Projeto de Graduação

DEPARTAMENTO
DE ENGENHARIA
ELÉTRICA

18 de dezembro de 2020

# Desenvolvimento de solução embarcada baseada em FPGA do tipo Zynq para sistema de ultrassom por transdutores EMAT

Lucas Grativol Ribeiro

PUC RIO

Projeto de Graduação

# Desenvolvimento de solução embarcada baseada em FPGA do tipo Zynq para sistema de ultrassom por transdutores EMAT

**Aluno(s):** **Lucas Grativol Ribeiro**

**Orientador(es):** **Alan Conci Kubrusly**

**Co-orientador(es):** **Felipe Calliari**

Trabalho apresentado com requisito parcial à conclusão do curso de Engenharia Elétrica na Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, Brasil.

## Agradecimentos

À minha família que sempre me apoiou e não mediu esforços para possibilitar que eu trilhasse meu caminho na graduação.

Aos meus orientadores Alan e Felipe e ao Miguel Freitas do CPTI da PUC-Rio, que me acompanharam na elaboração dessa monografia, tendo sempre muita paciência e vontade em me orientar, corrigir meus erros e me indicar os caminhos a serem seguidos.

Agradeço em especial, aos técnicos do departamento de elétrica da PUC-Rio, Carlos Afonso Pacheco, Evandro Costa dos Reis e Manuel Ramos Martins e aos professores do departamento de elétrica da PUC-Rio, Karla Tereza Figueiredo Leite e Mauro Schwanke da Silva, por todos os conselhos, aprendizados e parceria, que tornaram a universidade um ambiente incrível na minha vida.

Por fim, a todos os amigos Ailton de Barros, Arthur Portugal, Bruno Willi, Bráulio Ferreira, Bruno Yago Damasceno, Christopher Feitosa, Felipe Valente, Francisco Renan Lopes, Frederico Gutierrez, Guilherme Gusmão, Henrique Saraiva, Lillyane Valle, Lorena Satler, Lucas Gribel, Olivia Maximiliano, Paula Cristina Gomes, Pedro Villarino, Rachel Marzano, Rafaela Costa Ramon Felipe Nascimento, Reny Barrozo e Thiago Portocarrero que tornaram a graduação uma experiência fantástica e me ajudaram a chegar ao seu fim.

## Resumo

O projeto tem como objetivo desenvolver uma solução embarcada para instrumentação de sistema ultrassônica por transdutores EMAT. O sistema utiliza lógica programável (FPGA) e do processador embarcado (ARM) disponíveis na Zynq. A forma de onda de excitação do transdutor é de grande importância, portanto foi desenvolvida uma unidade capaz de sintetizar formas de onda arbitrárias. As formas de onda podem ser alteradas durante operação, isso permite o, posterior, estudo de seus efeitos na propagação. Alguns dos processos que foram abordados são: a parametrização, o controle e a geração de uma forma de excitação, sua conversão digital-analógica (DAC), transmissão, recuperação do sinal analógico adquirido pelo transdutor e sua conversão analógica-digital (ADC).

**Palavras-chave: FPGA, EMAT, Zynq, CORDIC**

DEPARTAMENTO
DE ENGENHARIA
ELÉTRICA

# Development of Zynq-type FPGA-based embedded solution for EMAT transducer ultrasound system

## Abstract

The project aims to develop an embedded solution for ultrasonic system instrumentation by EMAT transducers. The system uses of the programmable logic part (FPGA) and the processor onboard (ARM) available at Zynq. The excitation waveform of the transducer is of great importance, Therefore a unit capable of synthesizing arbitrary waveforms were developed. The waveforms may be altered during operation, this allows the subsequent study of its effects on propagation. Some of the processes that were tackled are: parameterization, the control and the generation of a excitation waveform, its digital-to-analogue conversion (DAC), transmission, recovery of the analogue signal acquired by the transducer and its analog-digital conversion (ADC).

**Keywords: FPGA, EMAT, Zynq, CORDIC**

## Sumário

**Lista de Figuras** v

**1 Introdução** 1
- a Motivação . . . 1
- b Objetivos . . . 1

**2 Revisão Teórica** 2
- a Plataforma de Desenvolvimento . . . 2
  - 1 FPGA . . . 2
  - 2 VHDL . . . 3
  - 3 ZYNQ . . . 3
  - 4 Protocolo AXI . . . 4
- b Síntese de sinais discretos em FPGA . . . 6
  - 1 Sinais digitais e sistemas digitais . . . 6
  - 2 DAC e Filtro Passa-Baixas . . . 7
- c Tipos de Sinais produzidos . . . 8
  - 1 Sinal arbitrário . . . 8
  - 2 Sinal Quadrado . . . 10
  - 3 Senos e Cossenos . . . 12

**3 Desenvolvimento** 15
- a Arquitetura da Solução . . . 15
  - 1 Visão Geral . . . 15
  - 2 Banco de Registros . . . 16
  - 3 FSM de Controle . . . 17
  - 4 Gerador de Sinal . . . 19
  - 5 Bloco de Médias . . . 23
  - 6 DAC e ADC . . . 24
- b Testes e Resultados . . . 24
  - 1 Estrutura dos testes e resultado parciais para o DDS CORDIC . . . 24
  - 2 Simulação da arquitetura completa . . . 25
  - 3 Resultados Implementação em FPGA . . . 28

**4 Trabalhos Futuros** 30

**5 Conclusão** 31

**6 Referências** 32

**Apêndice A Exemplo VHDL** 34

**Apêndice B Mapeamento dos registradores** 35

## Lista de Figuras

1 Placa Zybo Z7-20 . . . 2
2 Arquitetura ZYNQ embarcada na Zybo Z7-20 . . . 4
3 Canais de uma interface AXI4-Memory Mapped [1] . . . 5
4 Interface AXI-ST . . . 5
5 Exemplo conversão analógica-digital . . . 7
6 Exemplo conversão digital-analógica e filtragem da saída . . . 8
7 Exemplo tabela de pontos . . . 9
8 Exemplo sinal quadrado e os tempos de cada degrau . . . 10
9 Exemplo máquina de estados finitos para gerar um sinal quadrado . . . 11
10 Incremento de fase entre duas amostras . . . 14
11 Arquitetura da solução, diagram de blocos . . . 15
12 Entidade Banco de Registros . . . 16
13 Zonas de tempos . . . 17
14 FSM de Controle . . . 18
15 Bloco Gerador de Sinal . . . 19
16 Bloco Tabela . . . 20
17 Os tipos de sinal quadrado gerados . . . 20
18 Bloco *pulser* . . . 21
19 FSM do *pulser* . . . 21
20 Exemplo da transição de estados, quando o primeiro e o terceiro têm tempo zero. . . . 22
21 Arquitetura do DDS CORDIC . . . 23
22 Interfaces do bloco das médias . . . 23
23 Comparação máximo erro médio absoluto para os casos 8, 10 e 12 bits, no DDS CORDIC. . . 25
24 Resposta em frequência de um seno 500 kHz com 10 períodos e 10 bits de precisão . . . 30

# 1 Introdução

A primeira seção deste trabalho introduz a motivação para o desenvolvimento da solução proposta e estabelece os objetivos para a realização do trabalho, que são recapitulados e mencionados no texto deste trabalho.

## a Motivação

O CPTI (Centro de Pesquisa em Tecnologia de Inspeção) da PUC-Rio desenvolve soluções para inspeção de segmentos da malha dutoviária, construíndo PIGs (*Pipeline Inspection Gauge*) adequados as mais diversas situações em parceira com empresas de renome no mercado. Neste contexto é de suma importância a capacidade de processar os dados de inspeção em tempo real, sendo capaz de interagir com o sensorialmente do PIG, implementar algoritmos de processamento de sinal e manejar o sistemas de comunicação para recuperação de dados.

Para atender a esses requisitos é necessário um circuito de alto desempenho que se adeque as diferentes necessidades. Uma FPGA, um circuito digital reprogramável a partir de estruturas básicas de circuitos digitais, é uma das soluções empregadas, pelo seu alto desempenho e flexibilidade.

Focando no uso de transdutores do tipo EMAT, o sinal de excitação influencia a qualidade e o tipo de dados recuperados, possibilitando, a detecção de falhas especificas em tubulações. As referências [2] e [3] apresentam exemplos de trabalhos aplicados no estudo da influência da detecção de falhas usando transdutores do tipo EMAT.

Assim, o trabalho explora a construção de uma solução em FPGA para a produção de sinais genéricos que podem ser configurados e operados para a excitação de transdutores, possibilitando o estudo e desenvolvimento futuro de ferramentas de inspeção baseadas no uso de sinais genéricos.

## b Objetivos

Dada a motivação deste trabalho, o principal objetivo estabelecido é a construção de uma solução capaz de gerar sinais genéricos, fornecendo ao utilizador a capacidade de configurar o sinal e recuperar os dados em tempo real.

Além do estabelecido, a plataforma de trabalho escolhida foi uma FPGA do tipo ZYNQ, possibilitando à solução se aproveitar do sistema embarcado (ARM) presente na placa para a construção de um *hardware* dedicado ao processamento em tempo real, cabendo ao *software* a estruturação de um controle de mais alto nível. Com esse sistema é possível aproveitar as facilidades oferecidas pelos dois lados, *software* e *hardware*, na solução final.

O projeto passa pelas etapas de estudo da plataforma (FPGA), da geração de sinais digitais e as interfaces de comunicação entre o processador ARM e a FPGA. Com os conceitos fundamentados, o desenvolvimento introduz uma entidade de configuração do sinal a ser gerado, os blocos responsáveis pela geração dos sinais, se baseando nos métodos estudados, a geração de sinais analógicos a partir de sinais digitais, utilizando conversores DAC e ADC, e estabelece uma entidade para a recuperação de dados.

# 2 Revisão Teórica

Esta seção apresenta os conceitos e conhecimentos que serviram de base para o projeto.

Nessa essa seção são discutidos conceitos e elaboradas ideias, que assumem o entendimento do leitor sobre o funcionamento de eletrônica digital e, em especial, da diferença entre lógica combinatória e sequencial. Sendo de grande importância para melhor compreensão do texto, conceitos como o impacto de uma frequência de funcionamento, um *clock*, em um circuito digital e a noção de ciclos de *clock*, [4].

## a Plataforma de Desenvolvimento

A plataforma de desenvolvimento para esse trabalho é a placa Zybo Z7-20 [5], produzida pela Digilent ®, baseada na arquitetura ZYNQ, desenvolvida pela fabricante Xilinx®. A placa integra um processador ARM e uma FPGA (Field Programmable Gate Array) em único sistema.

A seguir são discutidos os detalhes sobre a placa em mais detalhes.

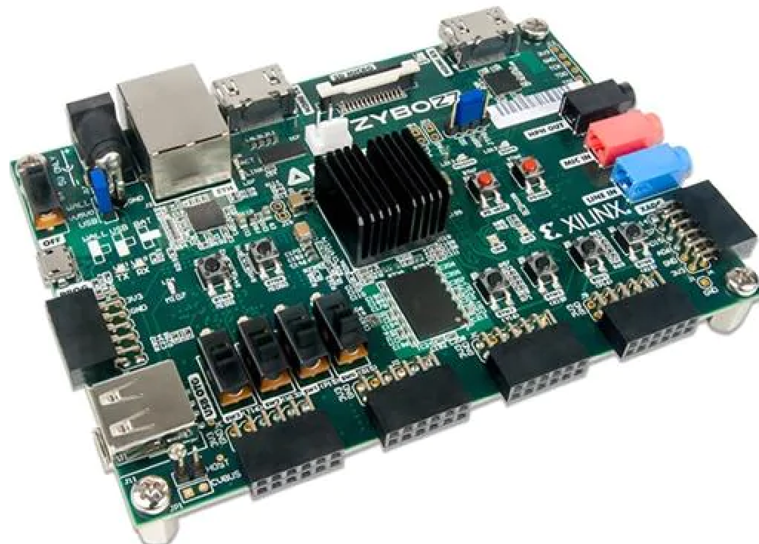



Figura 1: Placa Zybo Z7-20

### 1 FPGA

Uma FPGA é, em termos simples, um circuito eletrônico que pode ser reprogramado para realizar operações lógicas [6]. Em busca de um maior entendimento do seu funcionamento, o seu nome pode ser quebrado e os termos explorados individualmente, começando do último termo até o primeiro.

- O termo *"Gate Array"* se refere ao fato que a circuito eletrônico da FPGA é composta por grupos de blocos de circuitos lógicos, com portas lógicas e/ou elementos de memória, conectados entre si;
- O grande diferencial que leva ao termo *"Programmable"*, se dá pelo fato que os blocos podem ser configurados para implementar diferentes funções lógicas e as conexões entre blocos também podem ser configuradas;
- O primeiro termo *"Field"* marca a característica que uma FPGA pode ser reconfigurada a qualquer momento. Inicialmente, esse termo se referia ao fato de que as placas podiam ser configuradas cada vez que eram energizadas, mas, nos tempos atuais, é possível reconfigurar uma FPGA de forma parcial [7], alterando partes do circuito, enquanto outra parte continua a funcionar, não sendo necessário reiniciá-la;

O projeto de um circuito integrado típico conta com um ciclo de desenvolvimento com um alta complexidade e um alto custo ligado a prototipagem e testes do chip em questão. O resultado é uma eletrônica dedicada, de alto desempenho e baixo consumo, mas com pouca flexibilidade para ter seus erros corrigidos e/ou aumentar suas capacidades. A FPGA se insere nesse contexto como uma solução intermediária, podendo atuar como uma fase anterior a produção definitiva do chip ou até como sua substituta. A capacidade de reconfiguração de uma FPGA permite que a etapa de desenvolvimento e debug tenha seu tempo e seus

custos envolvidos reduzidos, tendo os testes em ambiente de simulação e testes físicos mais acessíveis, diretamente sobre a própria placa. Uma consequência lógica, é que uma FPGA não consegue, em muitos casos, alcançar o mesmo desempenho [8] e baixo consumo de um ASIC (*Application Specific Integrated Circuits*), mas isso é uma compensação da sua flexibilidade. A escolha entre ASIC e FPGA reside na escolha entre uma solução de baixo-custo, mas sem flexibilidade, em que qualquer erro no produto final é de difícil solução, o ASIC. E a FPGA, com seu custo unitário maior que o de um ASIC, mas com capacidade de remediar erros de *design* com facilidade.

Configurar uma FPGA, ou como comumente dito programar, é feito com uma linguagem de descrição de hardware, do inglês HDL (*Hardware Description Language*), que descreve a estrutura e o comportamento de um circuito eletrônico, nesse caso a FPGA, e um software transformar a linguagem em uma configuração [6]. O ambiente de trabalho com FPGAs é fortemente atrelado ao fabricante da placa, que detém todas as ferramentas necessárias para fazer a tradução de HDL para um circuito digital tradicional, a síntese, o testes da estrutura descrita, a simulação, a transformação do circuito digital em elementos presentes na FPGA, a implementação, e a programação da placa. Nesse trabalho o ambiente utilizado foi o *Vivado Design Suite* versão 2017.4, fornecido pela fabricante da placa, a Xilinx®, de forma gratuita. No mercado, por questões históricas e comerciais, existem diversas linguagens HDL que podem ser usadas, mas todas dependem da aceitação do ambiente de desenvolvimento usado. Entre todas as opções, nesse trabalho optou-se pelo uso do VHDL (*VHSIC Hardware Description Language*, em que VHSIC significa *Very High Speed Integrated Circuits*). A escolha foi feita pela familiaridade do autor com a linguagem.

## 2 VHDL

A principal diferença que separa uma linguagem de programação tradicional, como C ou Python, e uma linguagem de descrição de hardware, HDL, é a concorrência. Quando um código escrito em C é executado, o código de máquina gerado a partir dele será lido linha por linha pelo processador, sendo uma execução sequencial. Uma HDL descreve um hardware, portanto, uma vez descrito um adicionador ou uma porta lógica AND, por exemplo, esses dois elementos existirão fisicamente e sua "execução" acontecerá em paralelo, afinal são componentes reais. Outros conceitos, como variáveis, controles de execução e operações lógicas e aritméticas ainda existem para HDLs, mas com consequências diferentes, pois estão descrevendo algo que deverá ser implementado em forma de hardware.

No contexto geral, HDLs são usadas na etapa de concepção do circuito digital, servindo para descrever os diferentes componentes físicos, suas interações e procedimentos de testes capazes de verificar o funcionamento de tais circuitos. A descrição de um circuito, uma vez validada, é sintetizada para o seu equivalente em componentes digitais comuns, tais como portas lógicas, multiplexadores, registradores e blocos de memória. Esses componentes podem ser usados para gerar um compilado de descrição em forma de uma *netlist*, que diferentes ferramentas, não só uma FPGA, podem usar para criar um circuito eletrônico, usando elementos de eletrônica analógica como transistores.

O VHDL, usado neste trabalho, tem sua origem em 1983 no departamento de defesa do Estados Unidos (DARPA) [9] para, inicialmente, documentar os ASICs presentes em equipamentos do departamento. A partir das documentações foram desenvolvidas ferramentas que pudessem simulá-las e depois ferramentas de síntese, capazes de transformar as documentações em verdadeiros circuitos digitais.

Hoje em dia, o VHDL é mantido pelo IEEE (*Institute of Electrical and Electronics Engineers*, ou em português Instituto de Engenheiros Eletricistas e Eletrônicos), responsável por padronizar e revisar a linguagem. O VHDL utilizado aqui é o do padrão IEEE 1076-1993 [10], pois apesar de possuir novas versões, como a IEEE 1076-2008 [11] e a IEEE 1076-2019 [12], tais versões não são completamente suportadas pelas ferramentas de desenvolvimento em FPGA na indústria atualmente.

Com o intuito de familiarizar o leitor com a linguagem, são apresentados exemplos no apêndice A, da descrição de um circuito que implementa uma porta AND e um registrador, com comentários sobre a linguagem.

VHDL é apenas uma opção entre diversas outras HDLs, nos dias de hoje várias entidades comerciais ou não, tentam desenvolver suas próprias linguagens, por suas próprias razões e aplicações em mente, tais como MyHDL [13], Chisel [14] e SystemC [15], mas as mais famosas e adotadas pela indústria são o Verilog, o VHDL e o SystemVerilog.

