de *flag* deverá permanecer nível alto ativo. Caso contrário, o sinal de *flag* permanecerá em nível baixo até que a corrente e a tensão voltem a caracterizar a condução novamente.

Figura 24 – Sinais de *Flag* e PWM.



Fonte: Autoria Própria.

Figura 25 – Sinais de *Flag* e PWM Sobrepostos.

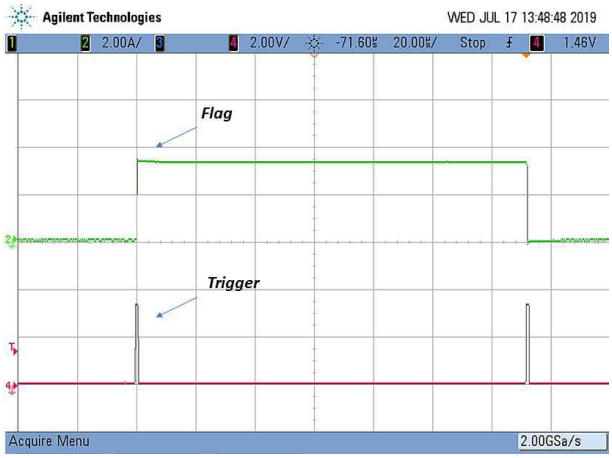


Fonte: Autoria Própria.

4.8 Sinais de *Trigger* e *Flag*

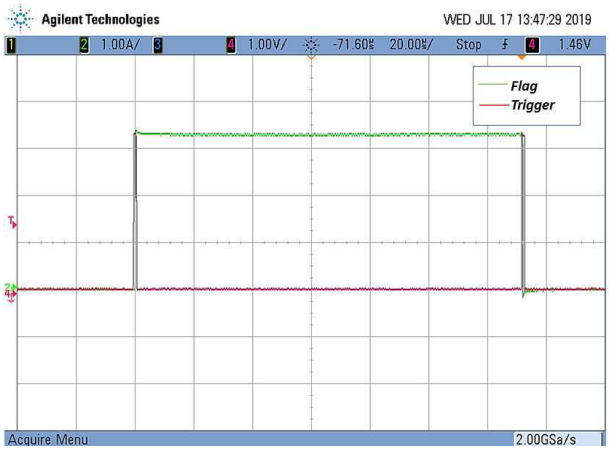
Temos na Figura 27, os sinais de *trigger* e *flag*. O sinal de *flag* é representado pela curva azul, enquanto que o *trigger* é representado pela curva verde. Esses sinais de *flag* e *trigger* representados serão utilizados para o cálculo das perdas por condução e por chaveamento, respectivamente.

Figura 26 – Sinais de *Trigger* e *Flag*.



Fonte: Autoria Própria.

Figura 27 – Sinais de *Trigger* e *Flag* Sobrepostos.

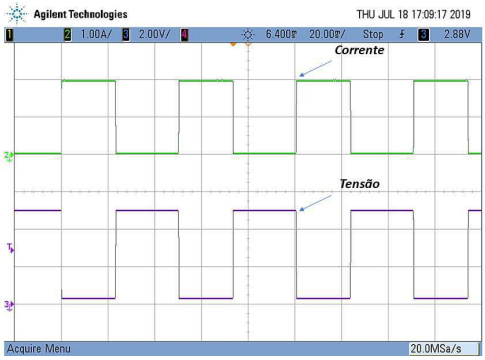


Fonte: Autoria Própria.

4.9 Sinais de Corrente e Tensão

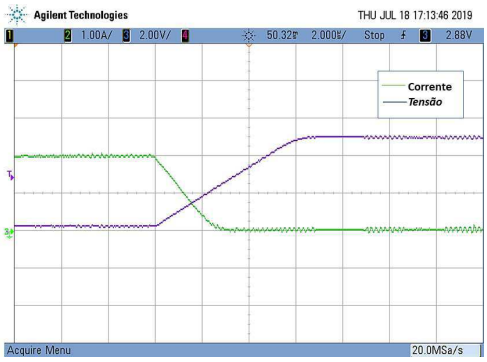
Temos na Figura 28, os sinais de corrente e tensão. O sinal de corrente é representado pela curva verde, enquanto que o sinal de tensão é representado pela curva azul. Podemos constatar a partir da Figura 28, que os seus sinais possuem comportamentos complementares, ou seja, nos instantes que a corrente começa a crescer a tensão começa a decair, e quando a corrente começa a decair a tensão começa a crescer para o seu valor de regime novamente. Isso pode ser verificado também na Figura 29 nos instantes de transição da corrente e tensão.

Figura 28 – Sinais de Corrente e Tensão.



Fonte: Autoria Própria.

Figura 29 – Sinais de Corrente e Tensão no Fechamento da Chave.



Fonte: Autoria Própria.

4.10 Sinais de *Trigger* e Perda de Potência por Chaveamento

Acerca dos sinais de *trigger* e das perdas de potência por chaveamento, podemos compreender que os cálculos de potência só existirão enquanto o *trigger* permanecer ativo. Fora desse escopo, o valor de potência calculada ao longo do período de chaveamento do PWM é nula.

4.11 Sinais de *Flag* e Perda de Potência por Condução

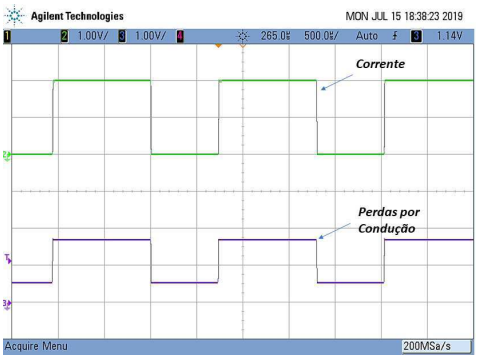
Acerca dos sinais de *flag* e das perdas de potência por condução, podemos compreender que os cálculos dessa potência só existirão enquanto o *flag* permanecer ativo. Fora desse escopo, o valor de potência calculada ao longo do período de chaveamento do PWM é nula.