## 3 ZYNQ

A placa Zybo Z7-20 [5] usada neste trabalho foi construída pela Digilent com o SoC (*System on Chip*) da arquitetura ZYNQ desenvolvido pela Xilinx®. Em termos práticos, a placa em si conecta o SoC ZYNQ no seu

meio a todos os periféricos e interfaces físicas as quais o SoC tem compatibilidade, formando uma placa de desenvolvimento que permite ao seu usuário um meio fácil de utilizar os pinos do chip localizado no meio da placa. O SoC ZYNQ não é referido como sendo uma FPGA, pois a arquitetura ZYNQ usada consiste da integração de um processador ARM e uma FPGA em único chip. No caso da Zybo Z7-20, o sistema integra um processador ARM Dual-Core Cortex A9, localizado na parte conhecida como PS (*Processing System*), e uma FPGA do tipo Artix-7, localizada na parte conhecida como PL (*Programming Logic*), possuindo o código do fabricante XC7Z020-1CG400C. O código do fabricante é uma forma única de identificar uma FPGA, aqui o SoC, servindo como a referência para a peça, tal como Zybo Z7-20 representa a placa.

A figura 2 destaca uma visão de blocos simplificada deste sistema. Mais detalhes sobre as conexões entre PS e PL, as portas AXI (*Advanced eXtensible Interface*), são dados no tópico seguinte.

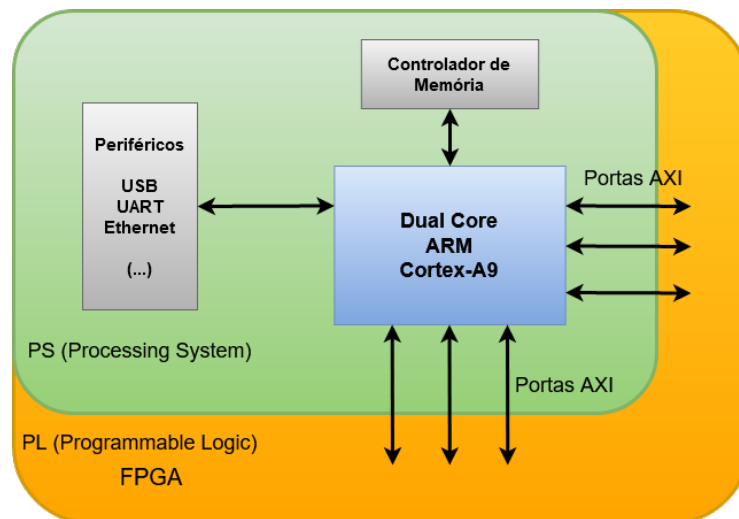



Figura 2: Arquitetura ZYNQ embarcada na Zybo Z7-20

Diferente da operabilidade de um FPGA tradicional que para uma integração de mais alto nível, com uma CPU (Central Processing Unit), por exemplo, requer a utilização de uma interface de comunicação suplementar, uma ZYNQ possui essa integração já presente no hardware. Essa integração possibilita a construção de um sistema embarcado que se aproveite do processador contido na PS e utilize uma solução dedicada construída na PL para realizar uma ou mais tarefas específicas com alto desempenho. Um tópico importante a se discutir nesse ponto é ao que o termo desempenho faz alusão. Uma FPGA configurada para executar uma certa operação, tem apenas aquela função e pode executar ela em alguns ciclos de *clock*, da ordem de centenas de megahertz. Uma CPU mesmo possuindo frequências de funcionamento muito superiores, na casa de gigahertz, está atrelada a outros dispositivos que por si funcionam em seu próprio tempo, assim, mesmo com mais poder de processamento, uma tarefa executada em uma CPU seria menos eficiente do que em uma FPGA.

Existem duas principais formas de se usar uma ZYNQ. A primeira se utiliza de um estado de baixo nível, onde o processador na PS executará uma aplicação chamada "bare-metal" [16], em que somente o código necessário para executar essa aplicação é carregada na memória da placa e os núcleos do processador ficam em perpetuidade executando somente essas linhas de código. A segunda, e a visada para esse projeto, é carregar um sistema operacional adequado ao contexto embarcado desse SoC e ter acesso a todos as funcionalidades de um sistema de alto nível. O sistema escolhido é o Petalinux 2017.4 [17], mantido pela própria Xilinx®.

## 4 Protocolo AXI

Como destacado na figura 2, PL e PS são conectados por meio das portas AXI (*Advanced eXtensible Interface*). O AXI é um protocolo que faz parte do AMBA (*Advanced Microcontroller Bus Architecture*), um grupo de barramentos criado e utilizado pela ARM, que a Xilinx® adota como interface padrão de suas soluções IP (*Intellectual Property*). O protocolo AXI funciona no esquema "*master-slave*" [18], baseando-se em um acordo entre as duas entidades, um *handshake*, do tipo *ready*/*valid*. O *handshake* funciona com uma das entidades sinalizando estar pronta/apta a processar uma informação no ciclo de *clock* atual, *ready*, e outra dizendo que a informação transmitida é válida no ciclo atual, *valid*. Quando ambos os casos acontecem em um mesmo ciclo de *clock*, uma transação é realizada. Existem dois tipos de interface

dentro do protocolo. O primeiro tipo *Memory-Mapped* utiliza a ideia de escritura e leitura em uma memória, possuindo endereçamento. O outro tipo, chamado de Stream, suprime a ideia de endereçamento para transmitir as informações de forma contínua , *streaming* de dados.

O tipo *Memory-Mapped* se baseia na utilização de 5 canais dentro da interface, cada um possuindo individualmente o paradigma *master-slave* e o *handshake ready/valid*. Os cinco canais apresentados na figura 3, são explicados em sequência.

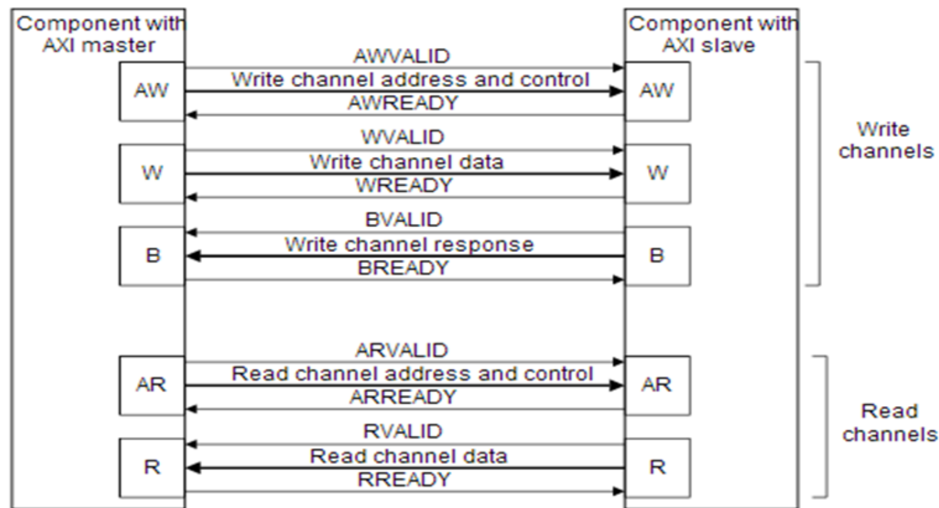



Figura 3: Canais de uma interface AXI4-Memory Mapped [1]

- *Write Address*: Utilizado para fornecer o endereço de escritura na memória, podendo ser realizado antecipadamente ou em conjunto com o canal de escrita de dados;
- *Write Data* : O dado propriamente dito que deve ser escrito no endereço fornecido no canal *Write Address*;
- *Write Response* : Usado para sinalizar que o último processo de escrita foi bem sucedido;
- *Read Address* : Utilizado para fornecer o endereço de leitura na memória, podendo ser realizado antecipadamente ou em conjunto com o canal de leitura de dados;
- *Read Data* : Usado para ler uma informação no endereço fornecido pelo canal *Read Address*.

Em conjunto com as especificações já passadas, a interface do tipo AXI-MM (*Memory Mapped*), ainda se divide mais uma vez, em uma interface chamada AXI-Lite, que é uma versão mais simples do protocolo apresentado e , normalmente, usada em microprocessadores.

Por fim, a última divisão do protocolo, é a interface AXI-ST (Stream), apresentada na figura 4, que muito mais simples do que o AXI-MM, apresenta, além do *handshake* e da informação transmitida, um sinal *last*, indicando o final de uma transação.

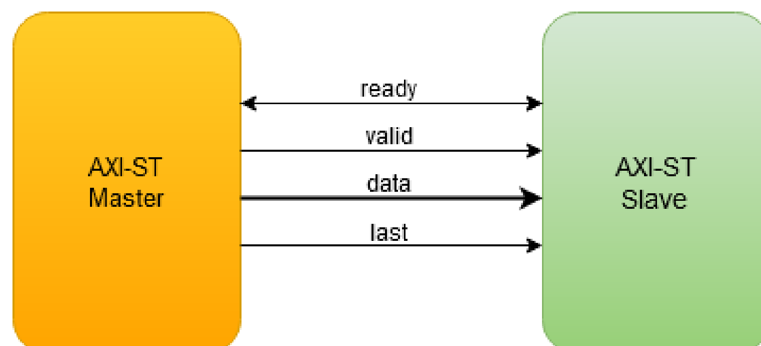



Figura 4: Interface AXI-ST

O trabalho se limita a introduzir o protocolo de forma geral, para maiores informações a documentação aberta do protocolo deve ser consultada, [19] e [20].

## b Síntese de sinais discretos em FPGA

Um dos objetivos desse trabalho é possuir uma plataforma capaz de gerar sinais analógicos em tempo real, controlados por parâmetros fornecidos à solução, que podem ser usados para excitação de transdutores. Nesse contexto, é de alto interesse introduzir a noção de um sinal digital, sua relação com um sistema digital e sua transformação em um sinal analógico.

### 1 Sinais digitais e sistemas digitais

Sinais do mundo real são analógicos, contínuos no tempo e contínuos em amplitude, contudo a eletrônica digital, sistemas digitais , necessitam de uma representam mais adequada. Para um mundo digital, é preciso transformar as duas quantidades contínuas em discretas, obtendo sinais digitais. O tempo de um sinal analógico é discretizado por meio de um processo conhecido como amostragem [21]. De forma simples e breve, um processo de amostragem captura valores de um sinal analógico em intervalos de tempo iguais, criando um sinal discreto $y[k] = y(kT_s)$, onde $k \in \mathbb{Z}$, e $T_s$ representa o período de amostragem, em que $f_s = \frac{1}{T_s}$ é a chamada frequência de amostragem. Sem maiores desenvolvimentos, para que o sinal amostrado possa representar propriamente o sinal analógico, e eventualmente reconstruí-lo, é necessário que a frequência de amostragem siga o Teorema de Nyquist [21], que estabelece que a frequência de amostragem deve ser, pelo menos, duas vezes maior que a componente de frequência mais alta do sinal amostrado. Os dois processos finais são baseados na ideia de usar números binários para limitar a precisão da quantização da amplitude da amostra em níveis discretos e como a sequência binária será usada para descrever o número, codificação. Uma palavra de n bits possui $2^n$ valores possíveis, assim, esses $2^n$ valores devem ser distribuídos para representar os diferentes valores assumidos pelas amostras. Existem diversas técnicas de quantização, tais como a quantização uniforme ou técnicas de quantização não-lineares, e diferentes formas de codificação, por exemplo, a lei $A$ e a lei $\mu$, mais detalhes são encontrados em [22]. Cada uma com suas vantagens e desvantagens para representar sinais de grandes ou pequenas excursões robustez ao ruído, dentre outros objetivos.

Neste trabalho é considerado os casos mais simples, dada uma precisão de $n$ bits, as $2^n$ palavras são igualmente distribuídas nas amplitudes possíveis. No caso da codificação, foram usados os valores quantizados e traduzidos para sistema binário simplesmente, sem aplicar nenhum tipo de codificação em especial. A figura 5 mostra essas etapas do processo com um sinal $y(t)$, sendo amostrado, em intervalos $T_s$, versus sua versão somente quantizadas, $y_Q(t)$, e como os dois resultados se juntam para formar um sinal digital, com uma quantização/codificação de 3 bits, $y_Q[k]$.

O presente tópico serviu para exemplificar o que é um sinal digital, sem recorrer a sua definição formal e matemática. O ponto principal é o lugar de tal sinal dentro de uma FPGA. Como já discutido, uma FPGA é formada por uma série de componentes lógicos e, portanto, é natural pensarmos em números binários, sendo armazenados dentro da FPGA, em registradores (*flip-flops*) ou em blocos de memória. Isso não é diferente de nenhum tipo de computador ou dispositivo eletrônico com um mínimo de processamento envolvido.

A diferença vem da forma como esses números binários são tratados. A ideia de tipagem proporciona um nível de interpretação mais alto para a informação binária armazenada, exemplos são tipos como *integer* (int), *float*, *char* e *string*. Essa é a forma tradicional de se tratar números binários, a partir de uma linguagem de programação que define e utiliza tipos. Em FPGA toda representação é binária, não existe uma noção de tipagem tal como a tradicional. Assim, os sinais digitalizados são puramente bits armazenados e que cabe ao programador manipulá-los de forma adequada. O estabelecimento dessa diferença serve para ressaltar um compromisso de desempenho e recursos utilizados em FPGAs. Esse compromisso não é exclusivo de FPGAs e aparece em sistemas embarcados em geral, mas aqui o foco é mantido em FPGAs. A quantidade de elementos digitais dentro de uma FPGA é limitado, a FPGA usada nesse trabalho, por exemplo, é capaz de armazenar em memória 630 kB, muito menos do que qualquer computador moderno. É nesse ponto que o compromisso de desempenho e utilização de recursos aparece em uma FPGA, falando do armazenamento de sinais digitais, existe um limite entre o tamanho da palavra (sua quantidade de bits), que será armazenado ou processado, com os recursos e capacidades da FPGA em questão. Esse é um quesito importante usado durante toda a elaboração de um design em FPGA.

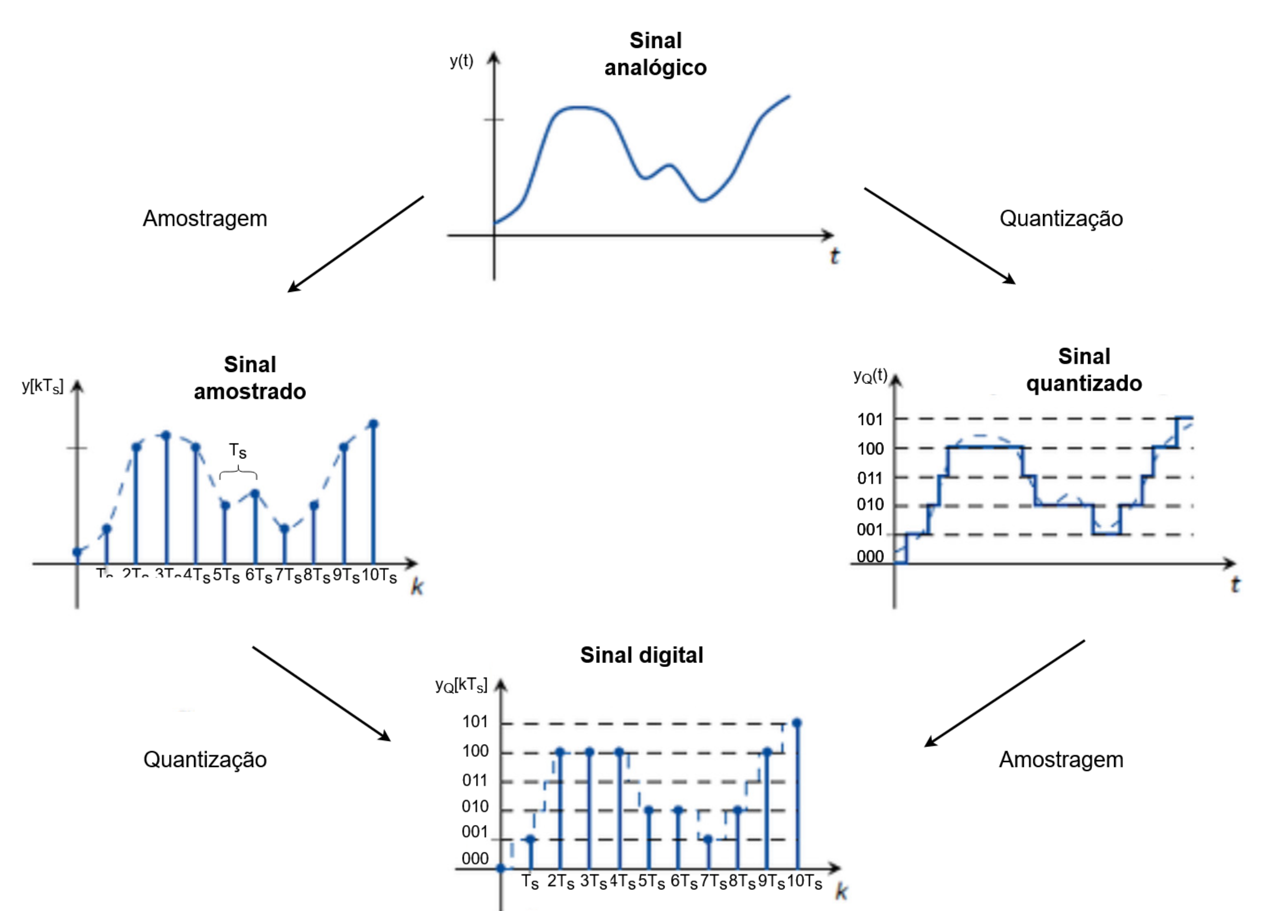



Figura 5: Exemplo conversão analógica-digital

## 2 DAC e Filtro Passa-Baixas

Recapitulando um dos objetivos desse trabalho, quer-se uma solução capaz de gerar um sinal de excitação para o transdutor e, em seguida, ser capaz de recuperar tal sinal para tratamento interno na FPGA. Neste contexto, existe a transição de sinal digital para analógico e vice-versa, tais feitos são alcançados usando um DAC (*Digital to Analog Converter*) [23] e seu recíproco ADC (*Analog to Digital Converter*) [23], respectivamente.

A primeira conversão, digital para analógica, ocorre com a ajuda de um DAC. Números binários utilizam a posição de seus dígitos para definir a presença ou a falta de uma potência de dois, tal como no nosso sistema decimal utiliza números de 0-9 e suas posições com potências de 10. Com essa ideia, um DAC utiliza a relevância de cada bit para somar níveis de tensão, que resultam em um sinal analógico, representando a palavra em bits fornecida ao conversor. Um dos parâmetros importantes do DAC relacionado com a quantização/codificação é a resolução do conversor, expressa em termos da quantidade de bits do conversor . É essa resolução que vai definir o tamanho das palavras em bits que podem ser convertidas, em que um maior número de bits representa uma maior precisão. Outro fator crucial, é a frequência de amostragem do conversor, definindo a maior frequência do sinal que pode ser utilizado, como expresso pelo Teorema de Nyquist. A etapa seguinte é aplicar um filtro passa-baixas sobre o sinal gerado pelo DAC. Esse filtro tem o objetivo de suavizar a saída do DAC, como exemplificado na figura 6, a saída do DAC apresenta níveis constantes, por causa da conversão de cada palavra no DAC a cada $T_s$ (intervalo de amostragem). O conjunto desses níveis, além de não representarem fielmente o sinal analógico correspondente, introduzem componentes de frequências mais altas que o do sinal, que são filtradas, para obter-se um sinal suavizado e reconstruído, no sentido de ser fiel ao sinal analógico correspondente, na saída. Na aplicação desse projeto um amplificador exterior é necessário para conformar a onda nas especificações necessárias para excitar o transdutor.

A etapa de conversão analógica-digital, necessária para recuperar o sinal e enviar de volta para a FPGA, é feita com um ADC. Muito parecido com o DAC, o ADC tem como parâmetros fundamentais a sua resolução

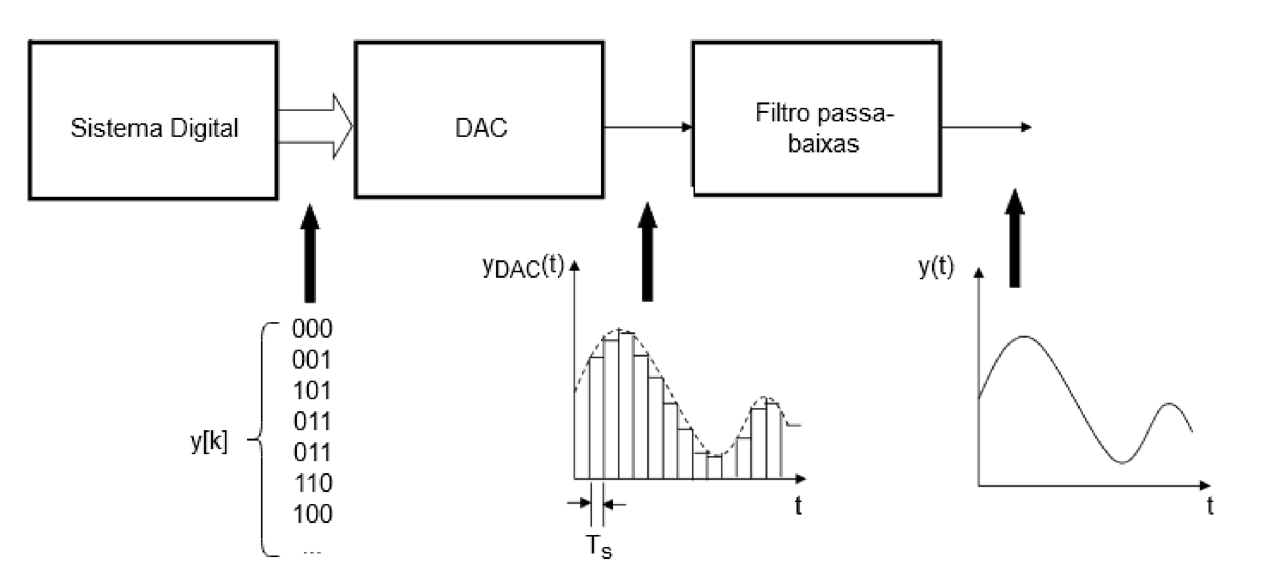



Figura 6: Exemplo conversão digital-analógica e filtragem da saída

em bits e a frequência de amostragem. O funcionamento básico de um ADC ideal, é comparável com um comutador que fecha em intervalos de tempo $T_s$, carregando um capacitor com o valor de tensão correspondente de cada amostra, e, em seguida, cada amostra é então comparada com valores predeterminados de tensão, para obter a palavra em bits que melhor define aquele nível de tensão. Aqui, novamente, é acrescentado um filtro passa-baixas, nesse caso antes do conversor, com a intenção de eliminar qualquer componente de alta frequência, fora do esperado para o sinal a ser convertido, devido a qualquer fonte de ruído.

É inegável a importância de tais componentes afim de fornecer sinais analógicos para computadores digitais. Vários tipos de aplicação utilizam tais conversores, tais como áudio, telecomunicações, sistemas de automação, sensores etc. A lista é tão grande quanto existem aplicações que necessitam de qualquer tipo processamento digital.

A próxima seção utiliza a noção de sinais digitais para descrever os tipos de sinais usados nesse trabalho e, nas próximas seções, quando a arquitetura é discutida, volta-se a discutir sobre esses conversores.

## c Tipos de Sinais produzidos

Esta seção inicia a discussão do lado mais técnico, falando de aspectos ligados a implementação da solução. Os tipos de sinal considerados nesse trabalho são enunciados em conjunto das técnicas usadas para produzi-los. Primeiramente, é apresentada uma solução genérica para síntese de sinais digitais, em seguida, são apresentados os métodos considerados para os casos particulares de um sinal quadrado e uma senoide.

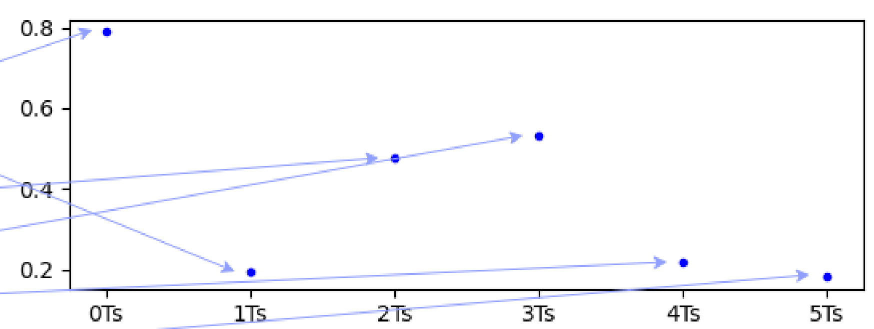
### 1 Sinal arbitrário

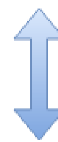
Revisitando a ideia de sinais digitais como amostras de um sinal analógico, é possível associar essa ideia a uma estrutura de dados comumente usada em programação, a de um vetor de dados. Esse vetor ordenado possui em cada um de seus elementos amostras que compõe o sinal digital. Agora, como citado na subseção 2.b.2, se cada amostra do vetor fosse enviada a um DAC, seguindo o período de amostragem, teríamos a produção de um sinal analógico a partir de uma estrutura que retém cada amostra do sinal a ser sintetizado.
Esse vetor ordenado com as amostras será referido aqui como sendo uma tabela, pois esse termo se aproxima mais da ideia do que será implementado na FPGA, de uma memória onde cada endereço possui uma amostra.

A figura 7 mostra uma tabela com pontos que constituem um sinal qualquer.

Como o método usado consiste apenas em ter uma tabela, memória para a FPGA, com o sinal a ser gerado, o que resta para o método é uma forma de preencher a tabela. As amostras, ou pontos, podem

| Tabela de pontos | |
|---|---|
| Índice | Ponto |
| 0 | 0.0445 |
| 1 | 0.9964 |
| 2 | 0.6644 |
| 3 | 0.3192 |
| 4 | 0.9215 |
| 5 | 0.7684 |



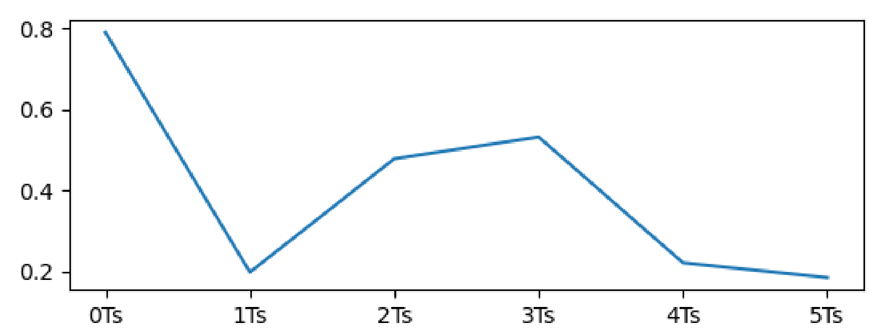



Figura 7: Exemplo tabela de pontos

ser gerados com o auxílio de uma linguagem de programação tradicional, tal como Python, ou usando uma ferramenta que nativamente tem suporte a geração de sinais como o Matlab. Duas formas de preencher a tabela podem ser propostas, a primeira é preencher com amostras fixas e a segunda é preencher a tabela de forma dinâmica.

A primeira forma é de fácil, a tabela é configurada na FPGA e não necessita de interferência externa para operar, mas perde a capacidade dinâmica de geração de um sinal arbitrário. Apenas o sinal já carregado na tabela poderia ser repetido indefinidamente.

A segunda forma resolve o problema do anterior, ao enviar os pontos à FPGA para preencher a tabela, ganhando liberdade para gerar um sinal, entretanto enviar um número binário para uma FPGA não é tão simples. Tal como qualquer sistema, uma FPGA possui interfaces para se comunicar com o exterior, tais como UART, USB e PCIe, mas cada interface requer que a FPGA seja programada para suportar tais protocolos, o que exige uma lógica adicional, e consequentemente o gasto de recursos da FPGA. O ganho em liberdade é contrastado com uma complexidade adicional de comunicação com a FPGA. Outra consequência, é o tempo necessário para realizar a transferência de dados para a tabela, em sistemas com processamento em tempo real, essa solução pode não ser ideal.

O método citado, apesar de uma solução o direta e simples, possui uma grande desvantagem que não permite que ele seja a solução ótima para todos os problemas. A tabela, uma vez sintetizada, ocupará blocos de memória da FPGA. Blocos de memória são elementos físicos dentro uma FPGA, que podem agrupar dados e endereça-los, sendo um recurso relativamente escasso e precioso. Outro fator é que fisicamente os blocos de memória, ou chamados blocos RAM (*Random-Access Memory*), não estão todos concentrados em único lugar e são distribuídos para ocupar todo o espaço físico da FPGA, de forma a proporcionar à todo o design a capacidade de usar esse recurso. Isto significa que conforme mais blocos RAM são necessários, mais complexo se torna o design.
Um exemplo que demonstra esse problema é o de um simples seno. Seja um seno de frequência $100kHz$ amostrado em $100MHz$, tal sinal possuí $\frac{100MHz}{100kHz} = 1000$ pontos e ele é quantizado/codificado em 8 bits. O tamanho total ocupado por 1 período é de cerca de 1 kB. A FPGA usada nesse trabalho possui 630 kB

total [5], dividida em 140 blocos de 36 kb, assim o sinal gasta 1 dos blocos e deixa 3.5 kB do bloco sem uso. Nesse exemplo, um período é facilmente armazenável dentro da memória. A questão é que conforme aumentam os números de pontos em um período e a precisão, número de bits, a ocupação de blocos de memória aumenta com a multiplicação desses fatores.

Para este trabalho ter um processador ARM fisicamente conectado a FPGA facilita a escolha do método da tabela, mesmo levando em conta o gasto de blocos RAM.
Com o objetivo de generalidade e maior liberdade aos futuros utilizadores, foram desenvolvidas entidades capazes de gerar sinais quadrados, senos e cossenos. A forma de gerar cada um desses sinais é descrita a seguir.

## 2 Sinal Quadrado

O sinal quadrado aqui se caracteriza pela presença de três amplitudes diferentes, $-1$ (um negativo), zero e $+1$ ( um positivo), e os diferentes tempos que cada amplitude pode assumir. A figura 8 apresenta o sinal descrito. Cada degrau, sinalizado pelos tempos $t_1$ à $t_5$, possui sua própria amplitude e pode assumir durações arbitrárias.

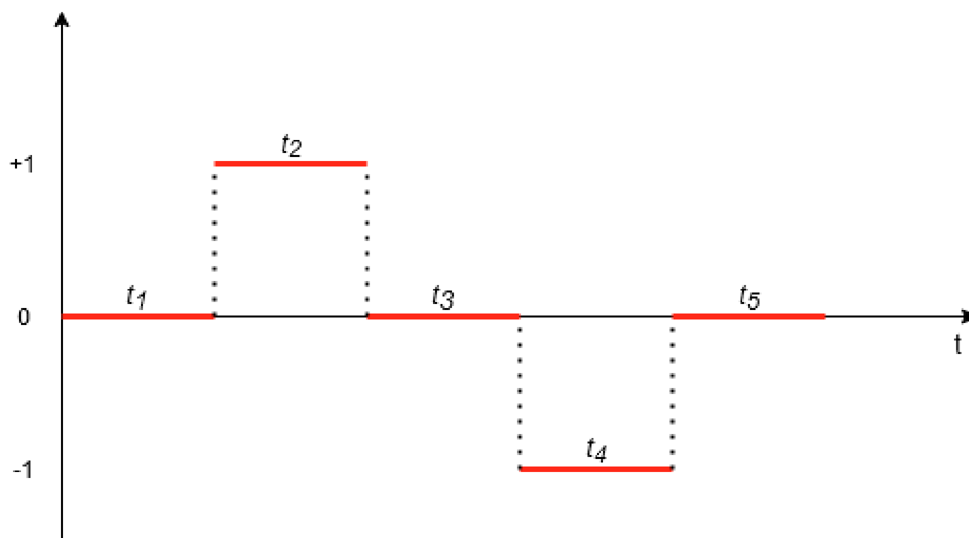


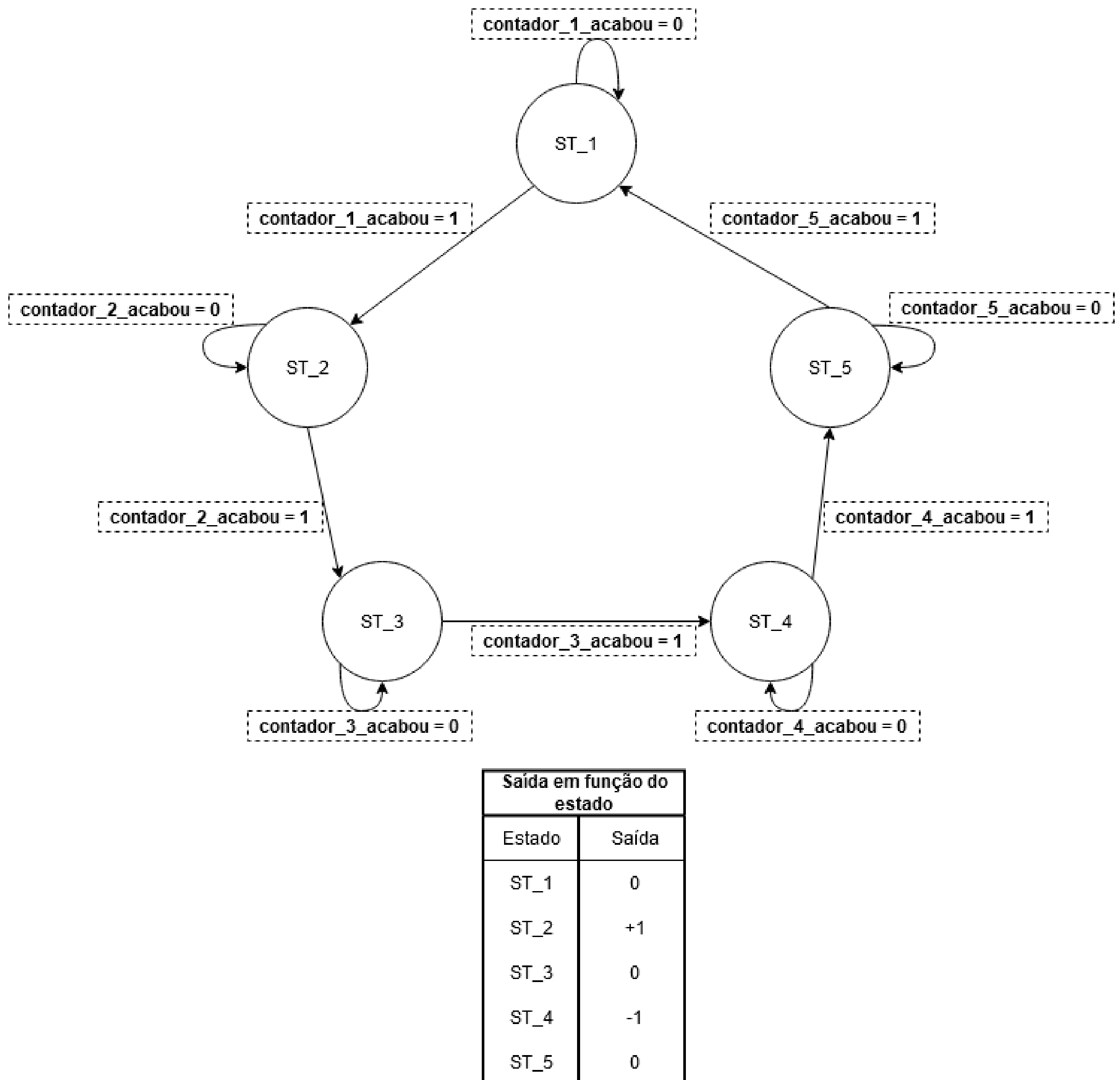

Figura 8: Exemplo sinal quadrado e os tempos de cada degrau

Esse é o sinal mais simples a se produzir, pois ele é natural para um circuito digital. As amplitudes são valores binários fixos, definidos usando-se complemento a 2 [24], e o tempo dos degraus são contadores binários com resolução de tempo igual ao *clock* da FPGA.

Existe uma noção de ordem e de sequência na forma de gerar o sinal descrito pela figura 8. Para cada degrau, o valor de amplitude atribuído a si é mantido constante, até que um contador, o elemento marcando o tempo, diga que o tempo requerido foi alcançado. Cada degrau é sucedido pelo seguinte conforme seus tempos são atingidos e o processo recomeça quantas vezes forem desejadas.