4.12 Sinais de Corrente e Perda de Potência Instantânea por Condução

Temos na Figura 30, os sinais de corrente e das perdas de potência instantânea por condução. O sinal de corrente é representado pela curva verde, enquanto que as perdas por condução instantâneas são representadas pela curva azul. O intuito dessa figura é demonstrar que a medida que os valores de corrente aumentam ou diminuem, os valores das perdas de potência irão aumentar ou diminuir proporcionalmente.

Observando a Figura 31, é válido ressaltar que os sinais de corrente e das perdas instantâneas por condução são diretamente proporcionais, logo quando a corrente é máxima a perda de potência por condução também será máxima.

Figura 30 – Sinais de Corrente e Perda de Potência Instantânea por Condução.



Fonte: Autoria Própria.

Figura 31 – Sinais de Corrente e Perda de Potência Instantânea por Condução.

Fonte: Autoria Própria.

4.13 Sinais de Perdas de Potência Média por Chaveamento e por Condução

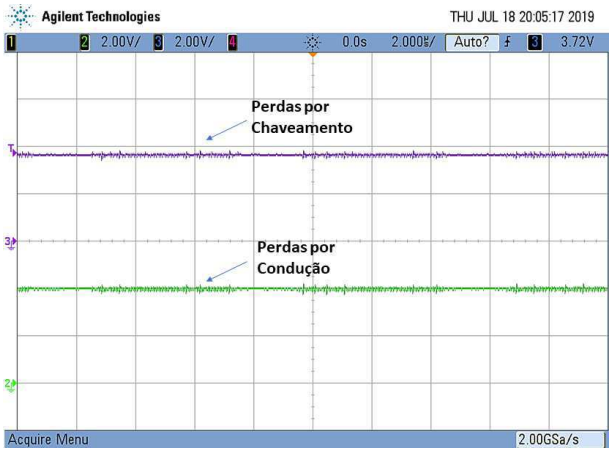
Temos na Figura 32, os sinais das perdas de potência por chaveamento e por condução. O sinal de perdas por chaveamento é representado pela curva azul, enquanto que as perdas por condução é representada pela curva verde.

Para realizar esse cálculo de potência das perdas por chaveamento e condução, basicamente serão multiplicados os valores de corrente e tensão nos instantes que os sinais de *flag* e *trigger* estiverem ativos, dentro de um período de chaveamento, como pode ser verificado nas equações 4.2 e 4.3.

No instante que o *trigger* estiver em nível lógico alto, é realizado o cálculo da potência de perdas por chaveamento dentro do período PWM:

$$P_{chav} = (V * I) * T_{PWM_{trigger}} \tag{4.2}$$

Figura 32 – Sinais de Perdas de Potência Média por Chaveamento e por Condução.



Fonte: Autoria Própria.

No instante que o *flag* estiver em nível lógico alto, é realizado o cálculo da potência de perdas por condução dentro do período PWM:

$$P_{cond} = (V * I) * T_{PWM_{flag}} \tag{4.3}$$

Levando-se em consideração a Equação 4.4 contida no *datasheet* do conversor LTC2600, teremos que a tensão de 3.7V representada pela curva azul na Figura 32, corresponde à nossa perda por chaveamento na prática de 37W, enquanto que para a tensão de 4V representada pela curva verde, corresponde à nossa perda por condução na prática de 40W (Linear Technology, 2003):

$$V_{OUT} = \left(\frac{k}{2^N}\right) V_{REF} \tag{4.4}$$

Em resumo, o valor da variável na prática corresponderia ao que fora visualizado no osciloscópio multiplicado por um fator de 10.

5 Conclusões

Esse trabalho objetivou a análise das perdas na chave em um conversor simulado, bem como os desdobramentos relativos ao conhecimento novo de como realizar a comunicação serial SPI e o manuseio correto dos equipamentos disponibilizados (e.g. osciloscópio e pontas de tensão), objetivando obter estimativas de perdas em conversores eletrônicos que fazem uso dessas chaves, de forma rápida e confiável.

Identificou-se na literatura uma falta de consenso para o cálculo das perdas por chaveamento, devido às naturezas não lineares presentes nos semicondutores, tais como, as capacitâncias parasitas. Além disso, é necessário fazer a interpretação correta das informações contidas nas folhas de dados dos fabricantes dos conversores D/A e dos CIs utilizados, e isso nem sempre é uma tarefa fácil, devido à falta de clareza das informações ali presentes. Portanto existe uma necessidade de um tutorial explicativo a respeito dessas informações.

Nesse trabalho podem ser identificadas as seguintes contribuições para análise de perdas em um conversor utilizando um módulo FPGA, que são:

- Implementação das equações de perdas em um simulador de linguagem de descrição de hardware;
- Comparação de diferentes resultados das simulações com as formas de onda que eram esperadas, como por exemplo, as transições realizadas dos sinais de corrente e tensão;
- Proposta da análise de perdas por chaveamento e condução a partir de módulos de medição contidos na implementação do hardware.

Quando a chave passa a operar dissipando uma potência bem maior que a sua potência nominal, resulta em uma considerável redução em sua vida útil. O cálculo equivocado no levantamento das perdas, levaria ao dimensionamento de uma chave de custo elevado, bem como, dissipador de calor de maior peso e volume.

As equações para o cálculo das perdas implementadas tanto por condução quanto por chaveamento no MOSFET, propiciaram um indicativo da chave adequada para aplicações em conversores de potência, bem como da topologia mais adequada em termos de perdas.

Como trabalhos futuros, planeja-se fazer uma análise das perdas geradas, porém num conversor físico com correntes e tensões reais, utilizando da arquitetura já implementada,

havendo mais tempo e condições para realizar as diversas análises. Ainda poderia-se utilizar de *kits* de desenvolvimento com um FPGA da ALTERA da linha Cyclone IV, um modelo mais avançado e com mais funcionalidades em relação à anterior Cyclone II, utilizada ao longo do projeto. Além disso, poderia-se incluir o efeito das capacitâncias parasitas nas equações de perdas do MOSFET, verificar a influência da temperatura na modelagem de perdas no MOSFET e estender o modelo de análise de perdas para chaves IGBT.