Uma forma prática e comum de se descrever o processo citado, é com uma uma máquina de estados finitos. Uma máquina de estados finitos, ou autômato finito, é um modelo comumente usado em programas de computador ou eletrônica digital para descrever um processo em termo de seu estado atual, os possíveis estados futuros e as transições entre estados. O modelo tem como vantagem representar com facilidade um processo em etapas, com cada etapa (estado) e suas transições a partir de estímulos (entradas) bem definidos. O fato de poder identificar em qual etapa o processo se encontra e/ou seus estados futuros, permite definir o comportamento do sistema em função do estado, simplificando o controle de circuitos digitais. Exemplos simples de autômatos finitos são máquinas de vendas, caixas eletrônicos, fechaduras digitais e elevadores.

A figura 9 mostra uma máquina de estados finitos simples, capaz de gerar o sinal descrito na figura 8. A transição de cada estado é feita quando o contador associado ao estado alcança o seu tempo final, representado pelos sinais ”contador_N_acabou = 1”, indicando que o contador de número “$N$” acabou de contar e ”contador_N_acabou = 0”, caso contrário.




| Saída em função do estado | |
|---|---|
| Estado | Saída |
| ST_1 | 0 |
| ST_2 | +1 |
| ST_3 | 0 |
| ST_4 | -1 |
| ST_5 | 0 |

Figura 9: Exemplo máquina de estados finitos para gerar um sinal quadrado

A forma de ser ler o diagrama de estados da figura 9 é a seguinte :

- Começando de ST_1;
- O sistema permanece no estado ST_N, enquanto o contador_N_acabou = 0, uma vez que ele acabe, o sistema deverá transitar para o estado ST_N $+1$;
- Em cada estado, a sua saída, ou resultado sobre o sistema, é descrito na tabela adjacente. Ex: Para o estado 1 a saída é 0, o estado 2 a saída é $+1$ e assim por diante, formando ao final, o sinal descrito na figura 8.

A máquina de estados finita apresentada é do tipo máquina de Moore [25], na qual a saída de cada estado é determinada pelo estado atual. Outro modelo possível seria o de Mealy [26], que descreve as saídas em função da transição dos estados.

O diagrama de estados citado é apenas para exemplificação, ele está simplificado e não representa o atual diagrama usado neste trabalho. A geração de um sinal quadrado é novamente discutida e os detalhes do atual diagrama utilizado são dados na subseção 3.a.4.

## 3 Senos e Cossenos

Senos e cossenos são sinais de grande utilidade em geral, o que motivou a criação de um algoritmo dedicado a sistemas digitais para sua geração. O algoritmo mais comum utilizado para a geração desses sinais é chamado de CORDIC (COordinate Rotation DIgital Computer), concebido para resolver problemas de navegação em tempo real, sendo creditado a Jack Volder. O trabalho original visava sistemas digitais com poucos recursos, necessitando de processamento em tempo real e alta precisão, sem depender de valores pré-calculados armazenados no interior das calculadoras.
Posteriormente a sua criação, o CORDIC foi estendido em outros trabalhos para se adaptar a funções mais complexas, como funções trigonométricas inversas, funções hiperbólicas, cálculo de recíproco, entre outras. Uma referência clássica ao algoritmo é [27], que faz uma recapitulação do algoritmo original e os algoritmos derivados a partir dele.

Mesmo a referência clássica sendo completa, o trabalho se permitirá introduzir uma dedução que leva ao algoritmo CORDIC original. A discussão de como o algoritmo funciona é importante para os impactos do seu design em uma FPGA.

A ideia é que se desejamos obter o seno ou cosseno para um ângulo $\theta \in [-\frac{\pi}{2}; \frac{\pi}{2}]$ qualquer, precisamos rotacionar um vetor pertencente ao círculo unitário até o ângulo desejado. Sendo o vetor $V = (1, 0)$, ele possui ângulo zero e portanto pode ser rotacionado , resultando em $V' = (\cos\theta, \sin\theta)$. O que precisamos agora é uma forma de obter um algoritmo que rotacione o vetor $V = (1, 0)$ de um ângulo $\theta$, usando elementos de eletrônica digital.

Partindo de uma matriz de rotação no $R^2$, que rotaciona o vetor $V = (x, y)$ para $V' = (x', y')$ de um ângulo $\theta \in [-\frac{\pi}{2}; \frac{\pi}{2}]$. Com um simples passo é possível obter a equação 2.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{1}$$

$$V' = (\cos(\theta))^{-1} \begin{bmatrix} 1 & -\tan(\theta) \\ \tan(\theta) & 1 \end{bmatrix} V \tag{2}$$

Iremos descrever o ângulo $\theta$ em termos de um ângulo $\theta_k = \arctan(2^{-k})$, de forma que $\theta = \sum_{k=0}^{\infty} d_k.\theta_k$ e $d_k = -1$ ou $+1$. A introdução dessa descrição transforma uma única rotação da matriz em uma série de pequenas rotações por $\theta_k$, dando origem a um processo iterativo para a obtenção da rotação desejada por meio de pequenas rotações controladas. Os termos $d_k$ são discutidos posteriormente.

A rotação é reescrita usando o ângulo $\theta_k$, os vetores $V$ e $V'$ são reescritos como $V_k = (x_k, y_k)$ e $V'_k = (x_{k+1}, y_{k+1})$ para representar a ideia de iteração que as rotações adquiriram.

$$V' = A_k \begin{bmatrix} 1 & -2^{-k} \\ 2^{-k} & 1 \end{bmatrix} V \tag{3}$$

$$A_k = (\cos(\arctan(2^{-k})))^{-1} \tag{4}$$

Agora, resolvendo a matriz para explicitar os valores de $x_{k+1}$ e $y_{k+1}$.

$$\begin{cases} x_{k+1} = A_k(x_k - d_k 2^{-k} y_k) \\ y_{k+1} = A_k(y_k + d_k 2^{-k} x_k) \end{cases} \tag{5}$$

O resultado é um algoritmo que consegue descrever os valores assumidos por $x_{k+1}$ e $y_{k+1}$ para cada iteração, ponderados pelos valores de $A_k$. Note que uma outra razão para introdução de $\theta_k$ aparece, o algoritmo necessita apenas de adições e divisões por 2, que são realizáveis usando *shifts* e somadores binários. Para obtermos o CORDIC ainda falta o controle que permite a escolha do ângulo.

Os valores assumidos por $\theta_k$ são : $\theta_0 = 45°$, $\theta_1 = 22.5°$, $\theta_2 = 11.25°$, ..., o que representa a precisão com a qual conseguimos a cada iteração nos aproximar do valor desejado de $\theta$. Os termos $d_k$ representam se para alcançar o ângulo deve-se somar, rotacionar sentido anti-horário, ou subtrair, rotacionar no sentido

horário, durante a iteração do ângulo $\theta_k$. Na literatura o valor do ângulo para cada iteração é comumente chamado de $z_k$ e assim será chamado aqui.

$$z_{k+1} = z_k - d_k \arctan(2^{-k}) \tag{6}$$

A tabela 1 mostra o exemplo para se obter o ângulo $30°$. A cada iteração os valores de $d_k$ são escolhidos para que $z_k$ se aproxime de $30°$.

| **Iteração** | $z_k$ | $d_k$ | $\theta_k$ | $z_{k+1}$ |
|---|---|---|---|---|
| 0 | 0 ° | +1 | 45° | 45° |
| 1 | 45° | -1 | 22.5° | 22.5° |
| 2 | 22.5° | +1 | 11.25° | 33.75° |
| 3 | 33.75° | -1 | 5.625° | 28.125 |
| 4 | 28.125° | +1 | 2.8125° | 30.9375° |

Tabela 1: Tabela descrevendo a obtenção do ângulo de $30°$

Com 4 iterações já se obtém um erro de $0.9375°$, quanto mais etapas menor será o erro.

A etapa final para obter o algoritmo consiste em relacionar $x_{k+1}$ e $y_{k+1}$ com $d_k$ para que cada rotação sofrida pelo ângulo também seja aplicada ao vetor, sucedendo em um algoritmo que é capaz de usar operações simples, para uma FPGA, e obter os valores de seno e cosseno de um ângulo.

Duas correções são necessárias para se obter a forma final do algoritmo usada nesse trabalho. A primeira é sobre o vetor $V$ inicial, como cada rotação do algoritmo introduz uma ponderação por $A_k$, é necessário ponderar o vetor inicial por $\frac{1}{A_k}$, com o intuito de não ter que calcular os termos $A_k$ .

$$\frac{1}{A_K} = \cos(\arctan(2^{-k})) = \sqrt{\frac{1}{1+2^{-k}}} \tag{7}$$

$$\lim_{k\to+\infty} \prod_{i=0}^{+\infty} \sqrt{\frac{1}{1+2^{-k}}} \approx 0.607253 \tag{8}$$

A segunda correção está relacionada a como o algoritmo funciona, foi descrito que partindo do zero itera-se para obter o valor do ângulo desejado, por uma mera questão de praticidade na implementação em FPGA, invés de começar em zero para obter o ângulo, o contrário é feito, ou seja, parte-se do ângulo para chegar em zero. Note que a alteração não acarreta em modificações no algoritmo, apenas no valor inicial do ângulo, $z_0$.

Assim o algoritmo final é expresso abaixo.

No começo da exposição desse algoritmo foi dito que o ângulo $\theta \in [-\frac{\pi}{2}; \frac{\pi}{2}]$, o intervalo restrito deve-se aos valores assumidos por $\theta_k$, qualquer valor fora do primeiro e quarto quadrantes fazem com que o algoritmo não consiga convergir.

O algoritmo citado é capaz de providenciar o valor do cosseno e do seno de um ângulo, mas não é capaz de providenciar um sinal senoidal.

O primeiro problema do CORDIC diz respeito ao domínio de convergência do algoritmo $[-\frac{\pi}{2}; \frac{\pi}{2}]$, para solucionar tal problema precisamos de um pre-processador, que converta qualquer ângulo para seu equivalente no primeiro e quarto quadrante.

A introdução do pre-processador resolve o domínio de convergência do algoritmo, mas introduz uma mudança de domínio que precisa ser corrigida se queremos obter o valor correto do seno e do cosseno do ângulo original. A mudança está relacionada ao sinal do seno e do cosseno, se ele é positivo ou negativo. Por exemplo, um ângulo do segundo quadrante que é convertido para o primeiro, apresenta o sinal correto para o seno, mas o sinal incorreto para o cosseno. Para corrigir isto utiliza-se um pós-processador, que

**Algorithm 1** Algoritmo CORDIC

O vetor $(x_k, y_k)$ é inicializado com o valor fixo que compensa a distorção devido aos termos $A_k$ e o ângulo inicial $z_k$ é inicializado com o valor desejado para se obter o $\sin(\theta)$ e $\cos(\theta)$
O algoritmo é iterado $R$ vezes, dependendo da escolha do usuário.
$x_0 \leftarrow 0,607253$
$y_0 \leftarrow 0$
$z_0 \leftarrow \theta$
for $k = 0$ to $R$ do
  if $z_k > 0$ then
    $x_{k+1} = x_k - 2{-}ky_k$
    $y_{k+1} = y_k + 2{-}kx_k$
    $z_{k+1} = z_k - \arctan(2^{-k})$
  else if $z_k < 0$ then
    $x_{k+1} = x_k + 2{-}ky_k$
    $y_{k+1} = y_k - 2{-}kx_k$
    $z_{k+1} = z_k + \arctan(2^{-k})$
  end if
end for
return $(x_r, y_r, z_r)$
Onde $= (\cos(\theta), \sin(\theta))$ e $z_r = 0$.

conhece o ângulo original, antes do pre-processador, e realiza as correções em relação aos quadrantes originais.

Por fim, é obter-se os ângulos. Isto é alcançado com um acumulador de fase e a ideia de amostragem.

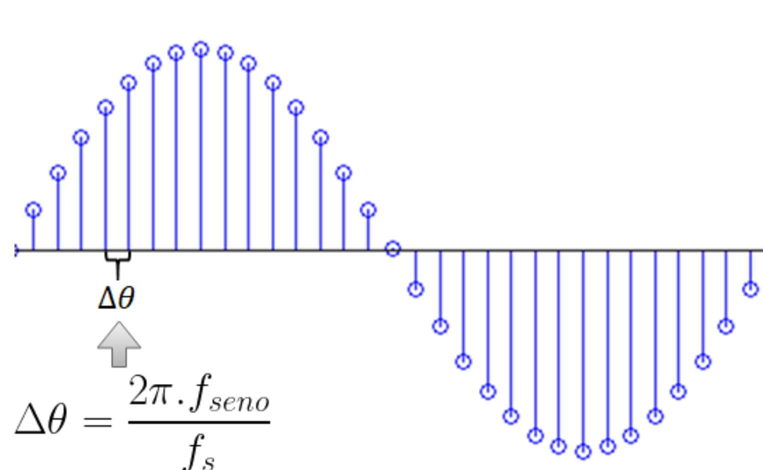



Figura 10: Incremento de fase entre duas amostras

O termo $\Delta\theta_s$, na figura 10, representa a variação de fase entre as amostras do seno, ou cosseno, para uma dada frequência do sinal $f_{seno}$ e uma frequência de amostragem $f_s$. Ao se acumular esse valor repetidamente a cada período de amostragem, obtém-se um gerador de ângulos entre $[0; 2\pi]$, que pode ser usado para produzir um sinal com o CORDIC.

Os nomes usados para cada etapa suplementar são nomes clássicos para quando o algoritmo é usado para a produção de sinais. As etapas foram apresentadas de forma superficial e mais detalhes são apresentados na 3.a.4.

# 3 Desenvolvimento

A presente seção apresenta a arquitetura da solução final, decorrendo sobre os aspectos principais da arquitetura, seu objetivo e seu funcionamento. A última subseção trata da parte dos testes realizados e dos resultados da implementação em FPGA.

## a Arquitetura da Solução

Esta subseção começa com a apresentação da arquitetura da solução desenvolvida em forma de diagrama de blocos e discute sua funcionalidade. Nas partes seguintes são detalhadas as unidades centrais do *design*.

### 1 Visão Geral

A arquitetura apresentada na figura 11 tem dois objetivos, o primeiro é ser uma estrutura capaz de gerar um sinal em tempo real para excitar o transdutor. O segundo objetivo é que a transmissão do sinal e o próprio sinal possam ser configurados, utilizando as interfaces de acesso disponíveis.

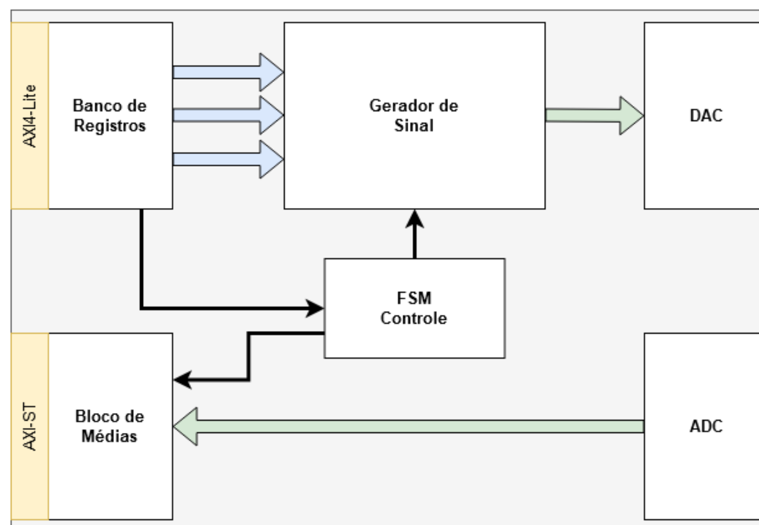



Figura 11: Arquitetura da solução, diagram de blocos

A arquitetura funciona atendendo a configurações e comandos exteriores, realizando a síntese do sinal escolhido e devolvendo o resultado. A configuração e os comandos são acessados pela interface AXI4-Lite que dá acesso ao banco de registros. Esse bloco que agrupa parâmetros em forma de registros, permite que os diferentes blocos do *design* se configurem para gerar o sinal pretendido. Uma vez configurada, a solução aguarda que um sinal de inicio que desencadeia a geração do sinal. O comando é recebido pela FSM (*Finite State Machine*, a máquina de estados finito) de controle que coordena quando os diferentes blocos devem cumprir suas funções. A parte superior do *design* chamada de Tx (Transmissor) recebe todos os parâmetros do banco de registros e o sinal de disparo da FSM, produzindo a cada ciclo de *clock* uma amostra do sinal parametrizado. As amostras são enviadas para o DAC que forma a primeira interface para o transdutor. O resultado da excitação é capturado pelo ADC, a segunda interface para o transdutor, que encaminha as amostras resultantes para o bloco de médias, na parte inferior da figura 11 chamada de Rx (receptor).

Uma das propostas do *design* é realizar a transmissão do mesmo sinal diversas vezes para que um possível ruído captado pelo ADC seja reduzido com o cálculo da média do sinal recebido [28]. A consequência de se gerar diversos disparos do mesmo sinal é a necessidade de guardar o resultado de cada disparo, para o cálculo da média, requerendo um espaço de memória dentro da FPGA, que estoque cada amostra e o resultado do cálculo. Na última etapa, as amostras resultantes recebidas pela solução, após o cálculo da média, precisam agora retornar para entidade que demandou a geração do sinal, uma interface AXI-ST é usada para atender a necessidade de transmissão das amostras. A interface AXI-ST foi escolhida pela praticidade em ser implementada, tendo menos requisitos que uma interface AXI-MM e por ser construída para uma alto fluxo de dados [20].

Os dois cenários referidos anteriormente são atendidos pelo bloco de médias. Essa entidade agrupa uma memória interna, para o cálculo das médias e armazenamento das amostas, possuindo na saída uma interface AXI-ST.

Na descrição do funcionamento foi destacada a função da FSM como sincronizadora das partes Tx e Rx. Essa imposição acontece, pois não existe um controle por parte da solução em qual momento o sinal enviado ao DAC será recebido pelo ADC, devendo um controle interno, baseado em um contador de tempo, indicar o momento correto para que um sinal vindo do ADC seja aceito.

Assim, o banco de registro recebe os parâmetros de transmissão e recepção do sinal, encaminha um comando de inicio para a FSM, que coordena Tx e Rx. Os conversores, o DAC e o ADC, realizam a interface com o transdutor e o bloco de médias age como memória e interface para as amostras resultantes.

Os tópicos seguintes visam destacar o funcionamento individual de cada bloco da arquitetura, descrendo especificidades das implementações e as opções de cada bloco.

## 2 Banco de Registros

O banco de registros é o bloco na interface de entrada da arquitetura. Sua interface de entrada é baseada no protocolo AXI4-Lite [20], parte esquerda da figura 12. O uso do protocolo AXI4-Lite é motivado pela existência de uma interface AXI4-Lite entre PS e PL, discutido na subseção 2.a.3, e, também é motivada, pelo acesso em formato endereço + dados do protocolo AXI-MM.