Referências Bibliográficas

AHMED, A. *Eletrônica de potência*. [S.l.]: Pearson Education do Brasil, 2000. ISBN 9788587918031. Citado na página 17.

ALTERA. *Cyclone II Device Handbook*. 2007. Disponível em: <https://tinyurl.com/y42fkkm6>. Citado 2 vezes nas páginas 2 e 3.

AnalogDialogue. *Introduction to SPI Interface*. 2019. Disponível em: <https://tinyurl.com/y5s3ku9q>. Citado na página 14.

Costa, C. da. Nova abordagem para o ensino de eletronica digital. *Saber Eletrônica*, v. 428, p. 26–32, Set 2008. Citado na página 6.

EPE. *Plano Nacional de Energia 2030*. 2007. Disponível em: <https://tinyurl.com/y63x9an6>. Citado na página 1.

eSpot. *Ponta de prova diferencial Keysight*. 2019. Disponível em: <https://tinyurl.com/y2hcwv8j>. Citado na página 10.

FLUKE. *Explaining electricity's fundamental components*. 2019. Disponível em: <https://tinyurl.com/y6ddx9e2>. Citado na página 17.

GRAOVAC, S.; PURSCHEL, M.; KIEP, A. *Mosfet Power Losses Calculation Using the Data-Sheet Parameters*. [S.l.]: INFINEON, 2006. Citado na página 17.

Keysight Technologies. *DSO7034A Oscilloscope*. 2019. Disponível em: <https://tinyurl.com/y3ddsxt3>. Citado na página 10.

Linear Technology. *Octal 16-/14-/12-Bit Rail-to-Rail DACs in 16-Lead SSOP*. 2003. Disponível em: <https://tinyurl.com/y2cmtu4m>. Citado 3 vezes nas páginas 15, 16 e 27.

Linear Technology. *DEMO MANUAL DC579A*. 2004. Disponível em: <https://tinyurl.com/yypzasmh>. Citado na página 15.

PINCLIPART. *Graphic Transparent Library Wires Clips Raspberry*. 2019. Disponível em: <https://tinyurl.com/y5kamxck>. Citado na página 11.

TERASIC. *Altera Corporation and Terasic Technologies*. 2014. Disponível em: <https://tinyurl.com/y5ugcjpe>. Citado 2 vezes nas páginas 3 e 5.

Total Phase. *Total Phase Knowledge Base Serial Protocols*. 2019. Disponível em: <https://tinyurl.com/y2qd6f6l>. Citado 3 vezes nas páginas 12, 13 e 14.

USINAINFO. *Cabo Flat para Pinos GPIO*. 2019. Disponível em: <https://tinyurl.com/y2wdcxka>. Citado na página 11.

ANEXO A – Códigos Implementados em Verilog

Neste apêndice é apresentado os códigos para a implementação dos módulos utilizados, em linguagem de descrição de *hardware Verilog*.

- Módulo PWM:

```
module PWM(
  input [7:0]duty_cycle,
  input clock,
  output reg [7:0]cont_pwm,
  output reg out_pwm);

  always@(posedge clock)
    begin
      if (duty_cycle > cont_pwm)
        out_pwm = 1'd1;
      else
        out_pwm = 1'd0;
      if(cont_pwm == 8'hFF)
        cont_pwm = 8'd0;
      else
        cont_pwm = cont_pwm + 8'd1;
    end
endmodule
```

- Módulo CORRENTE_TENSAO:

```
module CORRENTE_TENSAO_NOVO(
  input clock,out_pwm,
  output reg [7:0]corrente, tensao);

  reg auxc = 8'd1;          // A corrente inicia crescendo
  reg auxv = 8'd0;          // A tensao inicia decrescendo
  reg inicio = 1'd1;          // Para colocar a tensao inicial de 50V

  always@(posedge clock)
    begin
      if(inicio)
        begin
```

```
13            corrente <= 8'h00;
14            tensao <= 8'd50;
15            inicio <= 1'd0;
16          end
17
18        if((auxc == 1) && (out_pwm == 1))
19          begin
20            if(corrente == 8'd20)       // Se a corrente atingir 20A ...
    (2A vezes 10)
21            auxc = 1'd0;
22            else
23            corrente <= corrente + 8'd1;   //  Incrementar a corrente
24          end
25
26        if((auxc == 0) && (out_pwm == 0))
27          begin
28            if(corrente == 8'h00)       // Se a corrente atingir 0A
29            auxc = 1'd1;
30            else
31            corrente <= corrente - 8'd1;   //  Decrementar a corrente
32          end
33
34        if((auxv == 1) && (out_pwm == 0))
35          begin
36            if(tensao == 8'd50)       // Se a tensao atingir 50V (5V ...
    vezes 10)
37            auxv = 1'd0;
38            else
39            tensao <= tensao + 8'd1;   // Incrementar a tensao
40          end
41
42        if((auxv == 0) && (out_pwm == 1))
43          begin
44            if(tensao == 8'd02)       // Se a tensao atingir 2V
45            auxv = 1'd1;
46            else
47            tensao <= tensao - 8'd1;   //  Decrementar a tensao
48          end
49      end
50  endmodule
```