A construção de uma entidade que seja compatível com os protocolos da família AXI-MM é um trabalho complicado, a interface padrão que compõe um AXI-MM, por exemplo, possui treze sinais diferentes que devem estar em acordo com o protocolo e serem validados entre si. Como o uso de um banco de registros é algo comum no *design* de processadores, existem ferramentas gratuitas dedicadas a construção de bancos de registros baseadas nas descrições de cada registro no esquema de endereçamento, onde cada registro possui um endereço. A ferramenta gratuita *airhdl* [29] foi usada para a geração do código para o bloco banco de registros. O *airhdl* usa o esquema de endereçamento e o tamanho dos registros em bits, para gera uma entidade em VHDL com interface AXI4-Lite.

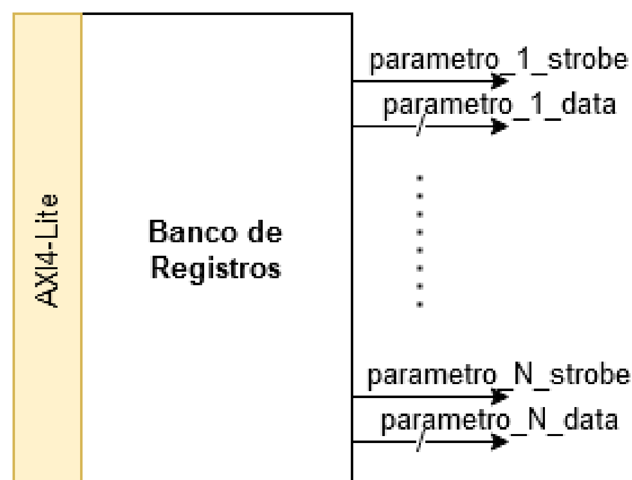



Figura 12: Entidade Banco de Registros

A arquitetura do banco de registros pode ser dividida em duas partes, a primeira, uma FSM que implementa o controle da interface AXI, se encarregando dos acessos de escrita e leitura dos registros. A segunda parte da arquitetura são os registros. Cada processo de escrita/leitura na interface AXI do bloco, com o par endereço + dados, é processada pela FSM da *airhdl* para atender as especificações do protocolo AXI-MM, verificando se o endereço do registro existe e realizando a operação de escrita ou leitura. No caso de uma leitura, a interface AXI de leitura será populada com os dados pertinentes. No caso de uma escrita, o registro correspondente ao endereço terá seu conteúdo sobrescrito pelos novos dados e o registro emitirá um sinal com nível lógico alto, chamado de *strobe*, durante um ciclo de *clock* para indicar que um novo conteúdo está disponível. Esse é o funcionamento obtido ao se utilizar o airhdl, existem outras opções e configurações especificas que podem ser acessadas na ferramenta [29].

A estrutura descrita é a implementação do banco de registros obtido ao se usar o airhdl. O *design* conectado ao banco de registros não faz uso dos sinais *strobes* de cada registro, ao invés disso um registro

chamado *bang*, de um bit, serve como sinal de disparo para o restante dos blocos da arquitetura. O sinal *bang* é um registro com a opção de auto-limpeza, o que significa que ele mantém seu valor como 1 (um) durante um ciclo de *clock*, retornando a zero passado esse ciclo. Uma vez que o usuário tenha configurado todos os parâmetros por meio dos registros dedicados, um processo de escrita no endereço do registro *bang* sinaliza ao restante da arquitetura a validade dos valores nos registros e o inicio da geração de um sinal.

Os parâmetros passíveis de configuração da arquitetura e portanto os registros internos do bloco, servem para configurar os diferentes tempos de controle da FSM de Controle, discutidos na subseção 3.a.3, a quantidade de vezes que o sinal a ser gerado deve ser repetido e parâmetros pertinentes ao sinal a ser gerado. É possível escolher, por exemplo, entre gerar um seno ou um sinal quadrado, no caso do seno, é possível definir a fase inicial, a frequência do seno e o número de períodos. No caso do sinal quadrado, é possível definir a duração de cada degrau e a repetição do sinal descrito na figura 8. A lista de parâmetros abrange mais opções que são específicas as implementações de cada sinal e são discutidos na subseção 3.a.4.

A utilização do método da tabela para geração de um sinal arbitrário 1, é um caso especial do uso do banco de registros e exige uma implementação diferente do banco de registros. A tabela em si é uma memória no interior da FPGA, onde o utilizador escreve as amostras a serem lidas pelo DAC. Nesse caso, não é preciso o esquema de registros citados, a tabela em si comporta o estilo de interface com endereços e dados. O airhdl fornece, entre as suas opções, a capacidade de mapear a interface AXI4-Lite para uma interface memória, adequada ao preenchimento da tabela. Assim, ao se utilizar a tabela, os registros específicos ligados aos outros gerados de sinal são substituído por uma interface memória. A subseção 3.a.4 apresenta formalmente o bloco que implementa a tabela e a interface citada.

O apêndice B traz o mapeamento completo dos registros.

### 3 FSM de Controle

A FSM age como o ponto de controle da arquitetura garantindo a sincronia de envio e recepção do sinal. Como já discutido, a arquitetura não implementa nenhum cálculo interno para saber o tempo de propagação da onda através do transdutor, uma vez que o sinal seja gerado pelo DAC, até o seu retorno ao ADC, para ser amostrada. De fato, o cálculo poderia ser feito na FPGA, mas exigiria recursos e tempo de processamento que podem ser economizados ao realizar-se os cálculos em um ambiente mais adequado e enviar o resultado para a solução.

Não só o tempo de para sincronizar Tx e Rx é enviado a FPGA, são enviados cinco tempos de controle diferentes. A figura 13 destaca as cinco zonas de tempo consideradas e que são executadas em ordem pela FSM.

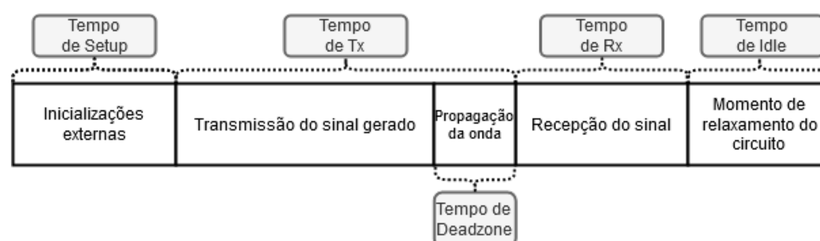



Figura 13: Zonas de tempos

- **Tempo de *Setup*** : Momento de atraso antes da geração do sinal, o tempo para que estruturas externas a FPGA possam se configurar ou iniciar;
- **Tempo de Tx** : O momento de geração do sinal, uma vez nessa zona a FSM indica ao gerador de sinal 4 o inicio da operação. A zona é configurada pelo usuário e ao terminar o tempo da zona qualquer possível saída do gerador de sinal é ignorada;
- **Tempo de *Deadzone*** : Existe um tempo de propagação física ligada ao transdutor, a peça e a eletrônica que deve ser levado em consideração. Durante esse tempo, de *Deadzone*, não é possível fazer nada, pois de um lado a transmissão ainda não se concretizou e, de outro, a recepção ainda não pode começar. Perceba que essa zona depende do tempo de transmissão total (Tempo de Tx), por exemplo, um tempo de Tx de $10\mu s$ e um tempo de *Deadzone* de $2\mu s$, indica que a solução irá gerar a onda durante $10\mu s - 2\mu s = 8\mu s$, reservando o tempo de *Deadzone* para aguardar a propagação da onda;

- **Tempo de Rx** : Marca o momento reservado a parte Rx da solução, indicando que a mesma pode aceitar as amostras do ADC e processar o resultado no bloco de médias;
- **Tempo de *Idle*** : O tempo de *idle* serve para deixar que efeitos remanescentes da propagação da onda, ligados a elemento indutivo/capacitivos ou atrasos, se atenuem ou dissipem, para reiniciar uma nova transmissão;

Em conjunto com as zonas descritas, a FSM comporta, também, a opção de quantas vezes é repetida a geração do mesmo sinal, com os parâmetros já fornecidos.

Para gerar todas as propostas apresentadas, a FSM, representada na figura 14, realiza toda a ordenação das zonas e a opção de repetição.

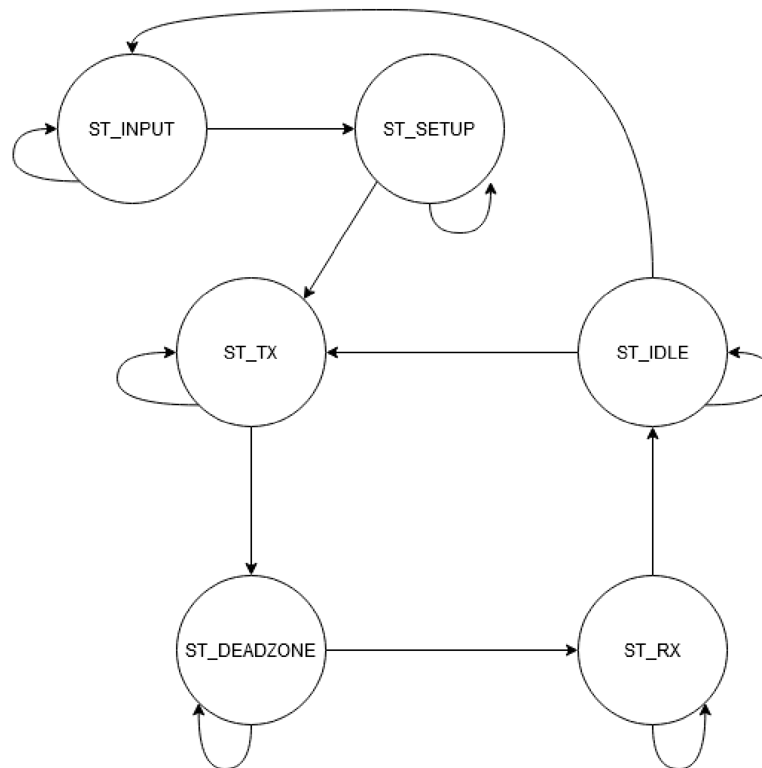



Figura 14: FSM de Controle

- **ST_INPUT** : Estado inicial da FSM, que se repete enquanto não houver um sinal de inicio válido. Um sinal de inicio válido é definido como sendo no mesmo ciclo de *clock* o sinal de inicio proveniente do banco de registro, do utilizador, em nível lógico alto e o sinal interno, chamado de *system_busy*, em nível lógico baixo. O *system_busy* indica que a parte Rx da solução já enviou todas as amostras processadas para o utilizador, evitando, assim, que transmissões diferentes possam se sobrepor. Perante um sinal de inicio válido a FSM transita para o estado ST_SETUP;
- **ST_SETUP** : Estado que comporta a zona de tempo *Setup*, composta de um contador binário que indica se o estado já alcançou o tempo determinado pelo usuário para o estado. Quando o contador indica sua conclusão, a FSM transita para o estado ST_TX. Durante esse estado nada acontece;
- **ST_TX** : Estado que comporta a zona de tempo Tx, diferente do estado anterior, esse estado depende da saída do gerador de sinal e, portanto, o contador binário que compõe o estado inicia a contagem uma vez que uma amostra tenha sido produzida pelo gerador de sinal. Isso é feito, pois entre o instante em que a FSM indica o inicio da geração e o tempo efetivo que a amostra é enviada ao DAC, existe um tempo processamento que depende do sinal sendo gerado. A FSM passa ao estado ST_DEADZONE, uma vez que o contador atinja o tempo indicado pelo usuário para a transmissão, *descontando-se o tempo de deadzone*.
- **ST_DEADZONE** : Estado de aguardo da propagação da onda antes da recepção efetiva. O estado força a solução a ignorar qualquer amostra gerada, sendo necessária atenção na definição do tempo da zona Tx, aguardando que o tempo determinado para a zona se esgote. Com o tempo da zona cumprido, a FSM vai para o estado ST_RX;

- **ST_RX** : O estado de recepção de amostras indica ao bloco de médias que qualquer amostra recebida do ADC, nessa zona de tempo, é válida e deve ser processada. No último ciclo de *clock* do estado, a amostra atual do ADC é marcada como sendo a última do sinal recebido. Isso é feito para garantir que a solução não entre em uma condição onde ela aguarda a última amostra, mesmo depois que a recepção já terminou. O estado seguinte é o ST_IDLE;
- **ST_IDLE** : O último estado da máquina de estados tem duas possibilidades de transição. Caso o número de repetições desejadas da solução tenha sido alcançado, seja ele um, dois, quatro, ou qualquer potência de dois desejada, a máquina de estado transita para o estado ST_INPUT. Caso contrário, a máquina retorna para o estado ST_TX, reiniciando o ciclo descrito. É aqui que a importância do sinal *system_busy* aparece, depois de um ciclo de geração e recepção de sinal concluído, todas as repetições realizadas, a solução entra em uma condição de aguardo que todas as informações coletadas pelo bloco de médias seja transmitida ao usuário, colocando o sinal *system_busy* em nível lógico alto até a conclusão dessa transmissão.

## 4 Gerador de Sinal

O bloco gerador de sinal agrega os diferentes blocos responsáveis pela geração de sinal. A ideia é que o bloco seja montado de acordo com a necessidade do utilizador e serve apenas como referência para sua posição na arquitetura e o modelo de interação com os blocos em seu interior.

É possível instanciar dentro do bloco, um gerador de senos, apelidado de DDS (*Direct digital synthesizer*) CORDIC, um gerador de sinal quadrado, apelidado de *pulser*, ou o bloco tabela para gerar um sinal arbitrário 1. A seleção entre os diferentes tipos de sinal é feito por meio do banco de registros 2. Cada gerador possui seu próprio código (identificador), que o habilita no interior do gerador de sinal. O identificador é transmitido pelo sinal *wave_config*. O bloco recebe acesso a todos os registros de configuração dos sinais e perante um sinal de inicio da FSM de Controle, a geração de sinal começa. A figura 15 mostra uma configuração possível para o gerador de sinal com um DDS CORDIC e um sinal quadrado, a interface de saída apresenta a interface padrão adotada no trabalho para tratar os sinais.

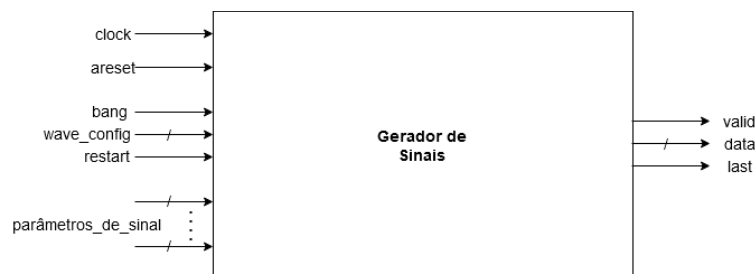



Figura 15: Bloco Gerador de Sinal

A interface para as amostras é composta pelos sinais *valid*, *data* e *last*. O sinal *valid* em estado alto indica que o conteúdo de *data* e o sinal last são válidos no atual ciclo de *clock*. O sinal last indica, em estado alto, que essa é a última transação do ciclo de geração da onda, indicando que a amostra é a última do sinal gerado. A escolha da composição da interface com esses três sinais foi motivada pela sua existência no protocolo AXI-ST. Os sinais *valid* e *data* têm sua importância para sincronizar a transferência entre outros blocos e o sinal *last* possibilita aos blocos dizer quando todas as amostras de um sinal foram recebidas, sem a necessidade de contadores durante etapas de processamento dos dados.

O bloco tabela, apresentado na figura 16, mostra a interface de acesso a memória interna do bloco, proporcionando acesso, endereço a endereço, as posições da memória instanciada em seu interior. Para o processo de geração do sinal armazenado na memória, é necessário fornecer, como parâmetro de configuração, o número de amostras armazenadas. Quando o sinal de inicio é recebido, um contador interno inicia o processo de leitura da memória, colocando na interface de saída as amostras lidas.

Na subseção 2.c.2 foi discutido o método escolhido para geração de um sinal quadrado. Em complemento ao descrito, a solução acrescenta duas opções de modificação do sinal gerado. A primeira modificação é a possibilidade de inverter o sinal gerado, sinal (1) na figura 17. A segunda modificação é a possibilidade de atribuir ao último degrau, $t_5$, um valor diferente de zero, sinal (2) na figura 17. As opções foram chamadas de inversão de pulso e triplo pulso, respectivamente. Existe, ainda, a possibilidade de combinar as duas opções, gerando o sinal (3) na figura 17.

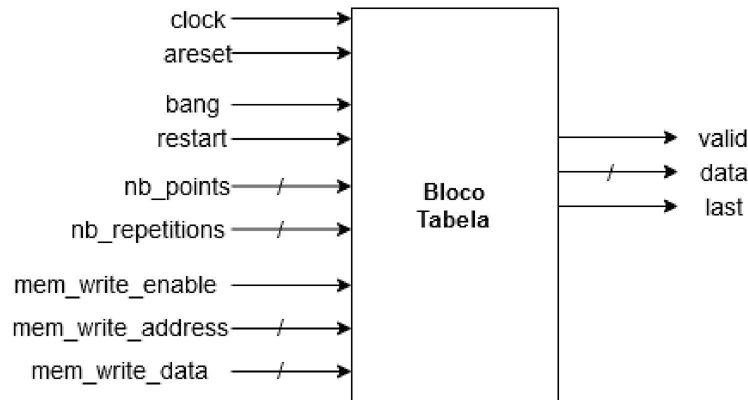



Figura 16: Bloco Tabela

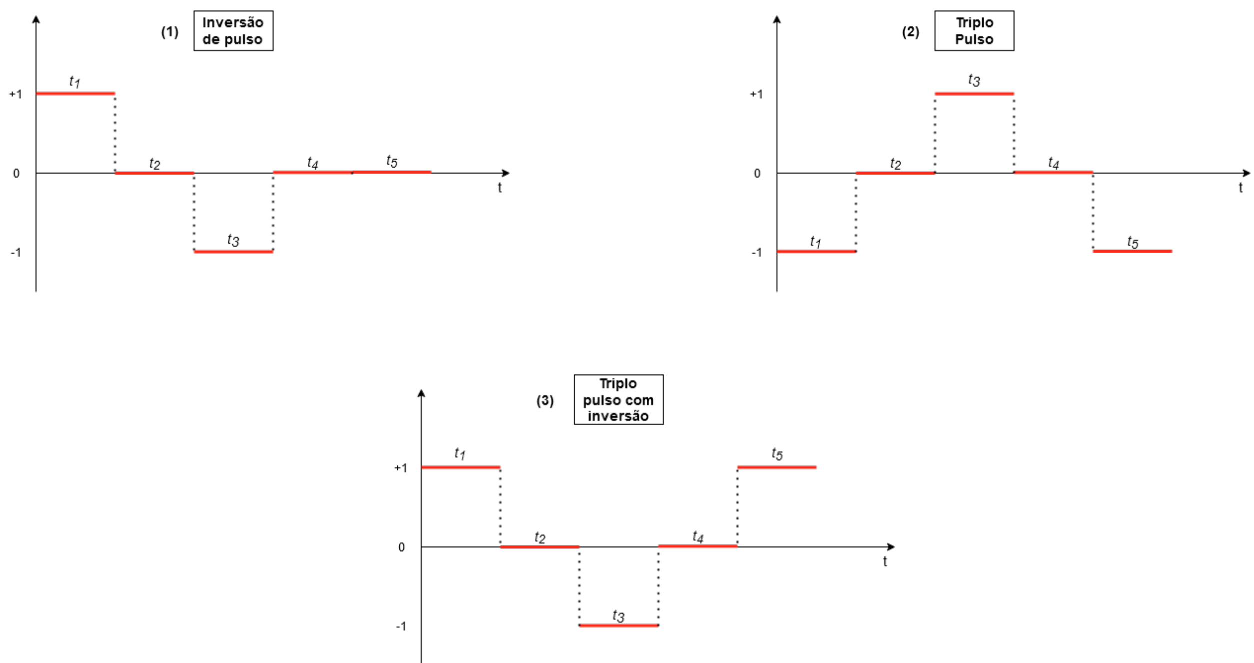



Figura 17: Os tipos de sinal quadrado gerados

As possibilidades apresentadas formam um período do sinal, que pode ser repetido pelo número de vezes que o utilizador desejar. A figura 18 mostras as interfaces do bloco com os diferentes parâmetros aceitos para a configuração. Todos os parâmetros são salvos quando o sinal ”valid” apresenta nível lógico alto e a geração do sinal começa com um nível lógico alto no sinal ”bang”.

A máquina de estados da figura 19 apresenta o controle do *pulser*. Considerando que qualquer estado associado com um degrau pode assumir valor zero, as transições entre estados dependem dos parâmetros fornecidos pelo usuário. No caso em que nenhuma duração de degrau é zero, o diagrama da figura 19 representa a FSM em sua totalidade.

O estado ST_T5 é especial pois ele não pode ser repetido. Para um período do sinal, o degrau em ST_T5 aparece logo depois do estado ST_T4, porém em casos de repetição, com mais de um período, o estado ST_T5 só aparece no final, nunca sendo repetido. Assim, em casos de múltiplos períodos, a FSM se mantém em ciclo entre os estados ST_T1 e ST_T4.

A tabela 2 lista a saída de todos os estados nas diferentes opções possíveis.

Quando qualquer valor é zero, as transições entre estados é feita levando-se em conta a ordem do degraus, como mostrado no exemplo da figura 20. O primeiro estado é sempre o ST_WAIT_BANG que aguarda o

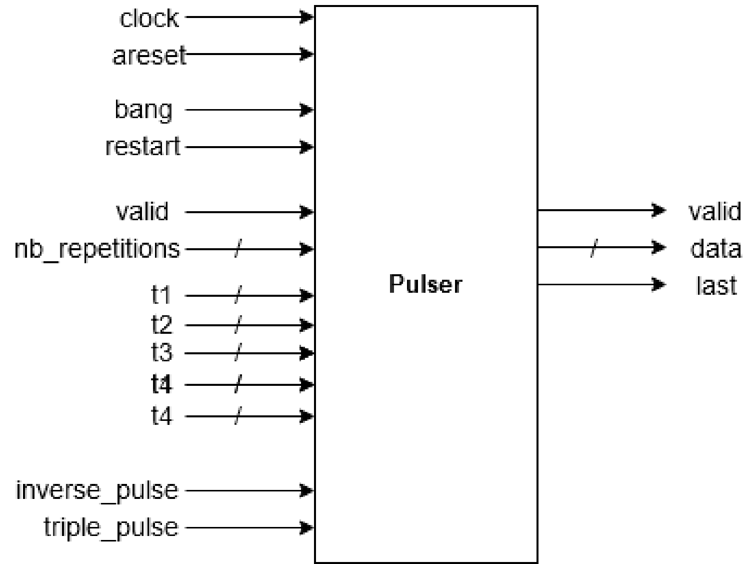



Figura 18: Bloco *pulser*

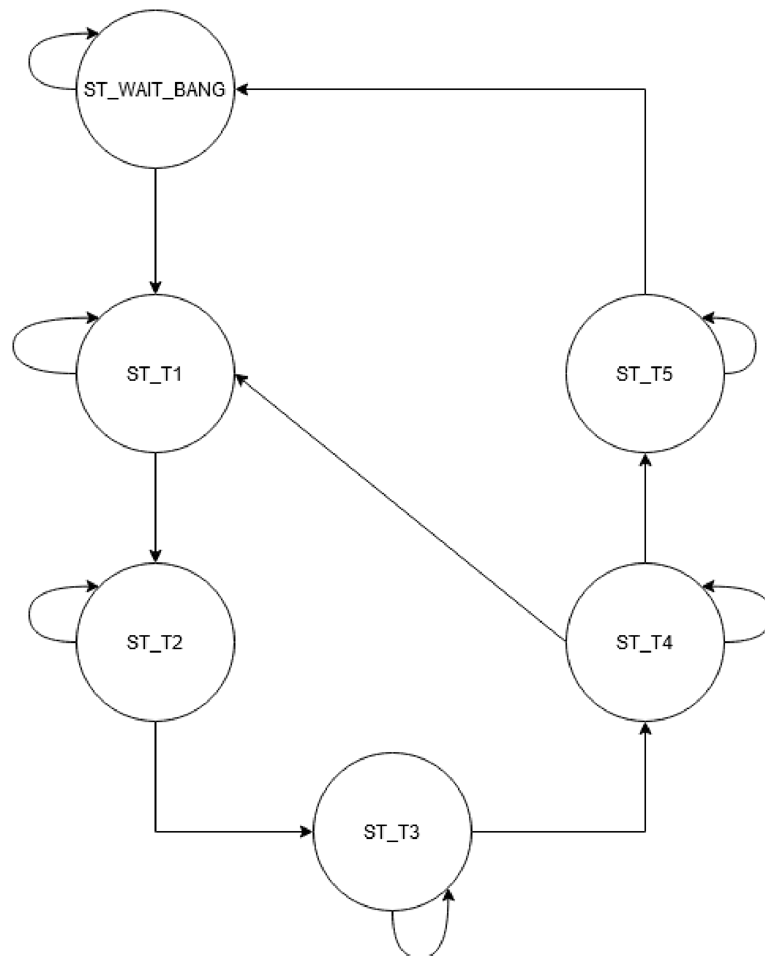



Figura 19: FSM do *pulser*

sinal de inicio para a geração das amostras. Por exemplo, se o primeiro e o terceiro estados, ST_T1 e ST_T3, possuem duração zero, a ordem dos estados é descrita pela figura 20.

O bloco responsável pela geração de senos e cossenos, denominado DDS CORDIC, é a implementação

| Estado | Valor normal | Valor com inversão de pulso | Valor com triplo pulso | Valor com inversão e triplo pulso |
|---|---|---|---|---|
| ST_T1 | -1 | +1 | -1 | +1 |
| ST_T2 | 0 | 0 | 0 | 0 |
| ST_T3 | +1 | -1 | +1 | -1 |
| ST_T4 | 0 | 0 | 0 | 0 |
| ST_T5 | 0 | 0 | -1 | +1 |

Tabela 2: Valores de amplitude dos estados do *pulser* para cada configuração

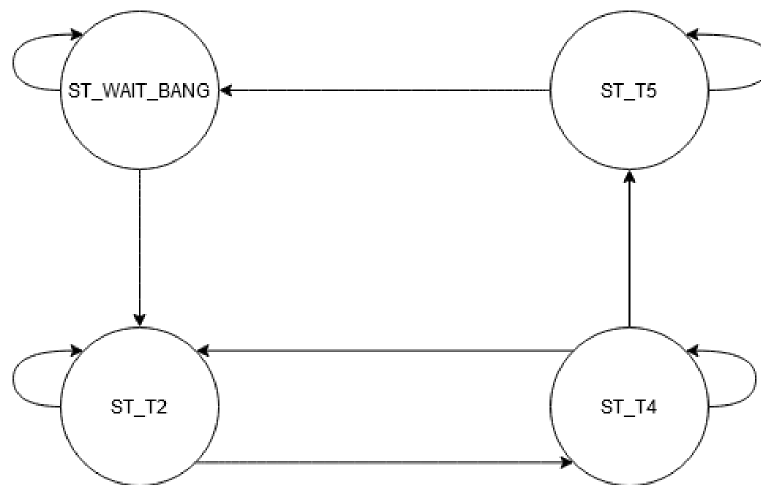



Figura 20: Exemplo da transição de estados, quando o primeiro e o terceiro têm tempo zero.

direta do algoritmo e das etapas de pré e pós processamento, descrita na subseção 2.c.3. A figura 21 mostra a arquitetura interna do DDS CORDIC.

O principal interesse em se gerar um par seno/cosseno é a capacidade de se definir a frequência do sinal gerado. Isso é alcançado com o parâmetro chamado de *phase_term*, que define a variação de fase entre cada amostra do sinal, definida como $"phase_term" = \frac{2\pi.f_{desejada}}{f_{amostragem}}$, em função da frequência desejada para o sinal, $f_{desejada}$, e a frequência de amostragem, $f_{amostragem}$.

A função do acumulador de fase é utilizar a variação de fase fornecida, em conjunto com o número de amostras em um período do sinal, definido como $nb_points = \frac{f_{amostragem}}{f_{desejada}}$, parar gerar, a cada ciclo de *clock*, o ângulo que o CORDIC precisa converter em seno/cosseno.

A convergência do algoritmo CORDIC, entre $[-\frac{\pi}{2}; \frac{\pi}{2}]$, força a existência do bloco pre-processador para mapear qualquer ângulo para seu equivalente no primeiro ou quarto quadrante. Após passar pelo algoritmo, o agora seno e cosseno, precisam ser corrigidos, uma última vez, para retornar aos quadrantes originais do ângulo gerado no acumulador de fase. O mapeamento de ângulos para o primeiro e o quarto quadrante introduz erros de sinais (+,-), que o o bloco pós-processador corrige.

O DDS CORDIC aceita, também, como parâmetros a fase inicial do sinal, a quantidade de períodos que o sinal deve possuir e um modo especial, chamado de ”mode_time”, que converte a fase inicial do sinal em uma diferença temporal para o sinal. A diferença temporal toma a forma $t = -\frac{(fase_{inicial})}{2\pi.f_{amostragem}}$.

De maneira a simplificar a implementação do DDS CORDIC, todas as opções e parâmetros são tratados no acumulador de fase, que se encarrega de gerar a fase correspondente para cada uma das opções do usuário.

Em todos os blocos descritos nessa seção, é possível utilizar o sinal *restart* para reiniciar o processo de geração do sinal, com os últimos parâmetros passados aos blocos. Na solução, esse controle é feito pela FSM para realizar múltiplos disparos dos sinais citados.

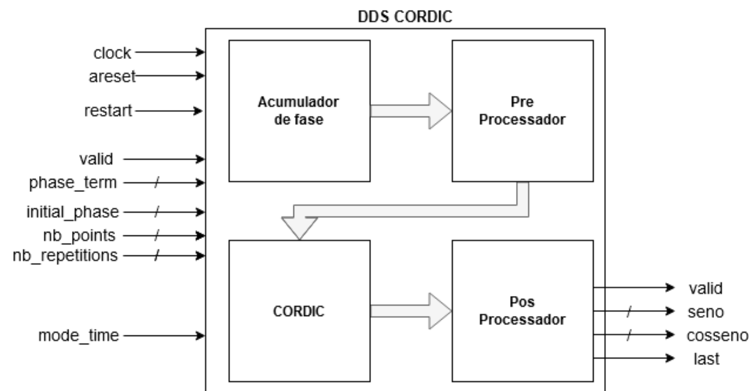



Figura 21: Arquitetura do DDS CORDIC

## 5 Bloco de Médias

Afim de cumprir com as funções de calcular as médias e servir de memória de recepção da solução, o bloco das médias foi implementado baseando-se em uma FIFO (*First In First Out*) circular (ou *buffer* circular) [30]. A base da implementação consiste em ter uma FIFO circular tendo seu tamanho configurável com o número de pontos em um sinal. Assim, é possível acumular o valor de cada amostra na FIFO, sabendo que o número de pontos será manejado naturalmente pela estrutura, e que cada repetição do sinal é acumulada com o último valor salvo na FIFO. A lógica de controle em torno da FIFO garante diferentes gerações sinal não interferem entre si.

A figura 22 mostras as interfaces e os parâmetros do bloco. A interface de entrada é separada em interface de configuração, que permite definir o número de pontos do sinal e quantas vezes ele é repetido, e a interface de dados que recebe o sinal no formato *valid*, *data* e *last*, citado na subseção 3.a.4. Uma escolha da implementação foi fixar o número de repetições como potências de dois, para simplificar o cálculo das médias, já que potências de dois podem ser implementadas como *shifts* binários.

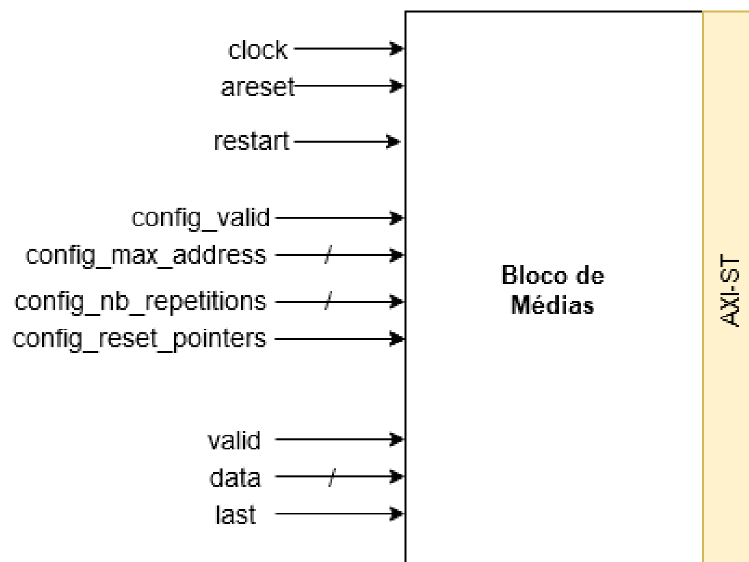



Figura 22: Interfaces do bloco das médias

A interface de saída do bloco possui uma interface AXI-ST e o sinal *sending* que indica que o bloco ainda não enviou todas amostras do sinal.

A diferença principal entre a interface AXI-ST e a interface, *valid*, *data* e *last*, adotada é o sinal *ready* do protocolo AXI-ST, discutido na subseção 2.a.4. A inclusão do sinal *ready* na interface do bloco, junto aos três sinais padrões, incluiu a necessidade de uma estrutura que imita uma FIFO unitária na saída do bloco de médias. A FIFO circular implementada usando BRAMs (*Block* RAM ou Bloco RAM) possui um ciclo de *clock* de latência, entre a leitura de uma amostra e a sua requisição [31]. Uma dificuldade que o valor do

sinal *ready* é avaliado no mesmo ciclo de *clock*, enquanto que a FIFO circular tem sua amostra atrasada de um ciclo. A possibilidade de assincronia levou ao inclusão da FIFO unitária, com um único endereço, que se encarrega de sincronizar as duas interfaces.

### 6 DAC e ADC

Os conversores DAC e ADC são fornecidos pelo CPTI, o circuito físico e o código de controle, portanto ambas as entidades não foram consideradas diretamente no trabalho. As interfaces desenvolvidas foram pensadas para se adequar aos conversores, mas nenhum teste físico foi feito com ambos.

## b Testes e Resultados

Essa subseção apresenta a forma como os testes foram conduzidos, para validação dos geradores de sinal. Uma simulação para toda a arquitetura é apresentada, com o resultado de sua implementação em FPGA. Nenhum teste referente a validação física do trabalho foi realizada.

### 1 Estrutura dos testes e resultado parciais para o DDS CORDIC

A metodologia de testes, para a validação dos blocos em geral, adotou a elaboração primeiro de test-benches [32] e depois a descrição efetiva das entidades e suas arquiteturas, tudo em VHDL. O simulador utilizado foi o QuestaSim 10.6C [33].

O valor de uma amostra, e o quão precisa ela é, está relacionado com a qualidade do sinal. Amostras mais precisas representam sinais com mais fidelidades, entretanto quanto mais precisa uma amostra, mais bits são necessários para sua representação, o que se traduz em mais recursos gastos em uma FPGA. Existe um compromisso entre a precisão e quantidade de recursos usados. Testes realizados com simuladores para HDLs permitem a localização de erros ligados a descrição dos blocos e interações entre diferentes entidades, verificando para cada ciclo de *clock* o comportamento resultante. A verificação da precisão dos sinais foi feita usando um *script* em Python para comparar o sinal gerado com um sinal equivalente gerado usando *floats* de 32 bits. O *script* escrito em Python é capaz de lançar o simulador QuestaSim, executar a simulação de acordo com os parâmetros fornecidos e recuperar as amostras produzidas pelo simulador. Para recuperar as amostras, um módulo em VHDL foi escrito que permitia ao simulador escrever os resultados diretamente em um arquivo texto, que era recuperado pelo *script* em Python para verificação dos resultados.

Todas as operações aritméticas, neste trabalho, consideram aritmética de ponto-fixo [34] para geração das amostras. Como VHDL-93 não suporta nativamente ponto-fixo, foi usada a biblioteca desenvolvida por David Bishop [35] para ponto-fixo.

O gráfico da figura 23 mostra, para frequências de 100 kHz até 500 kHz, o erro médio absoluto máximo para a geração de um seno usando o DDS CORDIC. A referência de comparação é um seno de mesma frequência sendo gerado usando a biblioteca Numpy do Python [36], *float* de 32 bits.