- Módulo TRIGGER:

```
module TRIGGER(
  input [7:0]corrente, tensao,
  input clock, out_pwm,
  input [7:0]cont_pwm,
  output reg flag, trig,
  output reg [7:0]cont_alternancia_flag,
  output reg [15:0]potencia_med_cond_final, potencia_med_chav_final...
   );

  reg [15:0]potencia_med_cond, potencia_med_chav;
  reg aux_med;

  always@(posedge clock)
    begin
      if ((tensao < 23) && (corrente > 1))  // Tensao menor que 2.3...
   v e corrente maior que 0.1 caracteriza a chave conduzindo
        flag = 1;        // flag = 1 -> conduzindo
      else
        flag = 0;        // flag = 0 -> nao conduzindo

      if(out_pwm == 1)  // Chave fechada
            begin
          if((corrente == 8'd20) && (tensao == 8'd02)) // trigger =...
    0 se i e v estarao em regime ( i = 20A e V = 2V)
            trig = 0;
          else
            trig = 1; //pwm sobe = trigger 1
          end
      else       //Chave aberta
            begin
          if((corrente == 8'd00) && (tensao == 8'd50)) // trigger =...
    0 se i e v estarao em regime (i = 0A e V = 50V)
            trig = 0;
          else
            trig = 1; //pwm sobe = trigger 1
            end

      if (flag)
        begin
          potencia_med_cond = (potencia_med_cond + tensao*corrente)...
    ;
        end

      if (trig)
```

```
40        begin
41            potencia_med_chav = (potencia_med_chav + tensao*corrente)...
    ;
42        end
43
44        if(cont_pwm == 8'hFF)
45        begin
46          if(aux_med == 0)
47          begin
48            potencia_med_chav_final = potencia_med_chav / 12750;
49            potencia_med_cond_final = potencia_med_cond / 12750;
50            aux_med = 1;
51          end
52        end
53      end
54
55    always@(flag)
56         begin
57             cont_alternancia_flag = cont_alternancia_flag + 8'd1; //...
    Verificar a quantidade de vezes que o flag mudou dentro do ...
    periodo de chaveamento
58         end
59  endmodule
```

- Módulo DIV_FREQ_PWM:

```
1   module DIVFREQ_CHAV_PWM(
2     input clock50,
3     output reg clock_novo);
4
5     reg [24:0]cont;
6
7     always @(posedge clock50)
8       begin
9
10        if(cont == 25'd1250) // Para frequencia de 20khz temos ...
    (25000000/1250) e periodo 50us
11        begin
12          cont <= 25'd0;
13          clock_novo = ¬clock_novo;
14        end
15      else
16        cont <= cont + 25'd1;
17      end
18  endmodule
```

- Módulo DIV_FREQ_CHAV:

```
module DIVFREQ_TRIG(
  input clock50,
  output reg clock_novo);

  reg [24:0]cont;

  always @(posedge clock50)
    begin

      if(cont == 25'd25) // Para frequencia de 1Mhz temos ...
    (25000000/25) e periodo 1000ns
      begin
        cont <= 25'd0;
        clock_novo = ¬clock_novo;
      end

    else
      cont <= cont + 25'd1;
    end
endmodule
```

- Módulo SPI_MASTER:

```
//////////////////////////////////////////
// Description: SPI (Serial Peripheral Interface) Master
//              Creates master based on input configuration.
//              Sends a byte one bit at a time on MOSI
//              Will also receive byte data one bit at a time on ...
    MISO.
//              Any data on input byte will be shipped out on MOSI.
//
//              To kick-off transaction, user must pulse i_TX_DV.
//              This module supports multi-byte transmissions by ...
    pulsing
//              i_TX_DV and loading up i_TX_Byte when o_TX_Ready is...
     high.
//
//              This module is only responsible for controlling Clk...
    , MOSI,
//              and MISO.  If the SPI peripheral requires a chip-...
    select,
//              this must be done at a higher level.
//
```

```
16 // Note:         i_Clk must be at least 2x faster than i_SPI_Clk
17 //
18 // Parameters:  SPI_MODE, can be 0, 1, 2, or 3.  See above.
19 //              Can be configured in one of 4 modes:
20 //              Mode | Clock Polarity (CPOL/CKP) | Clock Phase (...
   CPHA)
21 //               0   |             0              |        0
22 //               1   |             0              |        1
23 //               2   |             1              |        0
24 //               3   |             1              |        1
25 //              More: https://en.wikipedia.org/wiki/...
   Serial_Peripheral_Interface_Bus#Mode_numbers
26 //              CLKS_PER_HALF_BIT - Sets frequency of o_SPI_Clk.  ...
   o_SPI_Clk is
27 //              derived from i_Clk.  Set to integer number of ...
   clocks for each
28 //              half-bit of SPI data.  E.g. 100 MHz i_Clk, ...
   CLKS_PER_HALF_BIT = 2
29 //              would create o_SPI_CLK of 25 MHz.  Must be ≥ 2
30 //
31 ////////////////////////////////////
32
33 module SPI_Master
34   #(parameter SPI_MODE = 0,
35     parameter CLKS_PER_HALF_BIT = 2)
36   (
37    // Control/Data Signals,
38    input         i_Rst_L,     // FPGA Reset
39    input         i_Clk,       // FPGA Clock
40
41    // TX (MOSI) Signals
42    input [7:0]   i_TX_Byte,        // Byte to transmit on MOSI
43    input         i_TX_DV,          // Data Valid Pulse with ...
   i_TX_Byte
44    output reg    o_TX_Ready,       // Transmit Ready for next byte
45
46    // RX (MISO) Signals
47    output reg       o_RX_DV,     // Data Valid pulse (1 clock cycle...
   )
48    output reg [7:0] o_RX_Byte,   // Byte received on MISO
49
50    // SPI Interface
51    output reg o_SPI_Clk,
52    input      i_SPI_MISO,
53    output reg o_SPI_MOSI
54    );
55
```

```
56  // SPI Interface (All Runs at SPI Clock Domain)
57  wire w_CPOL;     // Clock polarity
58  wire w_CPHA;     // Clock phase
59
60  reg [$clog2(CLKS_PER_HALF_BIT*2)-1:0] r_SPI_Clk_Count;
61  reg r_SPI_Clk;
62  reg [4:0] r_SPI_Clk_Edges;
63  reg r_Leading_Edge;
64  reg r_Trailing_Edge;
65  reg       r_TX_DV;
66  reg [7:0] r_TX_Byte;
67
68  reg [2:0] r_RX_Bit_Count;
69  reg [2:0] r_TX_Bit_Count;
70
71  // CPOL: Clock Polarity
72  // CPOL=0 means clock idles at 0, leading edge is rising edge.
73  // CPOL=1 means clock idles at 1, leading edge is falling edge.
74  assign w_CPOL  = (SPI_MODE == 2) | (SPI_MODE == 3);
75
76  // CPHA: Clock Phase
77  // CPHA=0 means the "out" side changes the data on trailing edge ...
     of clock
78  //             the "in" side captures data on leading edge of ...
     clock
79  // CPHA=1 means the "out" side changes the data on leading edge ...
     of clock
80  //             the "in" side captures data on the trailing edge ...
     of clock
81  assign w_CPHA  = (SPI_MODE == 1) | (SPI_MODE == 3);
82
83
84
85  // Purpose: Generate SPI Clock correct number of times when DV ...
     pulse comes
86  always @(posedge i_Clk or negedge i_Rst_L)
87  begin
88    if (~i_Rst_L)
89    begin
90      o_TX_Ready      <= 1'b0;
91      r_SPI_Clk_Edges <= 0;
92      r_Leading_Edge  <= 1'b0;
93      r_Trailing_Edge <= 1'b0;
94      r_SPI_Clk       <= w_CPOL; // assign default state to idle ...
     state
95      r_SPI_Clk_Count <= 0;
96    end
```

```
97      else
98      begin
99
100       // Default assignments
101       r_Leading_Edge  <= 1'b0;
102       r_Trailing_Edge <= 1'b0;
103
104       if (i_TX_DV)
105       begin
106         o_TX_Ready       <= 1'b0;
107         r_SPI_Clk_Edges = 16;  // Total # edges in one byte ALWAYS ...
    16
108       end
109       else if (r_SPI_Clk_Edges > 0)
110       begin
111         o_TX_Ready <= 1'b0;
112
113         if (r_SPI_Clk_Count == CLKS_PER_HALF_BIT*2-1)
114         begin
115           r_SPI_Clk_Edges <= r_SPI_Clk_Edges - 1;
116           r_Trailing_Edge <= 1'b1;
117           r_SPI_Clk_Count <= 0;
118           r_SPI_Clk       <= ~r_SPI_Clk;
119         end
120         else if (r_SPI_Clk_Count == CLKS_PER_HALF_BIT-1)
121         begin
122           r_SPI_Clk_Edges <= r_SPI_Clk_Edges - 1;
123           r_Leading_Edge  <= 1'b1;
124           r_SPI_Clk_Count <= r_SPI_Clk_Count + 1;
125           r_SPI_Clk       <= ~r_SPI_Clk;
126         end
127         else
128         begin
129           r_SPI_Clk_Count <= r_SPI_Clk_Count + 1;
130         end
131       end
132       else
133       begin
134         o_TX_Ready <= 1'b1;
135       end
136
137
138     end // else: !if(~i_Rst_L)
139   end // always @ (posedge i_Clk or negedge i_Rst_L)
140
141
142   // Purpose: Register i_TX_Byte when Data Valid is pulsed.
```

```
143   // Keeps local storage of byte in case higher level module ...
       changes the data
144   always @(posedge i_Clk or negedge i_Rst_L)
145   begin
146     if (¬i_Rst_L)
147     begin
148       r_TX_Byte ≤ 8'h00;
149       r_TX_DV   ≤ 1'b0;
150     end
151     else
152       begin
153         r_TX_DV ≤ i_TX_DV; // 1 clock cycle delay
154         if (i_TX_DV)
155         begin
156           r_TX_Byte ≤ i_TX_Byte;
157         end
158       end // else: !if(¬i_Rst_L)
159   end // always @ (posedge i_Clk or negedge i_Rst_L)
160
161
162   // Purpose: Generate MOSI data
163   // Works with both CPHA=0 and CPHA=1
164   always @(posedge i_Clk or negedge i_Rst_L)
165   begin
166     if (¬i_Rst_L)
167     begin
168       o_SPI_MOSI     ≤ 1'b0;
169       r_TX_Bit_Count ≤ 3'b111; // send MSb first
170     end
171     else
172     begin
173       // If ready is high, reset bit counts to default
174       if (o_TX_Ready)
175       begin
176         r_TX_Bit_Count ≤ 3'b111;
177       end
178       // Catch the case where we start transaction and CPHA = 0
179       else if (r_TX_DV & ¬w_CPHA)
180       begin
181         o_SPI_MOSI     ≤ r_TX_Byte[3'b111];
182         r_TX_Bit_Count ≤ 3'b110;
183       end
184       else if ((r_Leading_Edge & w_CPHA) | (r_Trailing_Edge & ¬...
     w_CPHA))
185       begin
186         r_TX_Bit_Count ≤ r_TX_Bit_Count - 1;
187         o_SPI_MOSI     ≤ r_TX_Byte[r_TX_Bit_Count];
```

```
188       end
189     end
190   end
191
192
193   // Purpose: Read in MISO data.
194   always @(posedge i_Clk or negedge i_Rst_L)
195   begin
196     if (~i_Rst_L)
197     begin
198       o_RX_Byte       <= 8'h00;
199       o_RX_DV         <= 1'b0;
200       r_RX_Bit_Count <= 3'b111;
201     end
202     else
203     begin
204
205       // Default Assignments
206       o_RX_DV   <= 1'b0;
207
208       if (o_TX_Ready) // Check if ready is high, if so reset bit ...
    count to default
209       begin
210         r_RX_Bit_Count <= 3'b111;
211       end
212       else if ((r_Leading_Edge & ~w_CPHA) | (r_Trailing_Edge & ...
    w_CPHA))
213       begin
214         o_RX_Byte[r_RX_Bit_Count] <= i_SPI_MISO;  // Sample data
215         r_RX_Bit_Count            <= r_RX_Bit_Count - 1;
216         if (r_RX_Bit_Count == 3'b000)
217         begin
218           o_RX_DV   <= 1'b1;   // Byte done, pulse Data Valid
219         end
220       end
221     end
222   end
223
224
225   // Purpose: Add clock delay to signals for alignment.
226   always @(posedge i_Clk or negedge i_Rst_L)
227   begin
228     if (~i_Rst_L)
229     begin
230       o_SPI_Clk  <= w_CPOL;
231     end
232     else
```

```
233       begin
234         o_SPI_Clk <= r_SPI_Clk;
235       end // else: !if(~i_Rst_L)
236   end // always @ (posedge i_Clk or negedge i_Rst_L)
237
238
239 endmodule // SPI_Master
```