A figura 23 expõe a piora da precisão do algoritmo com o aumento da frequência a ser gerada, para os casos das palavras com 8, 10 e 12 bits de precisão. A configuração do ponto-fixo considera 2 bits para a representação da parte inteira do seno, que só varia entre -1 e +1, e o restante para a parte fracionária. A piora da precisão é explicada pelo menor número de amostras. Quanto maior a frequência a ser gerada, em uma mesma frequência de amostragem, menor será a quantidade de amostras em um período. Quanto menor o número de amostras, maior a diferença de fase entre cada amostra, $\Delta fase = \frac{2\pi.freq_{desejada}}{freq_{amostragem}}$, o que induz um maior erro de acumulação da fase.

Em termos de implementação, os resultados do DDS CORDIC, a quantidade de recursos, são afetados pela precisão do cálculo da fase, pelo tamanho em bits da palavra que representa o seno/cosseno e pelo número iterações que algoritmo é executado. A precisão da fase foi definida como sendo de 32 bits, a escolha foi motivada por ser o tamanho adotada no IP fornecido pela Xilinx® [37]. O tamanho da palavra que representa o seno/cosseno e o número de iterações estão diretamente relacionados. A precisão do algoritmo aumenta em um bit para cada iteração. Para chegar a essa conclusão é necessário observar a forma de $x_{k+1}$ e $y_{k+1}$ na equação 4 da subseção 2.c.3. Cada iteração do algoritmo o termo $2^{-k}$ toma valores que dependem de um só 1 bit, assim a precisão só pode aumentar 1 bit por iteração. A limitação aumenta no quesito de que não é possível ter mais precisão do que o número de bits, significando que se a palavra associado ao seno/cosseno no algoritmo tem **B** bits, a iteração que fornece a máxima precisão para esse caso tem **B** ciclos. O bloco CORDIC foi feito em estilo pipeline [38], em que cada iteração

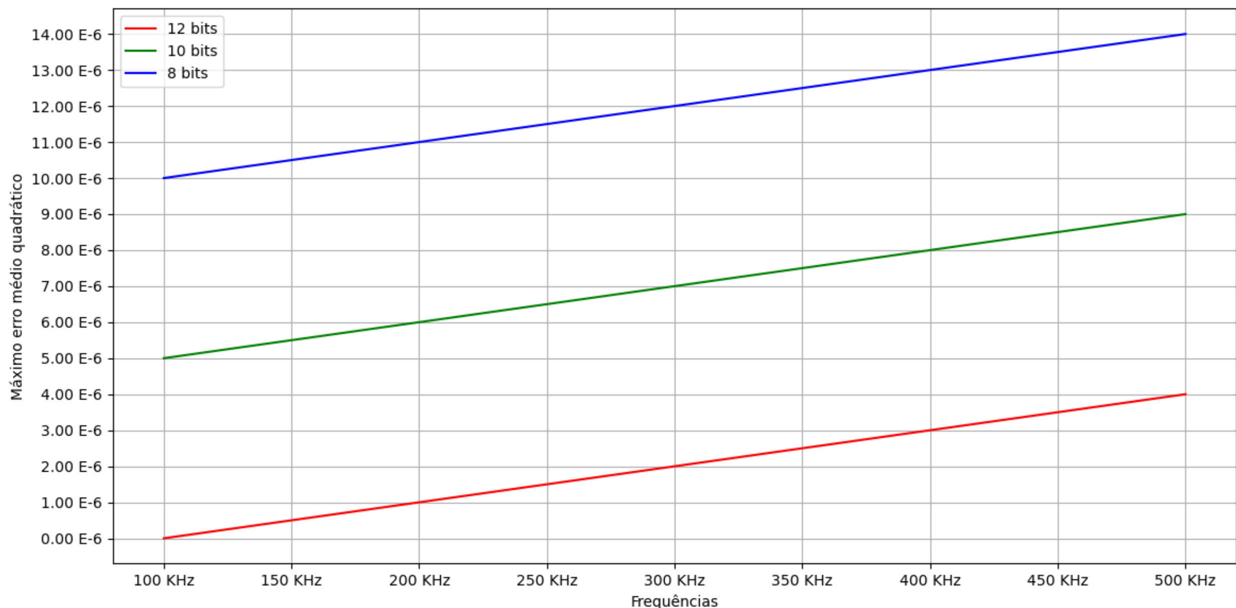



Figura 23: Comparação máximo erro médio absoluto para os casos 8, 10 e 12 bits, no DDS CORDIC.

representa uma cópia do algoritmo exposto na subseção 2.c.3. Desse modo, quanto maior o número de iterações, maior o gasto em recursos para a execução do algoritmo.

Os resultados da figura 23 foram gerados para DDS CORDICs com 32 bits para a fase e com o número de iterações máximos. Os resultados para o DDS CORDIC apresentam a motivação de se testar e comparar os resultados entre uma descrição em HDL e um *script* escrito em uma linguagem de programação de mais alto nível. A validação cruzada confirma que o *hardware* se comporta como esperado, frente a sua versão em *software*.

O *pulser* é baseado em contadores binários, o tornando adequado para testes diretamente em VHDL. O bloco tabela é a implementação de uma memória preenchida pelo usuário, que tem controle da geração das amostras em *software*. Por conta disso, somente os resultados da validação cruzada do DDS CORDIC são enfatizado.

A solução foi pensada como um *design* personalizável, em todos os blocos os tamanhos e definições foram deixadas a cargo do usuário, deixando livre a definição da arquitetura. Um utilizador pode definir o tamanho das memórias usadas, o número máximo de repetições e pontos de um sinal, o tamanho de ponto-fixo das amostras e os tempos máximos usados na FSM de Controle e no *pulser*.

A única restrição, como citado na subseção 3.a.5, é que o número de repetições do sinal gerado, a quantidade de vezes que o mesmo sinal é gerado, deve ser uma potência de dois, para que no bloco de médias a divisão da somatória das médias seja um *shift* lógico.

## 2 Simulação da arquitetura completa

O cenário de simulação considera um acesso externo feito ao banco de registradores, populando os registradores com a configuração do sinal exigido e, por fim, o registrador associado ao bang é escrito, iniciando a geração do sinal. Os conversores, DAC e ADC, não foram simulados, por serem fornecidos pelo CPTI, apenas a estrutura desenvolvida foi considerada nessa simulação.

O texto seguinte mostra o passo a passo da geração de um seno e depois de um *pulser*, para demonstrar as interações entre os diferentes elementos da arquitetura e o impacto de cada parâmetro destacado no apêndice B.

Supondo um utilizador que queira gerar um seno de 500 kHz e um sinal quadrado. Em ambos os casos é necessário fornecer as configurações do sinal, endereços de 0xC até 0x38, e depois os parâmetros específicos de cada sinal, endereço de 0x54 até 0x60 para o seno e 0x3C até 0x50 para o *pulser*.

Na primeira situação, o sinal consiste em três ciclos de seno e será repetido quatro vezes. Essa configuração implica nos valores vistos na tabela 3. O número de pontos do sinal, wave_nb_points, representa o número de pontos em um período do sinal multiplicado pela quantidade de períodos, esse valor é usado pelo bloco de médias para controlar a FIFO circular. A configuração do sinal, wave_config, é zero, indicando que a geração de um seno deve ser feita. Os valores para os tempos de setup, deadzone e idle são somente exemplos e não têm compromisso com valores físicos reais.

O valor para a variação de fase, phase_term, é o resultando da sua expressão, na seção 2.c.3, multiplicado por 2 elevado a 27 (134217728), a variação de fase é um número em ponto-fixo com 27 bits para a sua parte fracionária, tendo ao todo 32 bits. Para converter um número qualquer para seu equivalente em ponto-fixo com N bits, sendo P bits a parte inteira do número e Q bits a parte fracionária, é necessário fazer um *shift* para esquerda de Q bits, ou multiplicar por 2 elevado a Q bits. Essa operação é o equivalente, no sistema decimal, de posicionar a virgula, que separa inteiros e decimais, na posição correta [34].

| Endereço | Nome | Valor (decimal) |
|---|---|---|
| 0xC | wave_nb_periods | 3 |
| 0x10 | wave_nb_points | 600 |
| 0x14 | wave_config | 0 |
| 0x24 | fsm_nb_repetitions | 4 |
| 0x28 | fsm_setup_timer | 100 |
| 0x2C | fsm_tx_timer | 1100 |
| 0x30 | fsm_deadzone_timer | 500 |
| 0x34 | fsm_rx_timer | 600 |
| 0x38 | fsm_idle_timer | 100 |
| 0x54 | dds_phase_term | 4216574 |
| 0x58 | dds_nb_points | 200 |
| 0x5C | dds_init_phase | 0.0 |
| 0x60 | dds_mode | 0 |

Tabela 3: Valores para geração de um seno de 500 kHz com 3 períodos e 4 repetições.

A segunda situação, o *pulser*, repete o requerimento de um sinal com três períodos e quatro repetições, a tabela 4 representa os valores necessários para essa geração. O número de pontos de um período é calculado como a soma dos tempos de 1 até 4, multiplicada pelo número de períodos do sinal e, então, adicionado o quinto tempo, como descrito na subseção 3.a.4.

A simulação inicia com a escrita ao banco de registradores, respeitando o protocolo AXI4-Lite, endereço por endereço, os parâmetros desejados. Em cada ciclo, o endereço de escrita e a palavra associada, tal como expresso nas tabelas, são fornecidos.

Para gerar o seno, os valores da tabela 3 seriam utilizados para configurar a geração. Ao fim da configuração, a geração do sinal é iniciada ao se escrever o valor um no endereço 0x4, o registrador bang. Uma vez que a primeira escrita é feita, a solução irá gerar o sinal configurado e aguardar que ele seja recuperado pelo utilizador no fim do processo. Em toda essa etapa, geração e recuperação, é possível transmitir a solução novos parâmetros de configuração, sem alterar o funcionamento em andamento, mas tentativas de uma nova geração são ignoradas até o fim da recuperação dos dados pelo utilizador.

Com os parâmetros fornecidos para o seno, o gerador de sinal, multiplexa a entrada para que o DDS CORDIC receba os parâmetros e a saída para que o gerador de sinal forneça os resultados vindos do DDS CORDIC.

A FSM de Controle recebendo os parâmetros de tempos e o sinal de inicio, bang, começa a seguir o automato expresso na seção 3.a.3, aguardando o número de ciclos de setup para emitir o sinal de início ao bloco gerador de sinal.

Internamente, o DDS CORDIC recebe os parâmetros para geração de um seno de 500 kHz, ou seja, a variação de fase, o número de pontos em um período, o número de períodos, a fase inicial e o modo de

| Endereço | Nome | Valor (Decimal) |
|---|---|---|
| 0xC | wave_nb_periods | 3 |
| 0x10 | wave_nb_points | 366 |
| 0x14 | wave_config | 1 |
| 0x24 | fsm_nb_repetitions | 4 |
| 0x28 | fsm_setup_timer | 100 |
| 0x2C | fsm_tx_timer | 866 |
| 0x30 | fsm_deadzone_timer | 500 |
| 0x34 | fsm_rx_timer | 366 |
| 0x38 | fsm_idle_timer | 100 |
| 0x3C | *pulser*_t1 | 50 |
| 0x40 | *pulser*_t2 | 25 |
| 0x44 | *pulser*_t3 | 17 |
| 0x48 | *pulser*_t4 | 18 |
| 0x4C | *pulser*_t5 | 36 |
| 0x50 | *pulser*_config | 0 |

Tabela 4: Valores para geração de um sinal quadrado com 3 períodos e 4 repetições.

operação. O acumulador de fase inicia a geração de fase, acumulando a variação de fase, durante um número de ciclos equivalente ao número de pontos em um período. Ao acumular a variação nessa duração, a fase varia de 0 até $2\pi$, formando um período. Ao final, a acumulação retorna a zero, reiniciando, pelo número de períodos solicitados.

Durante a etapa de geração de sinal, a FSM de Controle se encontra no estado ST_TX, por uma questão de sincronia a contagem de tempo do estado aguarda a primeira amostra válida vinda do gerador de sinal. O resultado do DDS CORDIC é emitido pelo gerador de sinal. Na simulação as amostras são transmitidas para uma FIFO, com tamanho configurável, para simular o atraso entre a saída do DAC e a propagação de volta ao ADC, representando o tempo expresso no estado ST_DEADZONE.

Na parte de recepção do sinal, Rx, o estado ST_RX, indica que as amostras sendo recebidas são válidas. Tendo recebido o número de repetições e tamanho do sinal, o bloco de médias, para a primeira repetição, acumula as amostras recebidas na FIFO. O estado de recepção se encerra, indicando que a amostra recebida é a última desse ciclo de funcionamento. Como o número de repetições da geração do sinal é quatro, a FSM de Controle retorna ao estado ST_TX, após o tempo de espera em idle.

O ciclo descrito se repete mais três vezes. A partir da segunda geração, o bloco de médias passa a recuperar os valores salvo na FIFO e acumular com os valores recebidos, para, então, salva-los na FIFO de novo. Ao final do último ciclo de geração, o bloco de médias identifica o fim de todas as repetições e começa a fase de envio do sinal.

O envio é feito pela interface AXI-ST do bloco de médias, respeitando o protocolo AXI-ST, os valores acumulados na FIFO são recuperados e suas médias são calculadas com dois *shifts* para direita, equivalente a dividir as amostras por quatro (número de repetições do sinal gerado nesse exemplo).

Com todas as amostras enviadas, a FSM de Controle libera a solução para receber um novo sinal de inicio, bang. Como já citado, os valores da tabela 4 poderiam ser preenchidos no instante seguinte ao sinal de inicio para a geração do seno. Um sinal de inicio, com os parâmetros para o *pulser* escritos no banco de registradores, inicia a geração do sinal quadrado.

O tipo de sinal muda, wave_config, e, portanto, o gerador de sinal multiplexa a entrada e a saída, para selecionar o *pulser* como alvo da geração. Para a FSM de Controle, o valor dos tempos se modifica, para se adequar aos novos tempos, mas seu funcionamento permanece o mesmo.

A geração de sinal do *pulser*, baseada em uma FSM, é iniciada logo após o sinal de inicio, bang, proveniente da FSM de Controle, passando pelas fases citadas na subseção 3.a.4.

O funcionamento geral da arquitetura é o mesmo que o descrito para o seno, com a diferença que agora a geração de sinal é feito pelo *pulser*. A solução foi pensada dessa forma para que o controle fosse um só e um utilizador pudesse personalizar o sinal a ser gerado, respeitando o estilo de interface empregado no DDS CORDIC, *pulser* e o no bloco tabela.

## 3 Resultados Implementação em FPGA

A implementação em FPGA marca a última etapa antes da programação da placa, para a verificação do funcionamento do circuito físico descrito. É nessa etapa que verifica-se que o *hardware* descrito é capaz de funcionar sobre o *clock* demandado e que a placa possui recursos suficientes para a sua implementação [39].

A seção 3.b.1 mostra os resultados de precisão para diferentes tamanhos do DDS CORDIC, para reforçar o exemplo apresentado, a tabela 5 mostra o resultado da implementação dos diferentes cenários, considerando um *clock* de 100 MHz. A tabela expressa o resultado em função do número de LUTs (*Look-Up Tables*) e FFs (*Flip-Flops*) [40].

Os LUTs são memórias ROM (*Read-Only Memory)* configuradas com valores pré-definidos que podem ser usados, pelo *hardware*, para resolução de operações lógicas e aritméticas em forma de tabela verdade. Os FFs são os registradores comuns utilizados em eletrônica digital [4].

O resultado na tabela 5 mostra que o aumento da palavra não apresenta um crescimento proporcional na utilização de recursos. O resultado é esperado, quanto maior o tamanho da palavra o *software* responsável pela síntese e implementação [39] deve manejar os recursos interiores para se adequar aos novos recursos. Por exemplo, a FPGA considerada (Zybo Z7-20) possui LUTs de 6x1 bits (6 bits de endereçamento e 1 bit de dados) [41], se durante a implementação é detectada a necessidade de se utilizar mais de um LUT para a resolução de uma operação, diferentes LUTs deverão ser concatenados para atender uma única operação. O aumento da complexidade do *design*, pela necessidade de dedicar diversos recursos para uma operação, aumenta de forma não linear o uso de recursos.

| Tamanho da palavra | LUTs | FFs | Utilização dos recursos da FPGA | |
|---|---|---|---|---|
| | | | LUTs | FFs |
| 8 bits | 901 | 428 | 1.69 % | 0.40% |
| 10 bits | 1192 | 534 | 2.24 % | 0.50% |
| 12 bits | 1597 | 664 % | 3.00 % | 0.62 % |

Tabela 5: Resultados da implementação do DDS CORDIC na Zybo Z2-70

Para testar a *implementação* do design proposto, foi considerado um *clock* de 100 MHz, amostras de 10 bits e a memória do bloco de médias possuindo 64K endereços. Com 64k endereços é possível gerar sinais com até 65536 pontos. O tamanho da memória para o bloco de médias foi escolhido para verificar o impacto da utilização de diversos blocos RAMs para a implementação do *hardware*.

A tabela 6 mostra a quantidade de recursos gastos para a implementação do *design*, considerando o número de LUTs, FFs e BRAMs.

| Recurso | Utilização | Taxa de utilização dos recursos da FPGA |
|---|---|---|
| LUT | 1978 | 3.72 % |
| FF | 1436 | 1.35 % |
| BRAM | 30 | 21.43 % |

Tabela 6: Resultados da implementação da solução na Zybo Z7-20, considerando um DDS CORDIC e um *pulser*.

A implementação do *design* mostra que comparativamente poucos recursos são usados da FPGA. Enquanto recursos em termos de LUTs e FFs são pouco usados, a quantidade de BRAMs representa 22%, sendo o caminho crítico para implementação do *design*.

O caminho crítico em um circuito digital representa um caminho combinatório entre dois registradores, onde, em todo um *design*, encontra-se o maior tempo de propagação da informação. É com o tempo do

caminho crítico que se define se um circuito consegue funcionar com uma certa frequência de *clock*. O tempo máximo que permite o funcionamento de um circuito sequencial é o período de um *clock*, então todas as operações combinatórias entre dois registradores devem levar no máximo, um período de *clock* para acontecer, ou serem divididas em múltiplos ciclos. Se o tempo do caminho crítico é superior a esse limite, estabelecido pela frequência de *clock*, o mesmo não pode funcionar com tal frequência, necessitando frequências mais baixas para funcionar. O capítulo 5 da referência [42] mostra um discussão mais aprofundada sobre o tema e seu impacto diretamente em uma FPGA.