- Módulo SPI_MASTER_with_single_CS:

```
1  ////////////////////////////////////////
2  // Description: SPI (Serial Peripheral Interface) Master
3  //              With single chip-select (AKA Slave Select) ...
       capability
4  //
5  //              Supports arbitrary length byte transfers.
6  //
7  //              Instantiates a SPI Master and adds single CS.
8  //              If multiple CS signals are needed, will need to use...
       different
9  //              module, OR multiplex the CS from this at a higher ...
       level.
10 //
11 // Note:        i_Clk must be at least 2x faster than i_SPI_Clk
12 //
13 // Parameters:  SPI_MODE, can be 0, 1, 2, or 3.  See above.
14 //              Can be configured in one of 4 modes:
15 //              Mode | Clock Polarity (CPOL/CKP) | Clock Phase (...
       CPHA)
16 //               0   |             0             |        0
17 //               1   |             0             |        1
18 //               2   |             1             |        0
19 //               3   |             1             |        1
20 //
21 //              CLKS_PER_HALF_BIT - Sets frequency of o_SPI_Clk.  ...
       o_SPI_Clk is
22 //              derived from i_Clk.  Set to integer number of ...
       clocks for each
23 //              half-bit of SPI data.  E.g. 100 MHz i_Clk, ...
       CLKS_PER_HALF_BIT = 2
24 //              would create o_SPI_CLK of 25 MHz.  Must be ≥ 2
25 //
26 //              MAX_BYTES_PER_CS - Set to the maximum number of ...
       bytes that
27 //              will be sent during a single CS-low pulse.
28 //
```

```
29 //              CS_INACTIVE_CLKS - Sets the amount of time in clock...
       cycles to
30 //              hold the state of Chip-Selct high (inactive) before...
       next
31 //              command is allowed on the line.  Useful if chip ...
      requires some
32 //              time when CS is high between trasnfers.
33 ////////////////////////////////////////
34
35 module SPI_Master_With_Single_CS
36   #(parameter SPI_MODE = 0,
37     parameter CLKS_PER_HALF_BIT = 2,
38     parameter MAX_BYTES_PER_CS = 2,
39     parameter CS_INACTIVE_CLKS = 1)
40   (
41    // Control/Data Signals,
42    input         i_Rst_L,      // FPGA Reset
43    input         i_Clk,        // FPGA Clock
44
45    // TX (MOSI) Signals
46    input [$clog2(MAX_BYTES_PER_CS+1)-1:0] i_TX_Count,  // # bytes ...
      per CS low
47    input [7:0]   i_TX_Byte,        // Byte to transmit on MOSI
48    input         i_TX_DV,          // Data Valid Pulse with i_TX_Byte
49    output        o_TX_Ready,       // Transmit Ready for next byte
50
51    // RX (MISO) Signals
52    output reg [$clog2(MAX_BYTES_PER_CS+1)-1:0] o_RX_Count,  // ...
      Index RX byte
53    output        o_RX_DV,      // Data Valid pulse (1 clock cycle)
54    output [7:0] o_RX_Byte,    // Byte received on MISO
55
56    // SPI Interface
57    output o_SPI_Clk,
58    input  i_SPI_MISO,
59    output o_SPI_MOSI,
60    output o_SPI_CS_n
61    );
62
63   localparam IDLE        = 2'b00;
64   localparam TRANSFER    = 2'b01;
65   localparam CS_INACTIVE = 2'b10;
66
67   reg [1:0] r_SM_CS;
68   reg r_CS_n;
69   reg [$clog2(CS_INACTIVE_CLKS)-1:0] r_CS_Inactive_Count;
70   reg [$clog2(MAX_BYTES_PER_CS+1)-1:0] r_TX_Count;
```

```
71  wire w_Master_Ready;
72
73  // Instantiate Master
74  SPI_Master
75    #(.SPI_MODE(SPI_MODE),
76      .CLKS_PER_HALF_BIT(CLKS_PER_HALF_BIT)
77      ) SPI_Master_Inst
78   (
79   // Control/Data Signals,
80   .i_Rst_L(i_Rst_L),      // FPGA Reset
81   .i_Clk(i_Clk),          // FPGA Clock
82
83   // TX (MOSI) Signals
84   .i_TX_Byte(i_TX_Byte),          // Byte to transmit
85   .i_TX_DV(i_TX_DV),              // Data Valid Pulse
86   .o_TX_Ready(w_Master_Ready),    // Transmit Ready for Byte
87
88   // RX (MISO) Signals
89   .o_RX_DV(o_RX_DV),        // Data Valid pulse (1 clock cycle)
90   .o_RX_Byte(o_RX_Byte),    // Byte received on MISO
91
92   // SPI Interface
93   .o_SPI_Clk(o_SPI_Clk),
94   .i_SPI_MISO(i_SPI_MISO),
95   .o_SPI_MOSI(o_SPI_MOSI)
96   );
97
98
99  // Purpose: Control CS line using State Machine
100 always @(posedge i_Clk or negedge i_Rst_L)
101 begin
102   if (~i_Rst_L)
103   begin
104     r_SM_CS <= IDLE;
105     r_CS_n  <= 1'b1;   // Resets to high
106     r_TX_Count <= 0;
107     r_CS_Inactive_Count <= CS_INACTIVE_CLKS;
108   end
109   else
110   begin
111
112     case (r_SM_CS)
113     IDLE:
114       begin
115         if (r_CS_n & i_TX_DV) // Start of transmission
116         begin
117           r_TX_Count <= i_TX_Count - 1; // Register TX Count
```

```
118             r_CS_n     <= 1'b0;         // Drive CS low
119             r_SM_CS    <= TRANSFER;     // Transfer bytes
120           end
121         end
122
123       TRANSFER:
124         begin
125           // Wait until SPI is done transferring do next thing
126           if (w_Master_Ready)
127           begin
128             if (r_TX_Count > 0)
129             begin
130               if (i_TX_DV)
131               begin
132                 r_TX_Count <= r_TX_Count - 1;
133               end
134             end
135             else
136             begin
137               r_CS_n  <= 1'b1; // we done, so set CS high
138               r_CS_Inactive_Count <= CS_INACTIVE_CLKS;
139               r_SM_CS             <= CS_INACTIVE;
140             end // else: !if(r_TX_Count > 0)
141           end // if (w_Master_Ready)
142         end // case: TRANSFER
143
144       CS_INACTIVE:
145         begin
146           if (r_CS_Inactive_Count > 0)
147           begin
148             r_CS_Inactive_Count <= r_CS_Inactive_Count - 1'b1;
149           end
150           else
151           begin
152             r_SM_CS <= IDLE;
153           end
154         end
155
156       default:
157         begin
158           r_CS_n  <= 1'b1; // we done, so set CS high
159           r_SM_CS <= IDLE;
160         end
161       endcase // case (r_SM_CS)
162     end
163   end // always @ (posedge i_Clk or negedge i_Rst_L)
164
```

```
165
166   // Purpose: Keep track of RX_Count
167   always @(posedge i_Clk)
168   begin
169     begin
170       if (r_CS_n)
171       begin
172         o_RX_Count <= 0;
173       end
174       else if (o_RX_DV)
175       begin
176         o_RX_Count <= o_RX_Count + 1'b1;
177       end
178     end
179   end
180
181   assign o_SPI_CS_n = r_CS_n;
182
183   assign o_TX_Ready  = ((r_SM_CS == IDLE) | (r_SM_CS == TRANSFER &&...
        w_Master_Ready == 1'b1 && r_TX_Count > 0)) & ~i_TX_DV;
184
185 endmodule // SPI_Master_With_Single_CS
```

- Módulo MOD_TESTE:

```
1  //Samuel de Melo Barros
2
3  `default_nettype none        //Comando para desabilitar declaracao ...
       automatica de wires
4
5  module TCC_Analise_de_Perdas_Samuel(
6  //Clocks
7  input CLOCK_27, CLOCK_50,
8
9  //Chaves e Botoes
10 input [3:0] KEY,
11 input [17:0] SW,
12
13 //Displays de 7 seg e LEDs
14 output [0:6] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7,
15 output [8:0] LEDG,
16 output [17:0] LEDR,
17
18 //Serial
19 output UART_TXD,
20 input UART_RXD,
```

```
21 inout [7:0] LCD_DATA,
22 output LCD_ON, LCD_BLON, LCD_RW, LCD_EN, LCD_RS,
23
24 //GPIO
25 inout    [35:0] GPIO_0,GPIO_1
26 );
27
28 assign GPIO_1    = 36'hzzzzzzzzz;
29 assign GPIO_0    = 36'hzzzzzzzzz;
30
31 assign LCD_ON = 1'b1;
32 assign LCD_BLON = 1'b1;
33
34 wire [7:0] w_d0x0, w_d0x1, w_d0x2, w_d0x3, w_d0x4, w_d0x5,
35        w_d1x0, w_d1x1, w_d1x2, w_d1x3, w_d1x4, w_d1x5;
36
37 LCD_TEST MyLCD (
38       .iCLK ( CLOCK_50 ),
39       .iRST_N ( KEY[0] ),
40       .d0x0(w_d0x0),.d0x1(w_d0x1),.d0x2(w_d0x2),.d0x3(w_d0x3),.d0x4...
     (w_d0x4),.d0x5(w_d0x5),
41       .d1x0(w_d1x0),.d1x1(w_d1x1),.d1x2(w_d1x2),.d1x3(w_d1x3),.d1x4...
     (w_d1x4),.d1x5(w_d1x5),
42       .LCD_DATA( LCD_DATA ),
43       .LCD_RW ( LCD_RW ),
44       .LCD_EN ( LCD_EN ),
45       .LCD_RS ( LCD_RS )
46 );
47 //------modifique a partir daqui-----
48
49 wire clktrig, clkpwm ,wire_pwm, wire_trig, wire_flag;
50
51 wire w_DAC_SCK;
52 wire w_DAC_CS;
53 wire w_DAC_MOSI;
54 wire w_u_DAC_MISO; // Unused
55
56 wire [7:0] wire_corrente, wire_tensao,wire_cont_pwm;
57 wire [15:0] wire_potencia_cond, wire_potencia_chav;
58
59 assign LEDR[0] = wire_pwm;
60 assign LEDG[0] = wire_flag;
61 assign LEDG[4] = wire_trig;
62 assign GPIO_0[0] = wire_pwm;
63 assign GPIO_0[3] = wire_trig;
64 assign GPIO_0[6] = wire_flag;
65
```

```
66
67 //assign wire_corrente = w_d0x4;
68 //assign wire_tensao = w_d1x4;
69
70 assign w_d0x2 = wire_potencia_cond[15:8];
71 assign w_d0x3 = wire_potencia_cond[7:0];
72
73 assign w_d1x2 = wire_potencia_chav[15:8];
74 assign w_d1x3 = wire_potencia_chav[7:0];
75
76 // Pinos do mdulo DAC
77 assign GPIO_0[1] = w_DAC_SCK;
78 assign GPIO_0[4] = w_DAC_MOSI;
79 assign GPIO_0[5] = w_DAC_CS;
80
81 ////////////////////////////////////
82
83 DIVFREQ_TRIG div1(.clock50(CLOCK_50), .clock_novo(clkpwm));
84
85 DIVFREQ_CHAV_PWM div2(.clock50(CLOCK_50), .clock_novo(clktrig));
86
87 PWM p1(.clock(clkpwm), .duty_cycle(SW[7:0]), .out_pwm(wire_pwm), ....
    cont_pwm(wire_cont_pwm));
88
89 CORRENTE_TENSAO_NOVO c1(.clock(CLOCK_50), .out_pwm(wire_pwm),....
    corrente(wire_corrente), .tensao(wire_tensao));
90
91 TRIGGER t1(.clock(CLOCK_50), .out_pwm(wire_pwm), .corrente(...
    wire_corrente), .tensao(wire_tensao), .cont_pwm(wire_cont_pwm),....
    flag(wire_flag), .trig(wire_trig),.potencia_med_cond_final(...
    wire_potencia_cond), .potencia_med_chav_final(wire_potencia_chav...
    ),.cont_alternancia_flag(w_d1x4));
92
93 wire  [15:0]  w_word_corrente = wire_corrente * 1310;   // w_word =...
     tensao_que_quero * 5 * 1310 / 65536 que eh o mesmo que fulano ...
    da esquerda dividido por 10
94                                                  // w_word eh o nosso K ...
    da formula, 2^N = 2^16 = 65536, e Vref na nossa escala vai ate ...
    50, 1310 = 65536/50
95 wire  [15:0]  w_word_tensao = wire_tensao * 1310;
96
97 //wire  [15:0]  w_word_potencia_chav = wire_corrente * 1310;    // ...
    ajeitar depois la embaixo //o verde eh chaveamento   // w_word =...
     tensao_que_quero * 5 * 1310 / 65536 que eh o mesmo que fulano ...
    da esquerda dividido por 10
98 //                                                              // ...