As BRAMs são distribuídas sobre a placa com o intuito de proporcionar a todo o *design* acesso a esse recurso. Outro motivo da distribuição é a redução do tempo de propagação física do impulso elétrico entre os diferentes componentes da FPGA. No resultado de implementação, da solução na Zybo Z7-20, o maior caminho crítico é encontrado entre os contadores binários da FIFO circular, no interior do bloco de médias, e as portas de endereçamento das BRAMs, demonstrando a importância do recurso e o impacto sobre o desempenho do circuito.

# 4 Trabalhos Futuros

As opções de trabalhos futuros representam partes que foram em algum momento pensadas para o projeto, mas não foram realizadas e adições para tornar a solução mais rica. Cada tópico é apresentado e discutido brevemente, referências são fornecidas para exemplificar o assunto tratado.

O trabalho futuro com maior importância é o teste do trabalho desenvolvido direto em uma FPGA, para verificar em um primeiro momento que o circuito descrito funciona como esperado. Uma vez validado o funcionamento do circuito em FPGA, os testes passam para etapa com o circuito de amplificação e o transdutor, para tornar a solução aplicável. Um exemplo de uma situação teste que se aproxima do pretendido é encontrado em [2]. A implementação na Zybo Z7-20 teria, ainda, que passar por uma etapa de desenvolvimento da conexão entre PS-PL utilizando as interfaces citadas na subseção 2.a.4 e um *software* de controle para a solução.

Ao observar a analise em frequência do seno gerado com a solução são notadas componentes de frequências fora da frequência desejada, que podem vir a influenciar os resultados do transdutor. A figura 24 mostra o exemplo para um seno de 500 kHz, 10 períodos e 10 bits de precisão. Duas opções foram cogitadas para aumentar a distância entre a componente de frequência principal, 500 kHz no exemplo, e a segunda componente de maior amplitude. A primeira consiste em explorar uma nova arquitetura para o CORDIC que melhora sua precisão [43]. A segunda proposta cogitada foi o uso de janelas [44] para limitar a resposta em frequência do sinal, em especial as janelas do tipo somas de cossenos [45]. Ao se utilizar as janelas com soma de cossenos, o desenvolvimento feito com o CORDIC poderia ser reaproveitado para gerar os algoritmos referentes a cada janela.

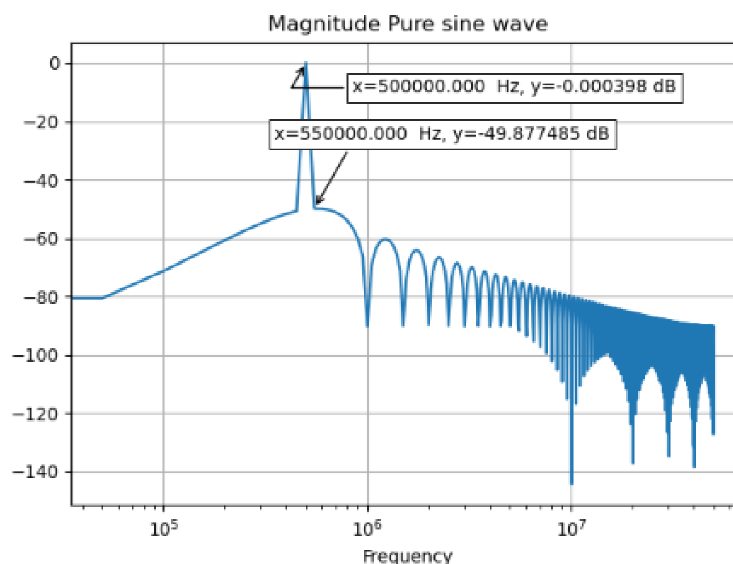



Figura 24: Resposta em frequência de um seno 500 kHz com 10 períodos e 10 bits de precisão

A última parte observada durante a realização do trabalho que poderia sofrer mudança é a parte de amostras do sinal. Até aqui foi considerado que sempre seriam produzidos sinais amostrados em 100 MHz, a frequência desejada de funcionamento para a FPGA do projeto, portanto os conversores, DAC e ADC, trabalham com essa mesma frequência de amostragem. Quanto maior a frequência de amostragem mais amostras são geradas por período, o que leva a necessidade de mais componentes de memória, BRAMs, na arquitetura e como exposto na subseção 3.b.3, esse é o principal problema para a implementação do *design*. A opção pensada consistiria em não modificar a geração dos sinais, ou seja, manter a geração como se a frequência de amostragem fosse de 100 MHz, mas utilizar técnicas de processamento de sinal para reduzir a frequência de amostragem do sinal, podendo aumentar ou reduzir, para se adequar a frequências mais baixas de amostragem. A técnica daria mais flexibilidade ao utilizador de escolher a frequência de amostragem independente da frequência de funcionamento da FPGA. A técnica para reduzir é chamada de decimação e para aumentar interpolação [46].

# 5 Conclusão

O trabalho alcança seu objetivo de explorar uma solução implementável em FPGA para a produção de sinais digitais em FPGA. Uma futura etapa desse trabalho possibilitaria o estudo dos impactos dos sinais nos transdutores, culminando na elaboração de melhores sistemas de inspeção. Os conhecimentos empregados neste trabalho, desde o estudo da plataforma digital, da linguagem VHDL, das interfaces e protocolos de comunicação, dos métodos de geração de sinais e as discussões geradas em torno do funcionamento geral da solução e seus trabalhos futuros, serviram de base para a construção não só da solução, mas de uma base de conhecimento para acessar uma área de desenvolvimento em sistemas digitais que vem crescendo no mundo, principalmente, nas áreas de DSP (*Digital Signal Processing*) [47] e inteligência artificial [48].

## 6 Referências

[1] "Amba axi protocol," http://referencedesigner.com/blog/amba-axi-protocol/2567/, acessado em 29 de outubro de 2020 às 20h30.

[2] B. Ren and J. Xin, "In-line inspection of unpiggable buried live gas pipes using circumferential emat guided waves," in *AIP Conference Proceedings*, vol. 1949, no. 1. AIP Publishing LLC, 2018, p. 020019.

[3] M. Clough, M. Fleming, and S. Dixon, "Circumferential guided wave emat system for pipeline screening using shear horizontal ultrasound," *NDT & E International*, vol. 86, pp. 20–27, 2017.

[4] A. K. Maini, *Digital electronics: principles, devices and applications*. John Wiley & Sons, 2007.

[5] "Zybo z7board reference manual," https://reference.digilentinc.com/_media/reference/programmable-logic/zybo-z7/zybo-z7_rm.pdf, acessado em 15 de Novembro de 2020 às 22h30.

[6] "What is an fpga? an introduction to programmable logic," https://www.allaboutcircuits.com/technical-articles/what-is-an-fpga-introduction-to-programmable-logic-fpga-vs-microcontroller/, acessado em 15 de Novembro de 2020 às 23h00.

[7] Xilinx, "Ug909 - vivado design suite user guide, partial reconfiguration," Xilinx, Tech. Rep., 3 2018, páginas: 6-10.

[8] "Fpga vs. asic: Differences and choosing best for your business," https://resources.pcb.cadence.com/blog/2019-fpga-vs-asic-differences-and-choosing-best-for-your-business, acessado em 15 de Novembro de 2020 às 23h30.

[9] D. L. Perry, *Vhdl*. McGraw-Hill, Inc., 1993.

[10] IEEE., *IEEE Standard VHDL Language Reference Manual: IEEE Std 1076-1993*. IEEE Press, 1993.

[11] I. D. A. S. Committee *et al.*, "Std 1076–2008, ieee standard vhdl language reference manual," *IEEE, New York, NY, USA*, 2008.

[12] ——, "Ieee standard for vhdl language reference manual," *IEEE Std 1076-2019*, pp. 1–673, 2019.

[13] "Myhdl," http://www.myhdl.org/, acessado em 15 de Novembro de 2020 às 23h52.

[14] "Chisel/firrtl hardware compiler framework," https://www.chisel-lang.org/, acessado em 15 de Novembro de 2020 às 23h56.

[15] "Accellera - systemc," https://www.accellera.org/downloads/standards/systemc, acessado em 15 de Novembro de 2020 às 23h58.

[16] J. McDougall, "Simple amp: bare-metal system running on both cortex-a9 processors," *Xilinx, San Jose, CA, USA, XAPP1079 (v1. 0.1)*, 2014.

[17] Xilinx, "Ug1144 - petalinux tools documentation," Xilinx, Tech. Rep., 12 2017.

[18] "Master-slave model," https://www.ibm.com/support/knowledgecenter/ssw_aix_72/generalprogramming/master_slave_model.html, acessado em 16 de Novembro de 2020 às 00h12.

[19] Arm, "Amba axi and ace protocol specification," Arm Holdings, Tech. Rep., 4 2020.

[20] ——, "Amba 4 axi4-stream protocol specification," Arm Holdings, Tech. Rep., 3 2010.

[21] "The scientist and engineer's guide to digital signal processing - chapter 3: Adc and dac -the sampling theorem," http://www.dspguide.com/ch3/2.htm, acessado em 16 de novembro de 2020 às 00h30.

[22] C. W. Brokish and M. Lewis, "A-law and mu-law companding implementations using the tms320c54x," *SPRA163*, 1997.

[23] M. Rabiee, "Analog to digital (adc) and digital to analog (dac) converters," *age*, vol. 3, p. 1, 1998.

[24] "Two's complement representation: Theory and examples," https://www.allaboutcircuits.com/technical-articles/twos-complement-representation-theory-and-examples/, acessado em 24 de Novembro de 2020 às 14h00.

[25] A. Church, "Edward f. moore. gedanken-experiments on sequential machines. automata studies, edited by c. e. shannon and j. mccarthy, annals of mathematics studies no. 34, litho-printed, princeton university press, princeton1956, pp. 129–153." *Journal of Symbolic Logic*, vol. 23, no. 1, p. 60–60, 1958.

[26] G. H. Mealy, "A method for synthesizing sequential circuits," *The Bell System Technical Journal*, vol. 34, no. 5, pp. 1045–1079, 9 1955.

[27] R. Andraka, "A survey of cordic algorithms for fpga based computers," in *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, 1998, pp. 191–200.

[28] "Use signal averaging to increase the accuracy of your measurements," https://www.allaboutcircuits.com/technical-articles/use-signal-averaging-to-increase-the-accuracy-of-your-measurements/, acessado em 25 de Novembro de 2020 às 09h15.

[29] "airhdl - register management done rigth." https://airhdl.com, acessado em 25 de Novembro de 2020 às 23h30.

[30] "The scientist and engineer's guide to digital signal processing - chapter 28: Circular buffering," http://www.dspguide.com/ch28/2.htm, acessado em 01 de dezembro de 2020 às 16h40.

[31] Xilinx, "Ug473 - 7 series fpgasmemory resources, user guide," Xilinx, Tech. Rep., 7 2019, capítulo 1.

[32] M. Hamid, "Application note (xapp199): Test benches - writing efficient testbenches," Xilinx, Tech. Rep., 5 2010, v1.1.

[33] "Questa advanced simulator," https://www.mentor.com/products/fv/questa/, acessado em 7 de Novembro de 2020 às 16h40.

[34] R. Yates, "Fixed-point arithmetic: An introduction," *Digital Signal Labs*, vol. 81, no. 83, p. 198, 2009.

[35] D. Bishop, "Fixed point package user's guide," *Packages and bodies for the IEEE*, pp. 1076–2008, 2006.

[36] S. v. d. Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: a structure for efficient numerical computation," *Computing in science & engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[37] D. Compiler, "v6. 0 logicore ip product guide," 2015.

[38] "The why and how of pipelining in fpgas," https://www.allaboutcircuits.com/technical-articles/why-how-pipelining-in-fpga/, acessado em 10 de Dezembro de 2020 às 14h20.

[39] Xilinx, "Ug888 - vivado design suite tutorial, design flows overview," Xilinx, Tech. Rep., 12 2018.

[40] ——, "Ug474 - 7 series fpgas configurable logic block, user guide," Xilinx, Tech. Rep., 9 2016.

[41] S. J. Xilinx, "Zynq-7000 all programmable soc overview (ds190 v1. 8)," 2015.

[42] Xilinx, "Ug906 - vivado design suite user guide, design analysis and closure techniques," Xilinx, Tech. Rep., 6 2020, capítulo 5.

[43] M. Jridi and A. Alfalou, "Direct digital frequency synthesizer with cordic algorithm and taylor series approximation for digital receivers," 2009.

[44] F. J. Harris, "On the use of windows for harmonic analysis with the discrete fourier transform," *Proceedings of the IEEE*, vol. 66, no. 1, pp. 51–83, 1978.

[45] H.-H. Albrecht, "A family of cosine-sum windows for high-resolution measurements," in *2001 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No. 01CH37221)*, vol. 5. IEEE, 2001, pp. 3081–3084.

[46] R. E. Crochiere and L. R. Rabiner, "Interpolation and decimation of digital signals—a tutorial review," *Proceedings of the IEEE*, vol. 69, no. 3, pp. 300–331, 1981.

[47] "Using fpgas to solve tough dsp design challenges," https://www.eetimes.com/using-fpgas-to-solve-tough-dsp-design-challenges/, acessado em 15 de Dezembro de 2020 às 00h30.

[48] G. Lacey, G. W. Taylor, and S. Areibi, "Deep learning on fpgas: Past, present, and future," *arXiv preprint arXiv:1602.04283*, 2016.

# A Exemplo VHDL

```
1   -- Isso é um comentário em VHDL
2   -- Autor : Lucas Grativol Ribeiro
3
4   library ieee; -- biblioteca padrão que define os tipos usados
5   use ieee.std_logic_1164.all; -- grupo de definições do tipo "std_logic"
6
7   -- std_logic age como um fio em VHDL, carregando bits de um lado ao outro
8
9   entity porta_and_reg is -- define as entradas e saídas do bloco
10          port (
11                  clock  : in  std_logic; -- clock do design, como porta de entrada
12                  reset  : in  std_logic; -- sinal de reset do design
13
14                  input1 : in std_logic; -- entrada 1 da porta AND
15                  input2 : in std_logic; -- entrada 2 da porta AND
16
17                  output : out std_logic -- saída da entidade
18          );
19  end porta_and_reg;
20
21  architecture rtl of porta_and_reg is --define o comportamento (arquitetura) do design
22
23          signal porta_and : std_logic; -- um sinal (fio/bit) interno do design
24          signal reg       : std_logic; -- um sinal (fio/bit) interno do design
25
26  begin
27
28          -- Porções de código fora de processos tem comportamento concorrente
29          porta_and <= input1 and input2; -- O AND entre os dois sinais de entrada
30
31          -- Porções de código dentro de processos tem comportamento sequencial
32          process(clock)
33          begin
34                  if (rising_edge(clock)) then -- Durante uma transição positiva do clock
35                          if (reset = '1') then
36                                  reg <= '0'; -- Reseta o valor do registrador
37                          else
38                              -- Guarda o valor da porta AND no registrador
39                                  reg <= porta_and;
40                          end if;
41                  end if;
42          end process;
43
44          output <= reg; -- Conecta o registrador com a saída
45  end architecture;
```

## B Mapeamento dos registradores

| Offset | Nome | Descrição | Tamanho | Acesso | Reset |
|---|---|---|---|---|---|
| 0x0 | version | Número da versão do código.<br>Usado para controle de versão. | 8 bits | READ_ONLY | 0x0 |
| 0x4 | bang | Botão de "start", com auto-reset,<br>imitando um tick. | 1 bit | WRITE_ONLY | 0x0 |
| 0x8 | sample_frequency | Frequência de amostragem.<br>Valor predefinido de 100 MHz. | 27 bits | READ_WRITE | 0x5F5E100 |
| 0xC | wave_nb_periods | Número de períodos do sinal<br>a ser gerado. | 8 bits | READ_WRITE | 0x0 |
| 0x10 | wave_nb_points | Número de pontos em um período.<br>Usado no DDS CORDIC. | 32 bits | READ_WRITE | 0x0 |
| 0x14 | wave_config | Seletor do sinal desejado.<br>"0" ->DDS CORDIC ; "1" ->*pulser* | 1 bit | READ_WRITE | 0x0 |
| 0x24 | fsm_nb_repetitions | Número de repetições do sinal<br>a ser gerado, controlado pela FSM. | 6 bit | READ_WRITE | 0x0 |
| 0x28 | fsm_setup_timer | Número de ciclos na zona Setup | 18 bits | READ_WRITE | 0x0 |
| 0x2C | fsm_tx_timer | Número de ciclos na zona Tx | 18 bits | READ_WRITE | 0x0 |
| 0x30 | fsm_deadzone_timer | Número de ciclos na zona Deadzone | 18 bits | READ_WRITE | 0x0 |
| 0x34 | fsm_rx_timer | Número de ciclos na zona Rx | 18 bits | READ_WRITE | 0x0 |
| 0x38 | fsm_idle_timer | Número de ciclos na zona Idle | 18 bits | READ_WRITE | 0x0 |
| 0x3C | *pulser*_t1 | Tamanho do degrau em t1 | 10 bits | READ_WRITE | 0x0 |
| 0x40 | *pulser*_t2 | Tamanho do degrau em t2 | 10 bits | READ_WRITE | 0x0 |
| 0x44 | *pulser*_t3 | Tamanho do degrau em t3 | 10 bits | READ_WRITE | 0x0 |
| 0x48 | *pulser*_t4 | Tamanho do degrau em t4 | 10 bits | READ_WRITE | 0x0 |
| 0x4C | *pulser*_t5 | Tamanho do degrau em t5 | 10 bits | READ_WRITE | 0x0 |
| 0x50 | *pulser*_config | Sinal de configuração para as opções<br>de pulso invertido e pulso triplo.<br>[0] inverted_pulse<br>[1] triple_pulse | 2 bits | READ_WRITE | 0x0 |
| 0x54 | dds_phase_term | DDS CORDIC phase term | 32 bits | READ_WRITE | 0x0 |
| 0x58 | dds_nb_points | Número de pontos em 1 período do seno | 18 bits | READ_WRITE | 0x0 |
| 0x5C | dds_init_phase | DDS CORDIC initial phase | 32 bits | READ_WRITE | 0x0 |
| 0x60 | dds_mode | DDS CORDIC opção mode_time | 1 bit | READ_WRITE | 0x0 |

Tabela 7: Exemplo mapeamento de registradores, endereço inicial 0x00