    w_word eh o nosso K da formula, 2^N = 2^16 = 65536, e Vref na ...
```

```
    nossa escala vai ate 50, 1310 = 65536/50
//wire  [15:0]  w_word_potencia_cond = wire_tensao * 1310;    //o ...
    azul eh conducao

wire  [7:0]   w_Command_corrente = {4'b0000, 4'b0010};      // ADC ...
    c
wire  [7:0]   w_Command_tensao = {4'b0000, 4'b0011};        // ADC ...
    D

reg   [2:0]   r_ByteContador = 0;

//reg   [23:0]  MOSI_Message; // Mensagem a ser enviada pelo MOSI
reg   [7:0]   r_QueueByte = 0;  // Byte que vai ser enviado pela ...
    SPI

// CPOL = 1 and CPHA = 1, based on ADC part ADC08S021
localparam SPI_MODE = 3;

// Go Board operates at 25 MHz, so divide by 10 to get to 2.5 MHz.
// This is clocks per half bit, so divide 10 by 2 to get 5.
localparam CLKS_PER_HALF_BIT = 5;

// Number of clock cycles to leave CS high after transaction is ...
    done
localparam CS_INACTIVE_CLKS = 100;  // check this value

// SPI Signals
wire [1:0] w_u_Master_RX_Count;    // u_  -> Unused(nao usado)
wire w_u_Master_RX_DV;
wire [7:0] w_u_Master_RX_Byte;

wire w_Master_TX_Ready;
reg r_Master_TX_DV;

assign LEDR[4] = w_Master_TX_Ready;

// Reset do SPI na chave 17
wire w_Rst_L = SW[17];

SPI_Master_With_Single_CS
    #(.SPI_MODE(SPI_MODE),
      .CLKS_PER_HALF_BIT(CLKS_PER_HALF_BIT),
      .MAX_BYTES_PER_CS(2),  // Always 2 bytes per CS
      .CS_INACTIVE_CLKS(CS_INACTIVE_CLKS)
    ) SPI_Master_CS_Inst
  (
   // Control/Data Signals,
```

```
140     .i_Rst_L(w_Rst_L),      // FPGA Reset
141     .i_Clk(CLOCK_50),       // FPGA Clock
142
143     // TX (MOSI) Signals
144     .i_TX_Count(2'b11),             // Always 3 bytes per CS     ...
            / i_ -> Input alguma coisa
145     .i_TX_Byte(r_QueueByte),        // Byte que estamos enviando ...
        para a SPI
146     .i_TX_DV(r_Master_TX_DV),       // Data Valid Pulse with ...
        i_TX_Byte  Mod_Teste dizendo que o dado ta na mao
147     .o_TX_Ready(w_Master_TX_Ready), // Transmit Ready for Byte   ...
            SPI dizendo que esta pronto para enviar dado
148
149     // RX (MISO) Signals
150     .o_RX_Count(w_u_Master_RX_Count), // Index of RX'd byte        ...
              / o_ -> Output do modulo implementado
151     .o_RX_DV(w_u_Master_RX_DV),     // Data Valid pulse (1 clock ...
        cycle)
152     .o_RX_Byte(w_u_Master_RX_Byte), // Byte received on MISO
153
154     // SPI Interface
155     .o_SPI_Clk(w_DAC_SCK),
156     .i_SPI_MISO(w_u_DAC_MISO),
157     .o_SPI_MOSI(w_DAC_MOSI),
158     .o_SPI_CS_n(w_DAC_CS)
159     );
160
161 always @(posedge CLOCK_50)
162 begin
163   r_Master_TX_DV <= w_Master_TX_Ready;
164 end
165
166 always @(posedge CLOCK_50)
167 begin
168
169   if (w_Master_TX_Ready)
170   begin
171     case (r_ByteContador)
172     3'b000:
173     begin
174       r_QueueByte    <= w_Command_corrente;
175       //r_Master_TX_DV <= 1'b1;
176     end
177     3'b001:
178     begin
179       r_QueueByte    <= w_word_corrente[15:8];
180       //r_Master_TX_DV <= 1'b1;
```

```
181       end
182       3'b010:
183       begin
184         r_QueueByte    <= w_word_corrente[7:0];
185         //r_Master_TX_DV <= 1'b1;
186       end
187       3'b011:
188       begin
189         r_QueueByte    <= w_Command_tensao;
190         //r_Master_TX_DV <= 1'b1;
191       end
192       3'b100:
193       begin
194         r_QueueByte    <= w_word_tensao[15:8];
195         //r_Master_TX_DV <= 1'b1;
196       end
197       3'b101:
198       begin
199         r_QueueByte    <= w_word_tensao[7:0];
200         //r_Master_TX_DV <= 1'b1;
201       end
202       endcase
203       r_ByteContador <= r_ByteContador + 1;
204
205       if (r_ByteContador == 3'b110)
206         r_ByteContador <= 0;
207     end
208   end
209   endmodule
```